Push-button verification of Files Systems via Crash Refinement

Verification Primer

- Behavioral
  - Specification and implementation are both programs
  - Equivalence check proves the functional correctness
- Hoare logic
  - Functional Specification are the preconditions and postconditions
- More ways e.g. DSL etc

```
int add(int a, int b)
{
    return a+b;
}
```

Test function

```
int spec_add(int a, int b)
{
    return a $+_{2scomp}$ b;
}
```

Behaviour specification

Pre: bitvector32 : a
        bitvector32 : b

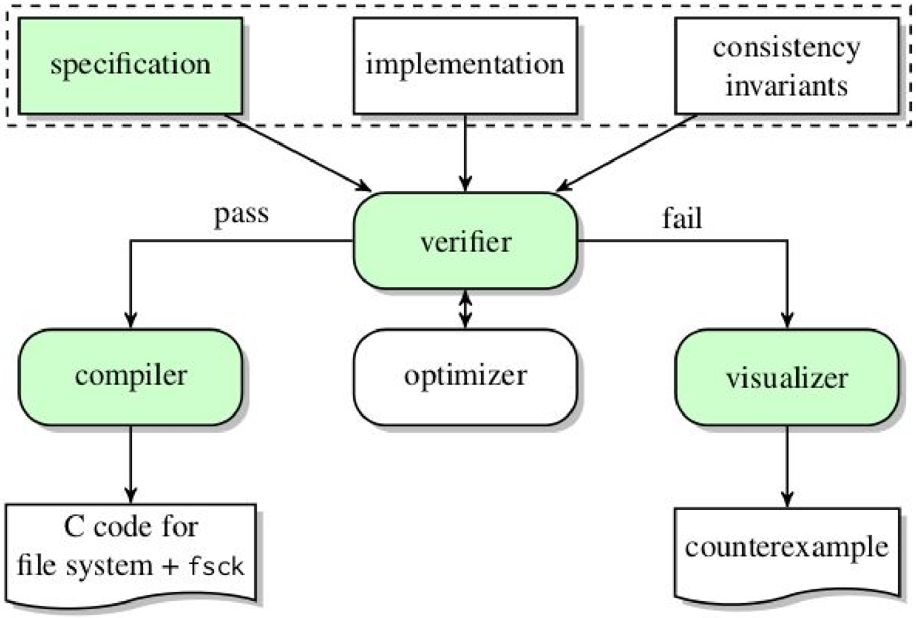
Post: return =
bvadd2scomp(a, b)

Hoare specification

Problem

- Verification of file system
    - Push-button i.e. automatic
    - No manual annotations or proofs
        - BilbyFs took 9.25 months, 13K LOP for 1350 LOC
        - FSCQ took 1.5 years, code size is 10x of xv6-fs
    - Functional correctness (stronger than the consistency requirement)
- What is special about file system verification?
    - Crash and recovery procedure
    - Reordering of writes

Overview of the technique



- Input: specification, implementation and consistency invariants
- Trusted components: specification, verifier, compiler and visualizer

Yggdrasil toolkit

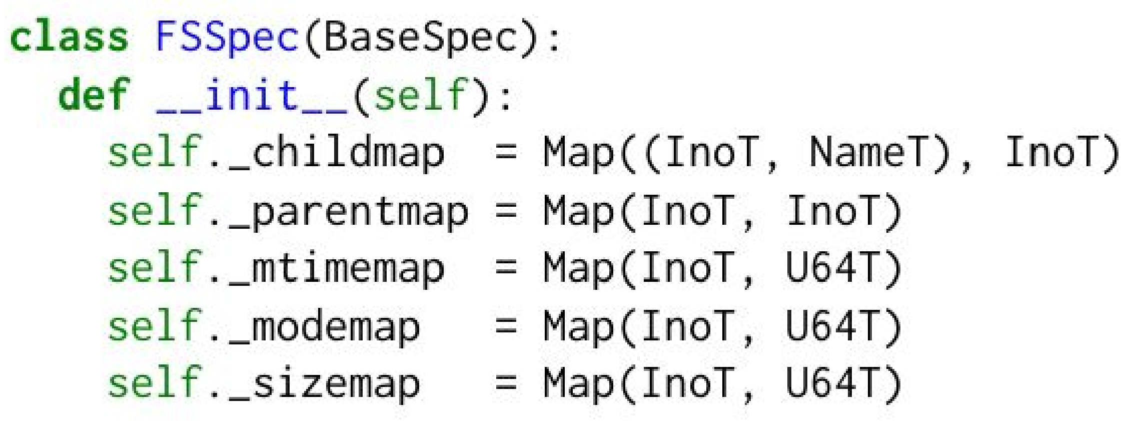
- Specification, implementation and consistency invariants are specified in a subset of python
- Counter examples are given when:
    - specification != implementation
    - consistency invariants do not hold
- Support for optimizations in the implementation e.g. disk flushes
- After verification it emits C code for the filesystem and fsck utility.

Example: YminLFS

- Simplified log-structured file system
- Development took less than four hours
- Even caught two bugs in the initial implementation

YminLFS: Specification

- **Abstract data structure**
- Operations
- Equivalence predicate



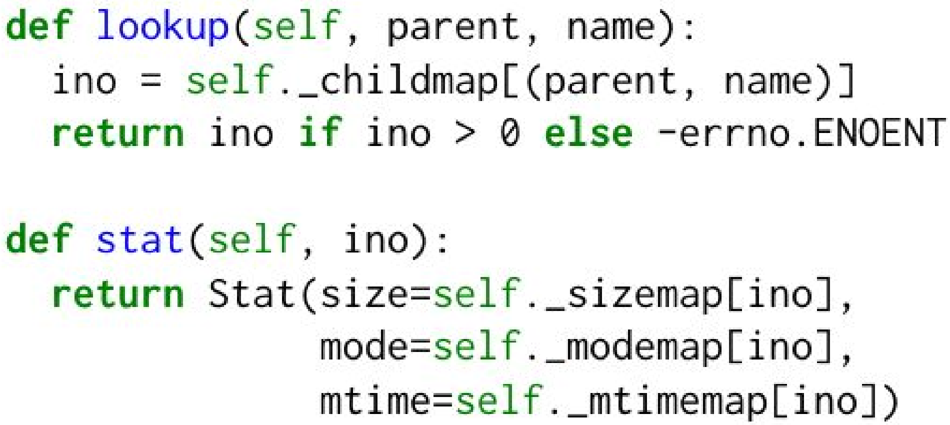
```
class FSSpec(BaseSpec):
  def __init__(self):
    self._childmap  = Map((InoT, NameT), InoT)
    self._parentmap = Map(InoT, InoT)
    self._mtimemap  = Map(InoT, U64T)
    self._modemap   = Map(InoT, U64T)
    self._sizemap   = Map(InoT, U64T)
```

Dir-inode * file-name -> file-inode
inode -> parent-inode
inode -> mtime-stat
inode -> mode-stat
inode -> size-stat

- Abstract maps
- Abstract types: **InotT**, **U64T** are 64-bits integers and **NameT** is a string type

YminLFS: Specification

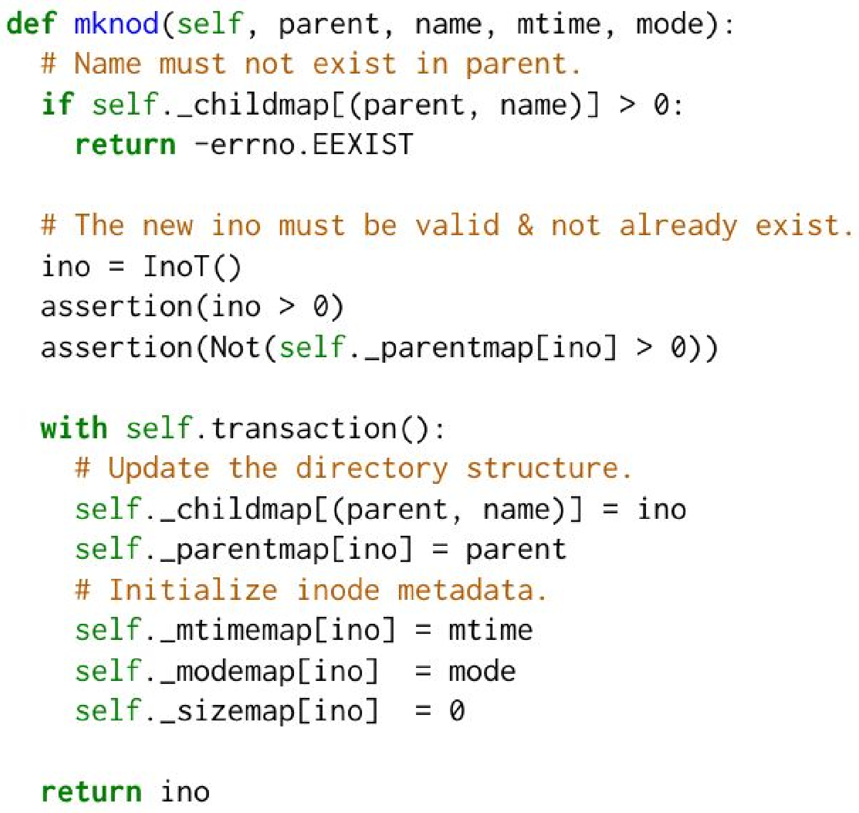
- Abstract data structure
- **Operations**
- Equivalence predicate



```
def lookup(self, parent, name):
  ino = self._childmap[(parent, name)]
  return ino if ino > 0 else -errno.ENOENT

def stat(self, ino):
  return Stat(size=self._sizemap[ino],
              mode=self._modemap[ino],
              mtime=self._mtimemap[ino])
```

YminLFS: Specification

- Abstract data structure
- **Operations**
- Equivalence predicate

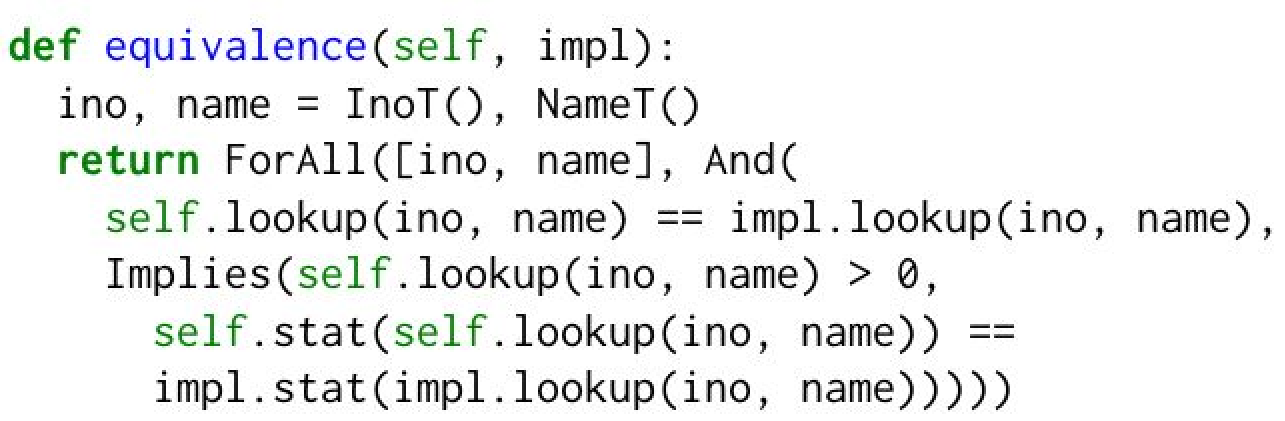


```
def mknod(self, parent, name, mtime, mode):
  # Name must not exist in parent.
  if self._childmap[(parent, name)] > 0:
    return -errno.EEXIST

  # The new ino must be valid & not already exist.
  ino = InoT()
  assertion(ino > 0)
  assertion(Not(self._parentmap[ino] > 0))

  with self.transaction():
    # Update the directory structure.
    self._childmap[(parent, name)] = ino
    self._parentmap[ino] = parent
    # Initialize inode metadata.
    self._mtimemap[ino] = mtime
    self._modemap[ino]  = mode
    self._sizemap[ino]  = 0

  return ino
```

- Transaction construct ensures all-or-nothing.

YminLFS: Specification

- Abstract data structure
- Operations
- **Equivalence predicate**



```
def equivalence(self, impl):
  ino, name = InoT(), NameT()
  return ForAll([ino, name], And(
    self.lookup(ino, name) == impl.lookup(ino, name),
    Implies(self.lookup(ino, name) > 0,
      self.stat(self.lookup(ino, name)) ==
      impl.stat(impl.lookup(ino, name)))))
```

- Represents equivalence between the state of specification and implementation.

YminLFS: Specification

- Yggdrasil specification is succinct and expressive
  - Functional correctness
  - Crash safety using transaction
- Specification is agnostic to the implementation. For the same specification, we can write log-structured and journaling filesystems.

Implementation

- Choose disk model e.g. asynchronous and synchronous
- Write each specified operation
- Consistency invariants
- YminLFS implementation is just 200 lines of python

Implementation: Disk model

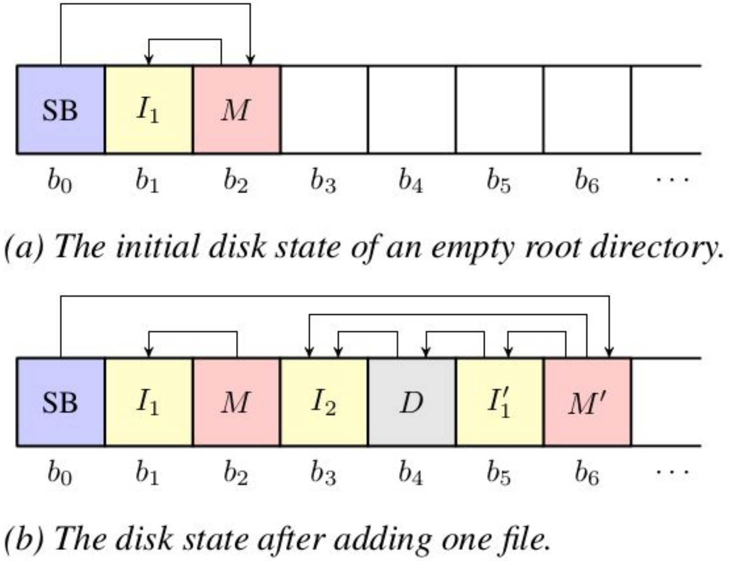
- Asynchronous model
  - Unbounded volatile cache
  - Allows arbitrary reorderings
  - Interface:
    - d.write(a, v)
    - d.read(a)
    - d.flush()
  - Block addresses are 64bits long.
  - Size of each block is 4KB
  - Single block read/write is atomic

Implementation: Disk layout

- Log-structured file system
  - Copy-on-write fashion
    - On writes: modification is done on copies blocks and old blocks are forgotten
  - No segments
  - No subdirectories
  - No garbage collection (fails when it runs out of blocks, inodes or directory entries)
  - Zero sized files (no read, write or unlink)
  - It still has to deal with crashes, reordering of writes etc

Implementation: operation mknod

1. add an inode block **I_2** for the new file
2. add a data block **D** for the root directory, which now has one entry that maps the name of the new file to its inode number 2
3. add an inode block **I'_1** for the updated root directory, which points to its data block **D**
4. add an inode mapping block **M'**, which has two entries: **1→b5** and **2→b3**
5. finally, update the superblock **SB** to point to the latest inode mapping **M'**.



(a) The initial disk state of an empty root directory.

(b) The disk state after adding one file.

SB: superblock
M: inode to block mapping

Disk flush after each write.

Implementation: consistency invariants

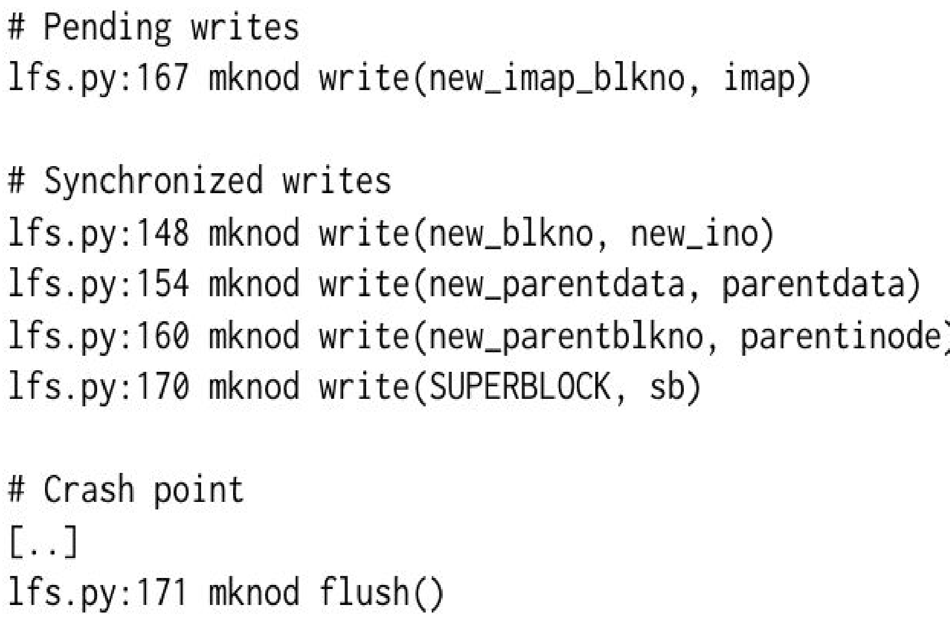
- Analogous to the well formedness invariant for the specification
- It determines whether a dist state is a valid log-structured file system image
- Implementation invariants are used for
  - Verification (do we really need for verification ??)
  - fsck util generation
- Invariants are checked for the initial file system and used in forming the precondition and postcondition.
- Invariants:
  - SB constraints
    - Next available inode number i > 1
    - Next available block number b > 2
    - Pointer to M belongs to (0, b) (shouldn't it be (1, b) ??)
  - Inode mapping constraints (M)
    - For each entry (I, B) : I belongs to (0, i) and B belongs to (0, b)
  - Root dir constraints (D)
    - For each entry (name, I) : I belongs to (0, i)

Verification

- Crash free executions: same behaviour of specification and implementation
  - Given consistent and equivalent states, specification and implementation produces equivalent and consistent states in the absence of crashes
- Crashing executions:
  - Each possible crash state (including the ones due to reordering) in the implementation must be equivalent to **some** state in the specification and the states should be consistent
- Equivalence is determined using the **equivalent** predicate given in the specification

Counterexample

1. add an inode block **I_2** for the new file
2. add a data block **D** for the root directory, which now has one entry that maps the name of the new file to its inode number 2
3. add an inode block **I'_1** for the updated root directory, which points to its data block **D**
4. add an inode mapping block **M’**, which has two entries: **1**→**b5** and **2**→**b3**
5. finally, update the superblock **SB** to point to the latest inode mapping **M’**.



```
      # Pending writes
Step4 lfs.py:167 mknod write(new_imap_blkno, imap)

      # Synchronized writes
Step1 lfs.py:148 mknod write(new_blkno, new_ino)
Step2 lfs.py:154 mknod write(new_parentdata, parentdata)
Step3 lfs.py:160 mknod write(new_parentblkno, parentinode
Step5 lfs.py:170 mknod write(SUPERBLOCK, sb)

      # Crash point
      [..]
      lfs.py:171 mknod flush()
```

Flush is missing between step 4 and 5.

Counterexample/proof

- Initial implementation contained two bugs in lookup logic and data layout.
  - Could not be detected in testing runs
  - Verifier found the same in seconds
- Proof:
  - If there is no counterexample found, then **none** exists, and the implementation is correct
  - Note that correctness hold for disks with up to 2^64 blocks and inodes
  - For all possible traces, crash scenarios and reorderings
  - The theorem only holds when disk is modified only through the file system

Optimizations and compilation

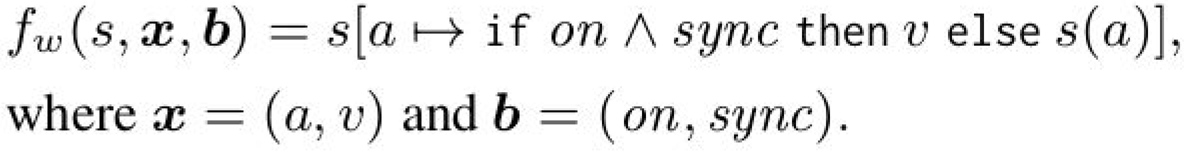
- Optimization
    - Minimize disk flushes
        - In mknod: first three disk flushes can be removed in 3 mins
- Yggdrasil compilation
    - Implementation -> executable
    - Implementation -> C code -> executable [using CPython]
    - The result is a single-threaded user-space file system
- Summary
    - No manual proofs
    - No annotations
    - Counterexample visualizer is useful for pointing bugs
    - Trusted computing base:
        - Yggdrasil (Verifier, visualizer and compiler). Optimizer is not trusted.
        - Dependencies like Z3, Python, gcc, FUSE, Linux kernel

Crash refinement

- Crash refinement intuition
    - F0 specification and F1 is the implementation
    - F1 is correct wrt F0 if starting from equivalent consistent states and invoking same operations on both systems any state produced by F1 is equivalent to some state in F0
    - We do this for all operations and for the whole system

Modeling crashes and flushes

- Each operation is modeled with a function with three inputs
        - Current state
        - External input
        - Crash schedule
    - Example: write operation (a -> v) fw
        - Current state s (s(a) represent data at address a)
        - External input = (a, v)
        - Crash schedule: for asynchronous disk model for the write operation is pair of boolean values (on, sync)
            - On: write operation completed and value is stored to volatile cache
            - Sync: write value is synchronized to persistent memory



$$f_w(s, \boldsymbol{x}, \boldsymbol{b}) = s[a \mapsto \texttt{if } on \wedge sync \texttt{ then } v \texttt{ else } s(a)],$$

where $\boldsymbol{x} = (a, v)$ and $\boldsymbol{b} = (on, sync)$.

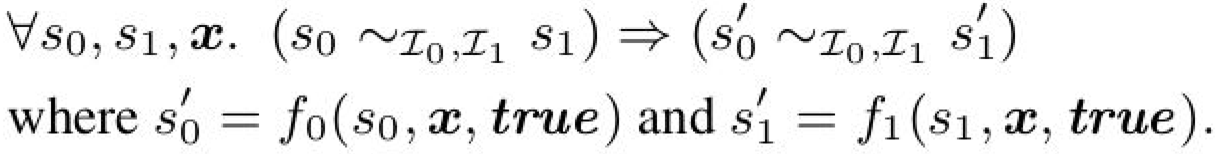
Crash refinement:

Definitions: State equivalence

$$s_0 \sim s_1$$

$$s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1 \triangleq \mathcal{I}_0(s_0) \wedge \mathcal{I}_1(s_1) \wedge s_0 \sim s_1 .$$

Defn: Crash-free equivalence

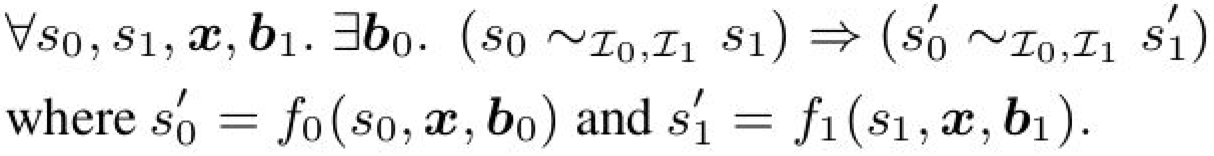


$$\forall s_0, s_1, \boldsymbol{x}.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$

where $s_0' = f_0(s_0, \boldsymbol{x}, \boldsymbol{true})$ and $s_1' = f_1(s_1, \boldsymbol{x}, \boldsymbol{true})$.

Defn: Crash refinement w/o recovery (crashes but no recovery)

- If the functions are crash-free equivalent and following holds:



$$\forall s_0, s_1, \boldsymbol{x}, \boldsymbol{b}_1.\ \exists \boldsymbol{b}_0.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$

where $s_0' = f_0(s_0, \boldsymbol{x}, \boldsymbol{b}_0)$ and $s_1' = f_1(s_1, \boldsymbol{x}, \boldsymbol{b}_1)$.

Defn: Recovery function idempotence

- Recovery function is idempotent if

$$\forall s, \boldsymbol{b}.\ r(s, \boldsymbol{true}) = r(r(s, \boldsymbol{b}), \boldsymbol{true}).$$

Defn: Crash refinement with recovery

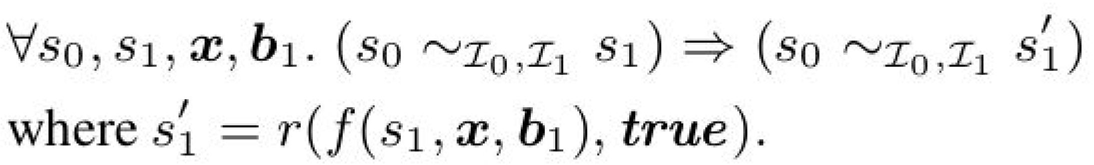
- If the functions are crash-free equivalent and following holds:

$$\forall s_0, s_1, \boldsymbol{x}, \boldsymbol{b}_1.\ \exists \boldsymbol{b}_0.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$

$$\text{where } s_0' = f_0(s_0, \boldsymbol{x}, \boldsymbol{b}_0) \text{ and } s_1' = r(f_1(s_1, \boldsymbol{x}, \boldsymbol{b}_1), \boldsymbol{true}).$$

Defn: No-op

- Function f with recovery function r is a no-op if
- r is idempotent and following holds:



$$\forall s_0, s_1, \boldsymbol{x}, \boldsymbol{b}_1.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$

$$\text{where } s_1' = r(f(s_1, \boldsymbol{x}, \boldsymbol{b}_1), \boldsymbol{true}).$$

- Background operations which do not change the externally visible state of the system are no-ops.

System crash refinement

- Given two systems F0 and F1 and recovery function r
- F1 is a crash refinement of F0 if every function in F1 with r is either a crash refinement of the corresponding function in F0 or a no-op.

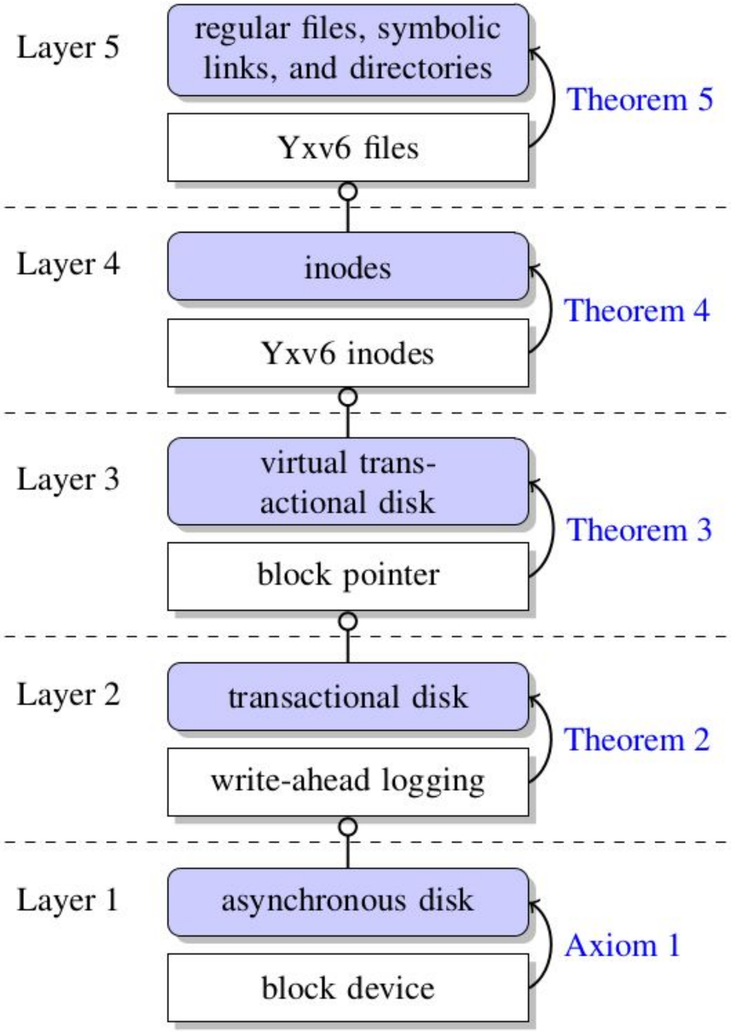
Yxv6 file system overview

- Journaling based file system similar to xv6
  - Write-ahead logging
- Module based
  - Reduces SMT encoding size
  - Faster SMT queries
  - Multiple disks for different logical parts of the disk e.g. log, free bitmap etc.
- Yxv6+sync and Yxv6+group-commit
  - Group-commit combines multiple transactions in to one.

Yxv6 file system layers

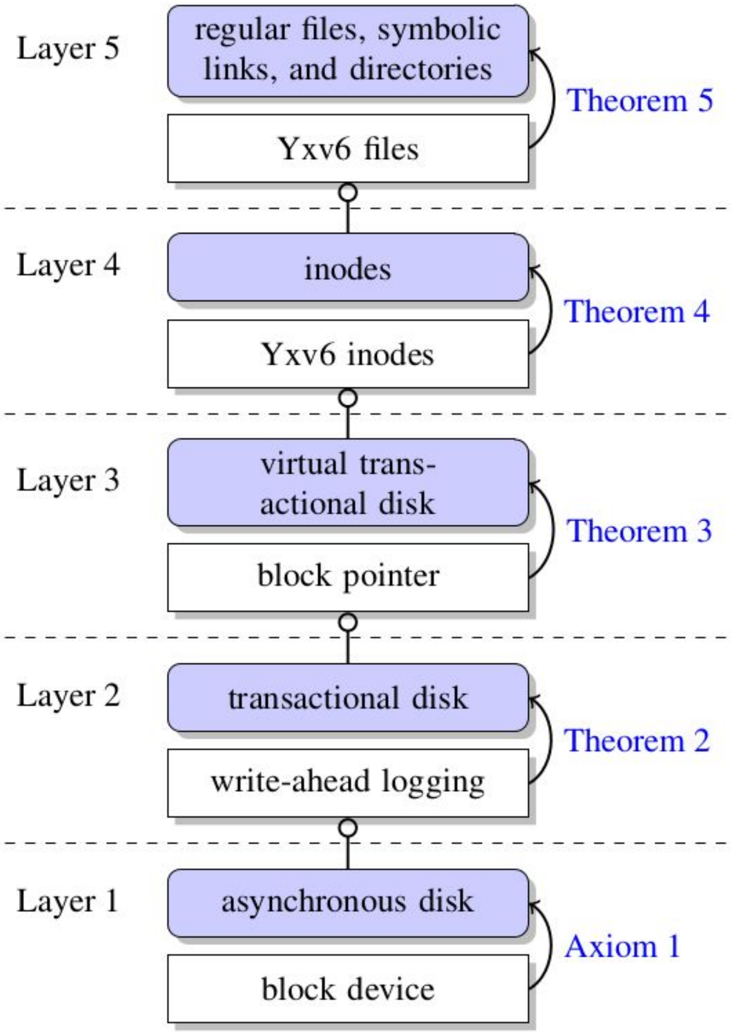
- A layer is proven in each step.
- Once a layer is proven, the top layer use the specification of bottom layers.

- Layer 1: Asynchronous disk
  - Axiom 1: block device is a crash refinement of asynchronous disk specification.



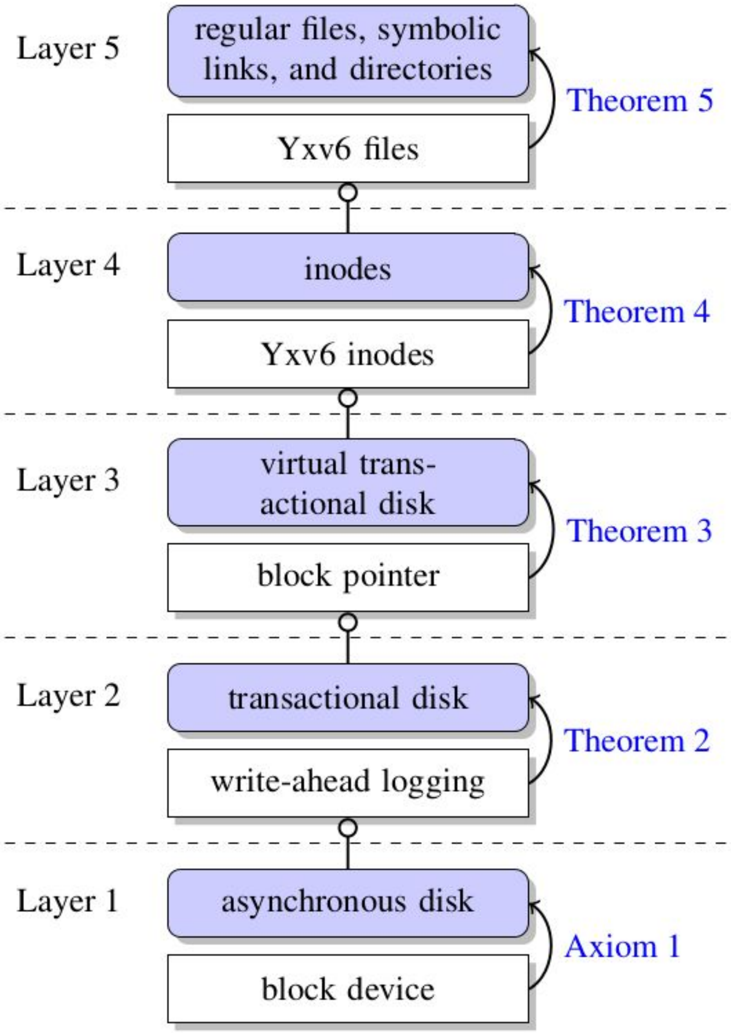
Yxv6 file system: Layer 2: Transactional disk

- **Specification:** Transactional disk manages **multiple disks** and provides abstractions:
  - d.begin_tx()
  - d.commit_tx()
  - d.write_tx()
  - d.read()
  - Operations in a transaction are atomic and sequential.
- **Implementation:**
  - Write-ahead logging
  - One log for all disks



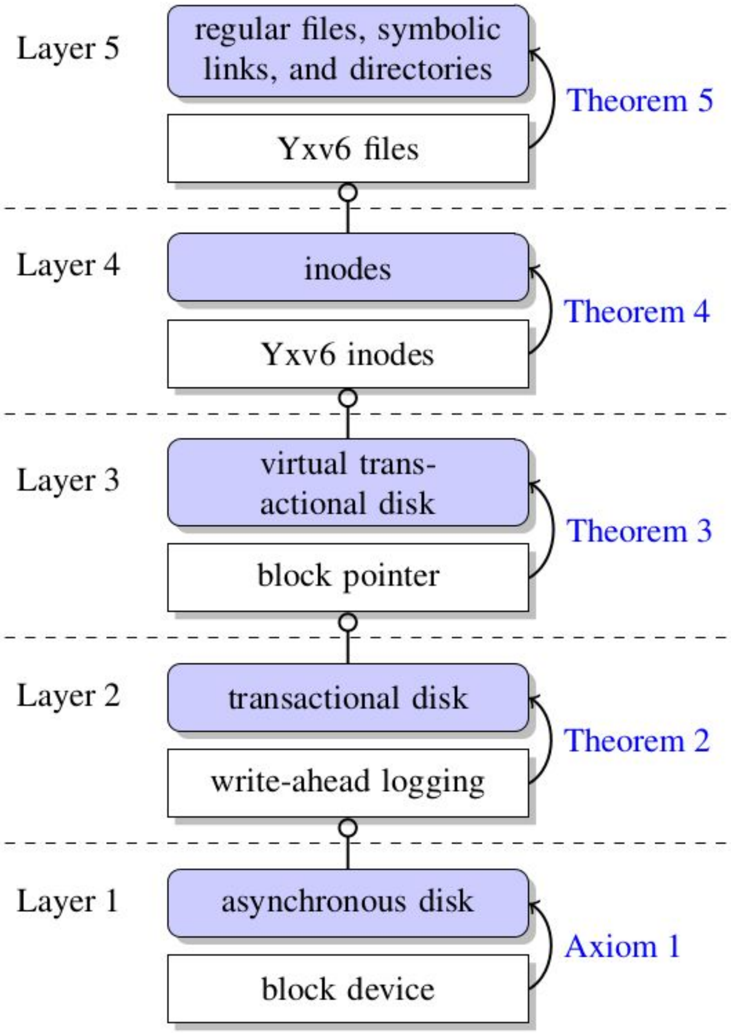
Yxv6 file system: Layer 2: Transactional disk

- **Specification:** Transactional disk manages **multiple disks** and provides abstractions:
  - d.begin_tx()
  - d.commit_tx()
  - d.write_tx()
  - d.read()
  - Operations in a transaction are atomic and sequential.
- **Implementation:**
  - Write-ahead logging
  - One log for all the managed disks



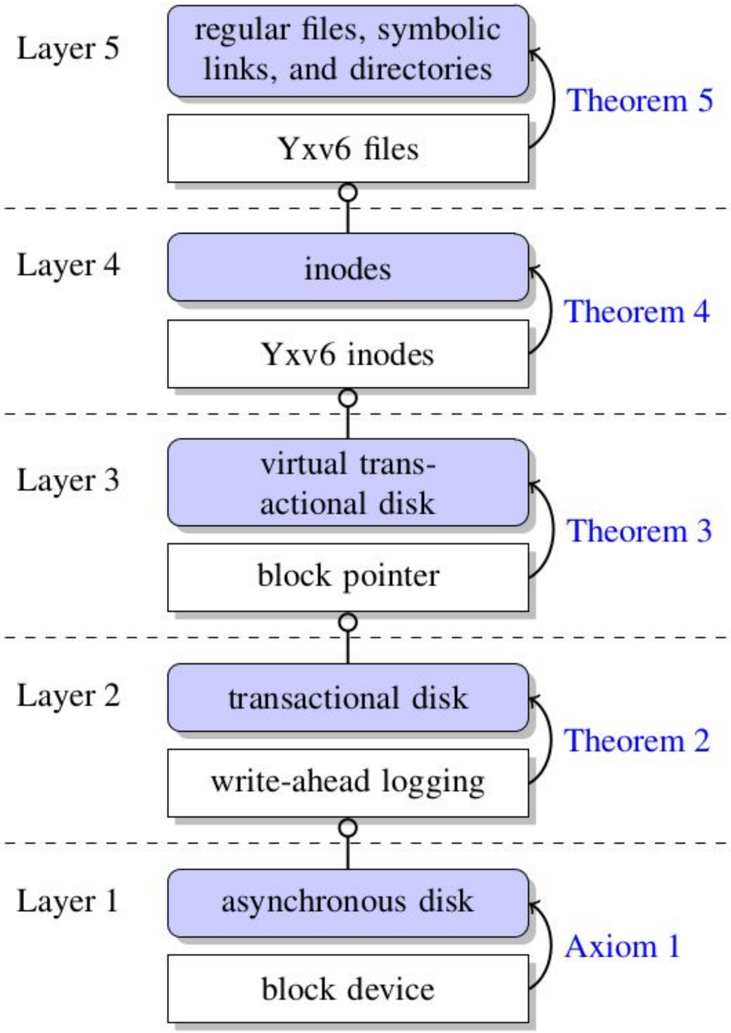
Yxv6 file system: Layer 3: Virtual transactional disk

- **Specification:**
  - 64-bit virtual disk addresses
  - Only the mapped addresses can be read/written
  - Simplifies inode implementation
- **Implementation:**
  - Uses one transactional disk with three data disks
    - Free block bitmap
    - Direct block pointers
    - Data + singly indirect block pointers
  - Free block bitmap: One bit in each block for SMT encoding simplification
- **Invariants:**
  - Injective mapping (one-to-one)
  - If block with address a is mapped then ath bit in block bitmap must be marked



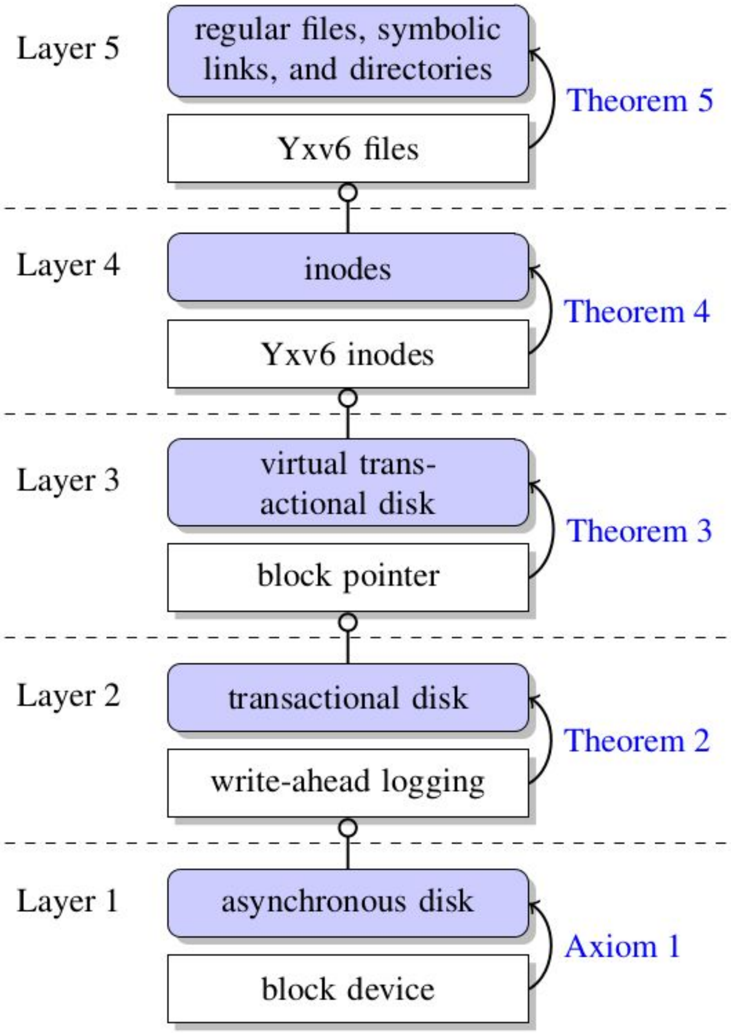
Yxv6 file system: Layer 4: Inodes

- **Specification:**
  - 32-bit long inode number
  - Each inode is mapped to 2^{32} blocks
  - Each inode is mapped to metadata like size, mtime and mode
- **Implementation:**
  - 64-bit virtual disk address space is split in 2^{32} ranges each with 2^{32} virtual blocks.
  - Uses separate disk for metadata.
- **Invariants:**
  - None



Yxv6 file system: Layer 5: File System

- **Specification:**
  - Extension of FSSpec with regular files, directories and symbolic links.
- **Implementation:**
  - Builds on top of inode specification
  - Inode bitmap disk
  - Orphan inode disk
- **Invariants:**
  - Size of unused inode must be zero
  - Inode using n blocks should have virtual blocks larger than n unmapped.

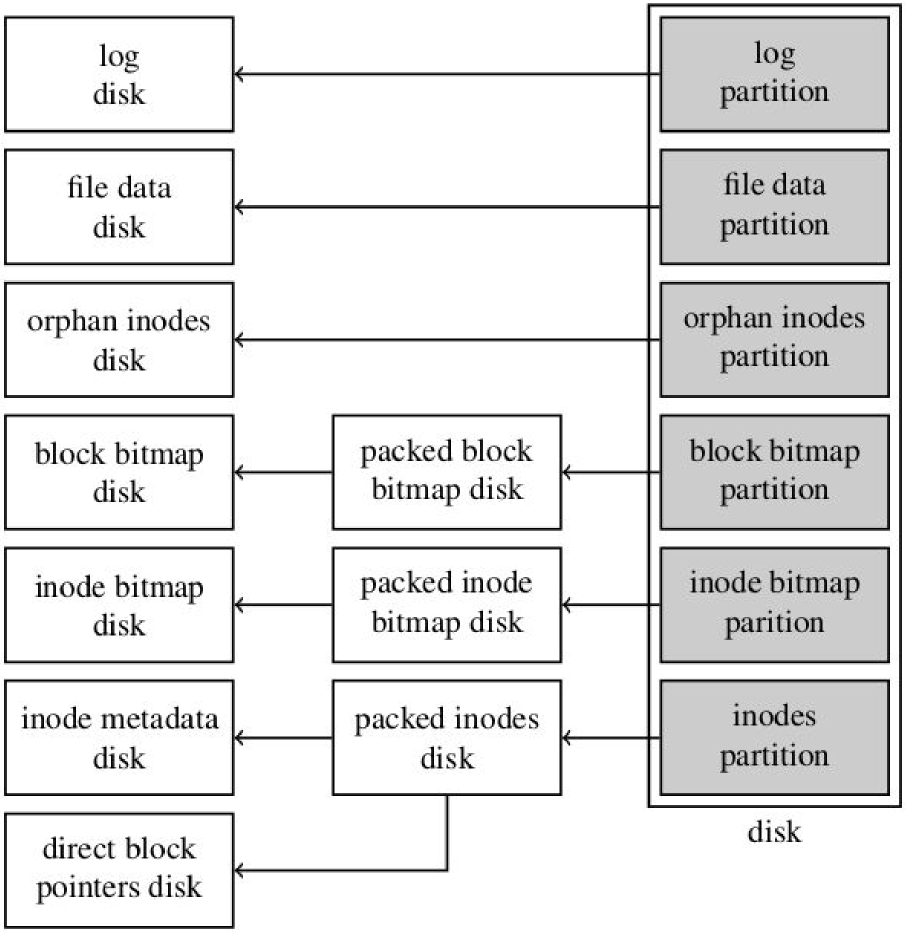


Finitization

- Most of the operations are finite (bounded loops)
- With two exceptions:
  - Search-related procedure like finding free bit in bitmap
    - Validation is used for these cases.
      - E.g. runtime check whether index returned is free in the bitmap
  - Unlink
    - To finitize: implementation moves the inode to orphan inodes disk. Garbage collector later reclaim the data blocks. And garbage collection is proven a no-op. Does not change the externally visible state.

Single disk and packed bitmaps

- Packed bitmap is a refinement of block bitmap
- Using single single disk is a refinement of using of seven disks (non-overlapping).



Yxv6+group-commit and Yxv6+sync

- Yxv6+group-commit is a crash refinement of Yxv6+sync

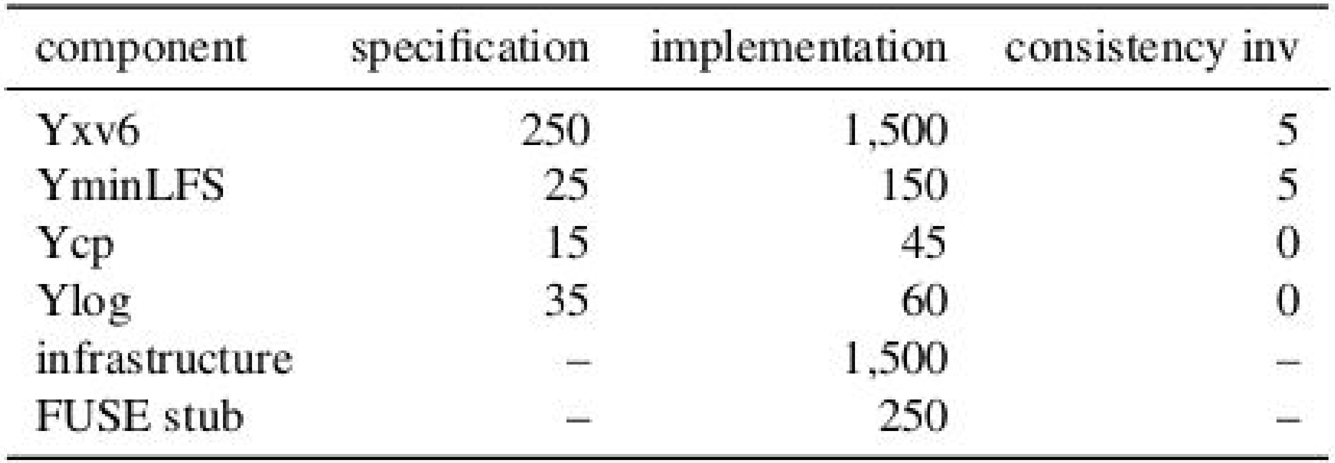
Beyond file systems

- Yggdrasil can be used for writing applications which use file systems e.g. Ycp
- Ycp spec:
    - If copy succeeds the target file is a copy of source file
    - If fails due to crash (or invalid target) file system is unchanged
- Ycp implementation:
    - Steps:
        - Create a tmp file
        - Write the source data to it
        - Rename
- Ycp implementation is proven to be a crash refinement of the specification

Yggdrasil limitations

- Single-threaded, does not support concurrency
- Cython is not verified
- SMT is limited to first order logic not as powerful is Coq and Isabelle. However, it is sufficient for Yxv6.
- Yxv6 does not support modern features like extents and delayed allocation (allocate-on-flush)
- Generated Fsck cannot repair

Implementation



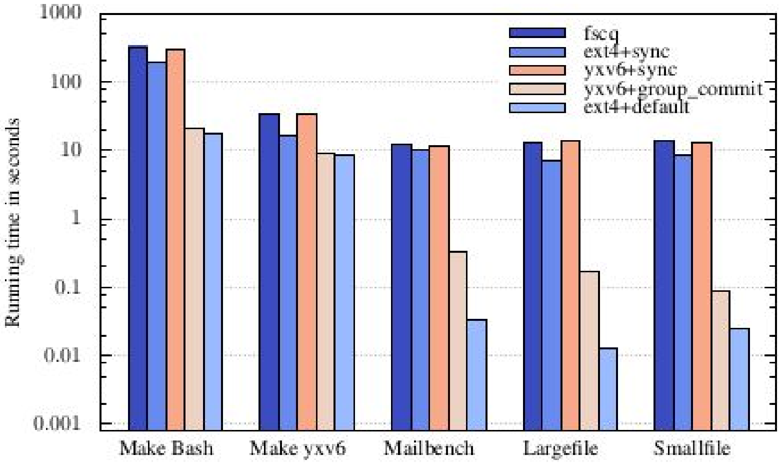
component	specification	implementation	consistency inv
Yxv6	250	1,500	5
YminLFS	25	150	5
Ycp	15	45	0
Ylog	35	60	0
infrastructure	–	1,500	–
FUSE stub	–	250	–

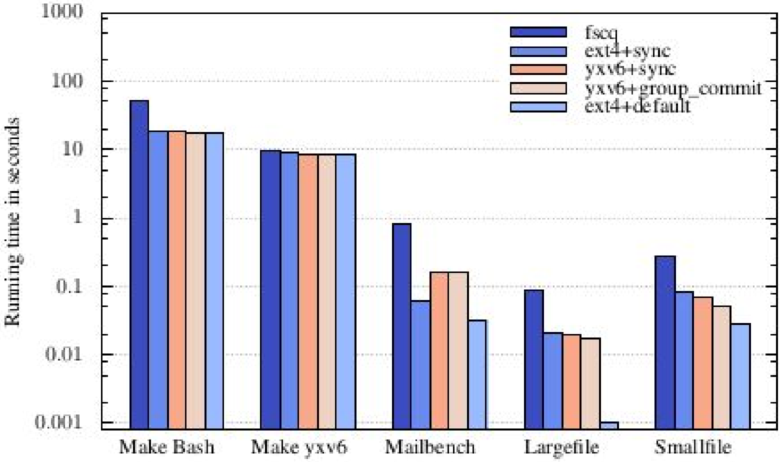
Evaluation: correctness

- fsstress tests from the Linux Test Project
- SibylFS POSIX conformance tests
- Yggdrasil development + writing of paper hosted on Yxv6
- Block Order Breaker to cross-check that the file system state was consistent after a crash and recovery.
- Manually corrupted the file system and ran fsck

Evaluation: Run-time performance

- SSD
  - Yxv6+sync performs similar to ext+sync and fscq
  - Group_commit is 3–150× faster than ext+sync and fscq
  - Group_commit is within 10× ext+default
- RAM disk to understand CPU overheads
  - Fscq is slow because of haskell extracted code
  - Yxv6 benefits from C code
  - Largefile is exception





Evaluation: Verification performance

- One hour to verify Yxv6+sync
- 1.6 hours to verify Yxv6+group-commit (on 24 cores) and 36 hours on single core
- Related: FSCQ takes 11 hours

Related work

- FSCQ: Crash Hoare logic
- Flashix: similar approach, interactive verification
- Bug-finding tools