Designing Systems for Push-Button Verification

Luke Nelson, Helgi Sigurbjarnarson, Xi Wang

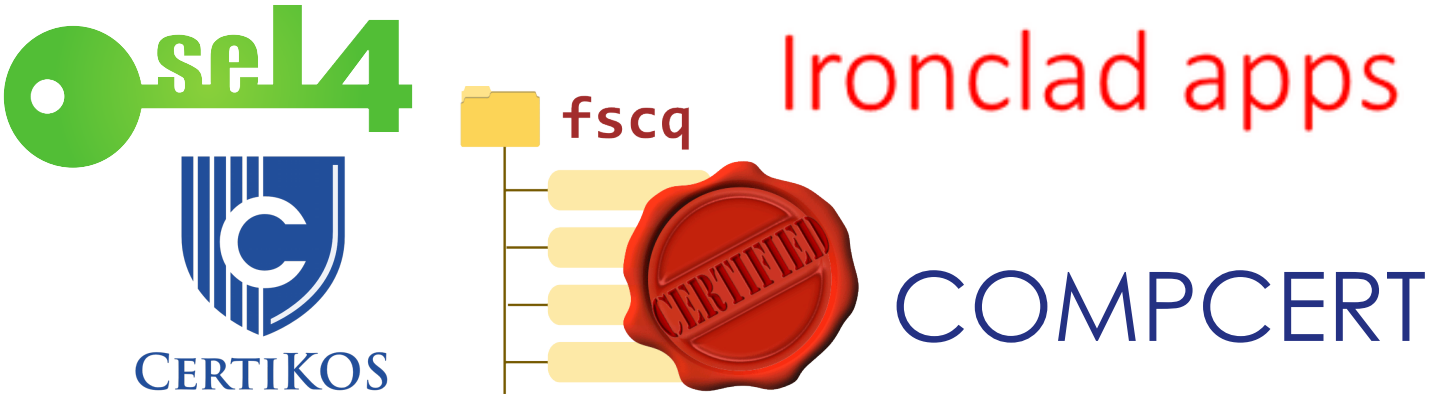
Joint work with James Bornholt, Dylan Johnson, Arvind Krishnamurthy, Emina Torlak, Kaiyuan Zhang

UNIVERSITY *of* WASHINGTON

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

Formal verification of systems

- Eliminate entire classes of bugs
- Write a spec & prove impl meets the spec



- Verification projects at UW: Bagpipe [OOPSLA'16], Neutrons [CAV'16], Verdi [PLDI'15], …

Challenge 1/3: non-trivial efforts

- Time-consuming: often person-years
- Require high-level of expertise
- Example: the seL4 kernel
  - 10 KLOC code,
  - 480 KLOC proof
  - 11 person-years

Challenge 2/3: spec

- What *is* a correct system
  - Low-level correctness is well-understood: no overflow
  - Some fields have been using formal specs: TLA+
  - Difficult in general

- Examples
  - The file system must ensure *crash safety*
  - The OS kernel must enforce *process isolation*

Challenge 3/3: integration w/ dev

- Learning curve
- Improve upon testing (e.g., Driver Verifier)
- Moving target
- Incremental deployment

Push-button verification

- System design for minimizing proof efforts
- Verifiability as a first-class concern
- Leverage advances in automated SMT solving
  - But need to use solvers wisely
  - Limitations on expressiveness

From static analysis to verification

"There has been a seismic shift in terms of the average programmer 'getting it.' When you say you have a static bug-finding tool, the response is no longer 'Huh?' or 'Lint? Yuck.' This shift seems due to static bug finders being in wider use, giving rise to nice networking effects."

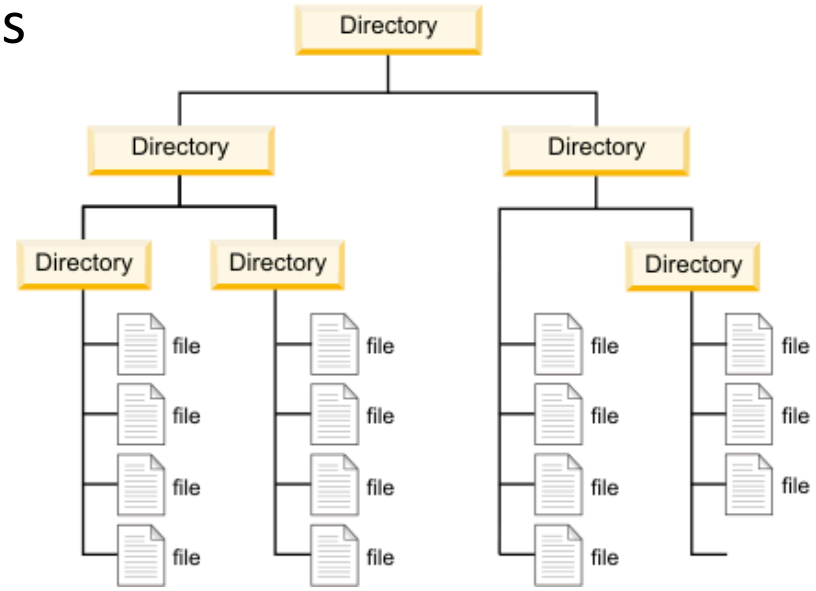
A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World — Coverity, CACM 2010

Outlines

- Yggdrasil: writing verified FSes [OSDI'16]
- Hyperkernel: a verified OS kernel [SOSP'17]
- Lessons learned & future work

Yggdrasil [OSDI'16]

- File systems are essential for data integrity
- But are difficult to get right
  - Complex on-disk data structures
  - Must ensure crash safety
- Bugs are hard to reproduce



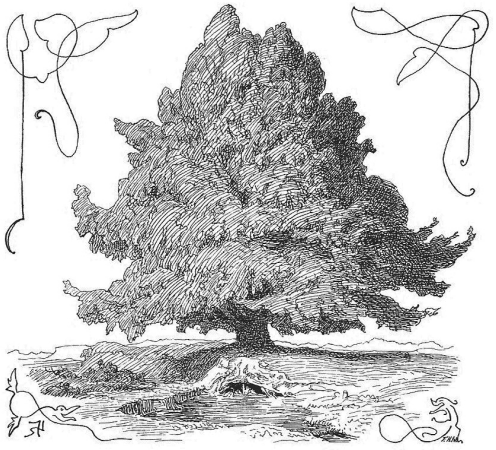
FS challenges

- Too many states
  - Disks are large; many execution paths
  - Non-determinism: crash, reordering writes
- Techniques
  - Testing: eXplode [OSDI ’06], EXE [CCS ’06]
  - Interactive proving: FSCQ [SOSP’15], Cogent [ASPLOS’16]
- How to automate FS verification

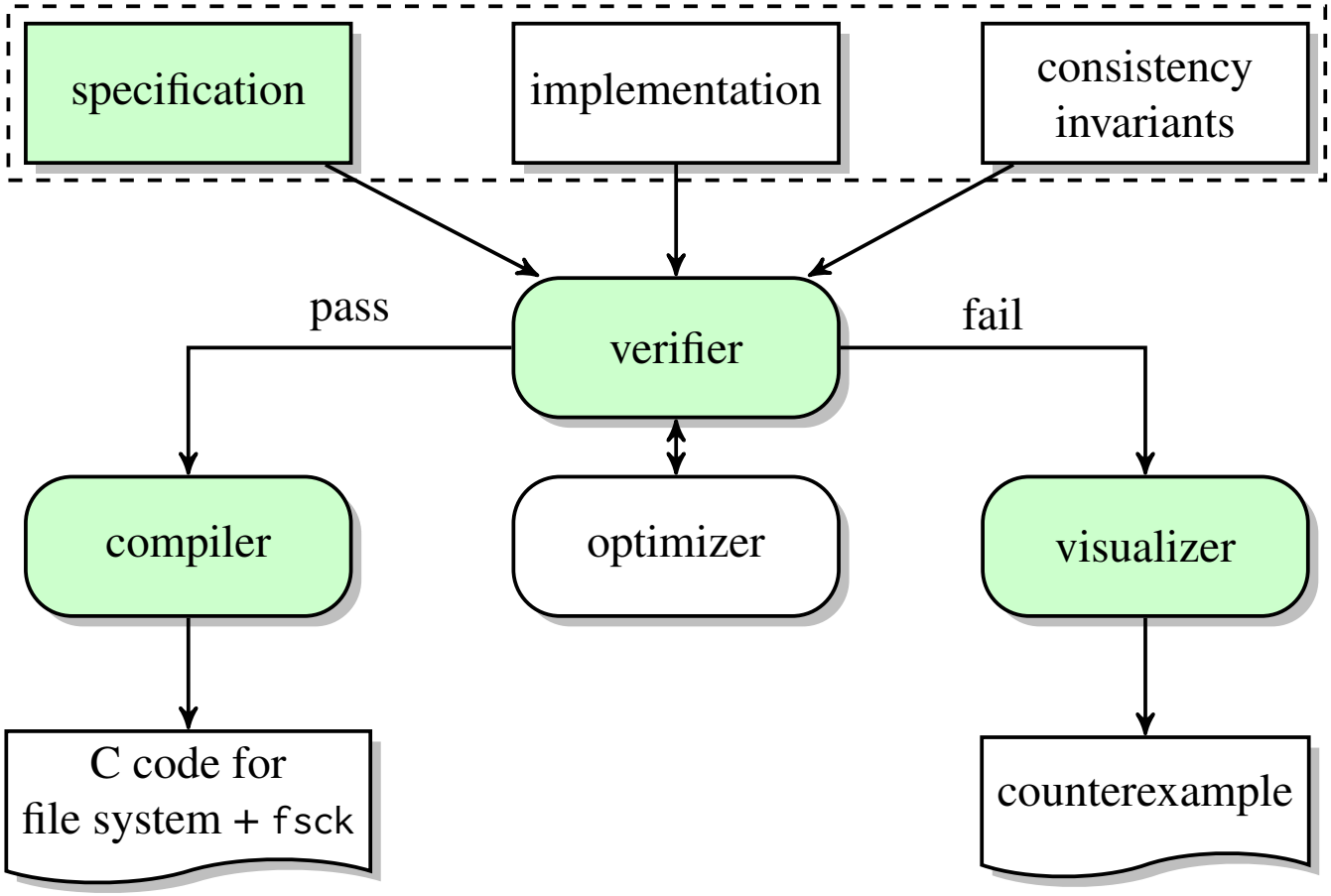
Yggdrasil: writing verified FSes

- Key ideas
  - A definition of FS correctness amenable to SMT solving
  - Layering to scale verification
  - Separating layout from correctness

- Main result: Yxv6 file system
  - Similar to ext3 and xv6
  - Verified functional correctness
  - Verified crash safety



Yggdrasil overview

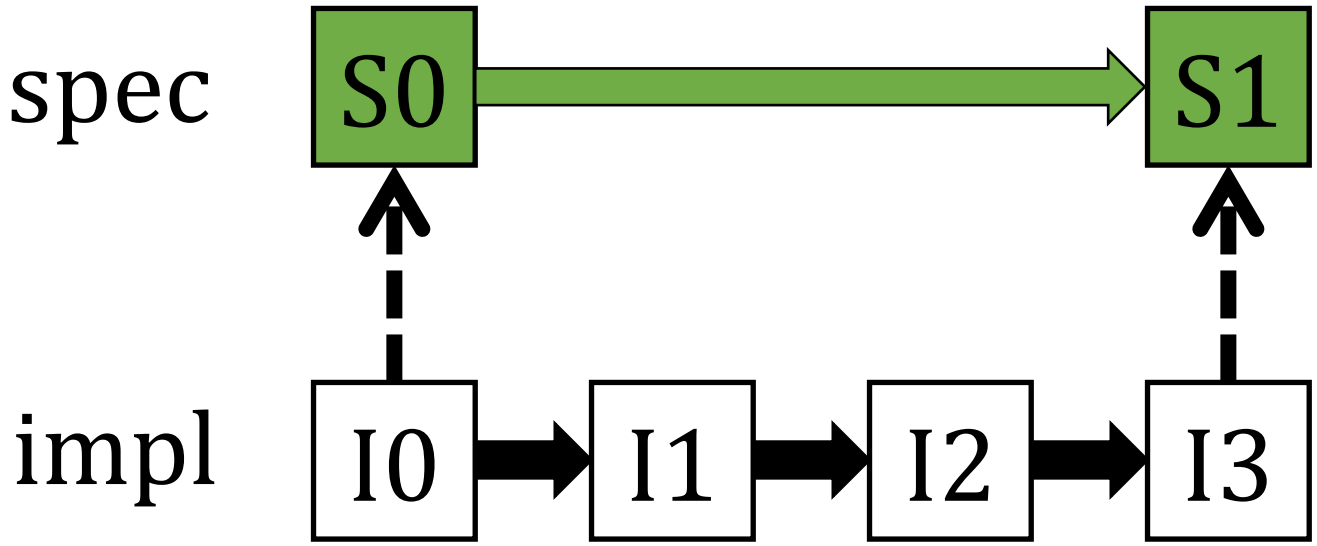


Example spec

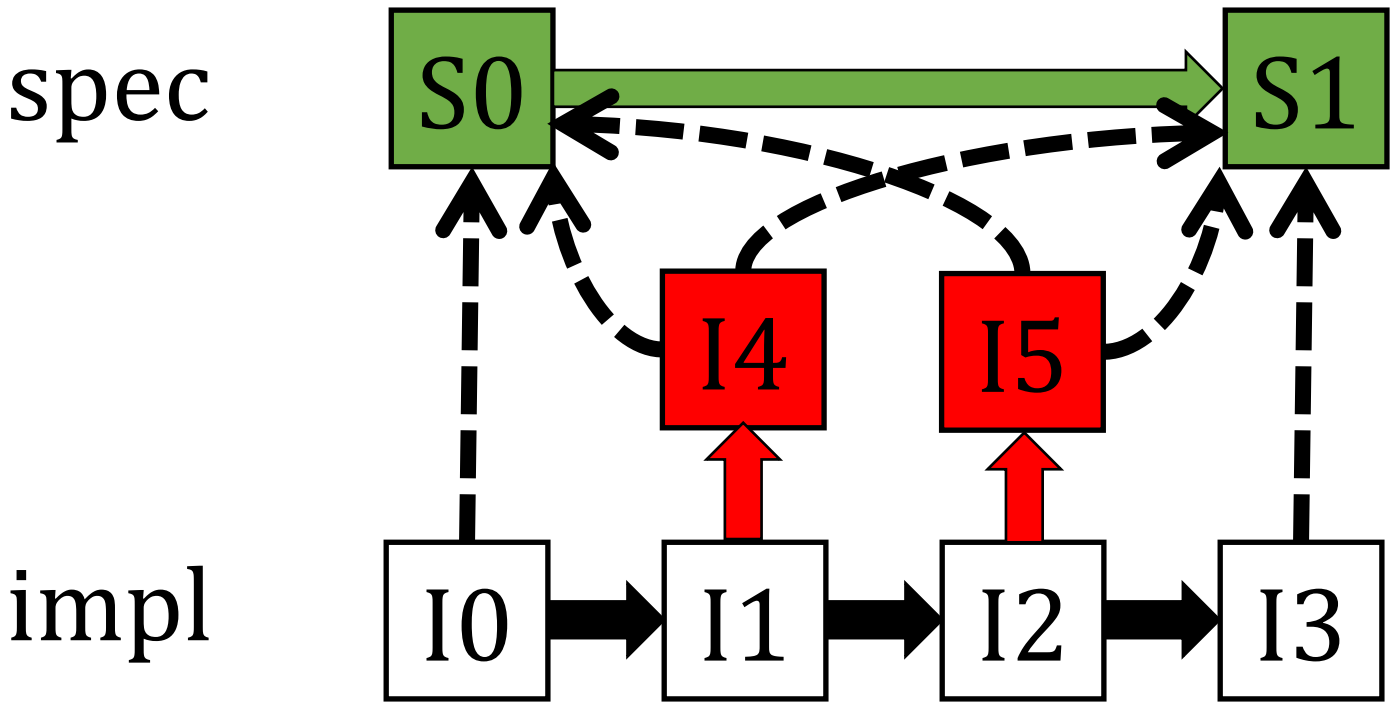
```
class TxnDisk(BaseSpec):
  def begin_tx(self):
    self._txn = []

  def write_tx(self, bid, data):
    self._cache = self._cache.update(bid, data)
    self._txn.append((bid, data))

  def commit_tx(self):
    with self._mach.transaction():
      for bid, data in self._txn:
        self._disk = self._disk.update(bid, data)
```

Strawman: doesn't capture crash

- Model FS as a state machine with a set of operations { create, rename, etc. }



Crash refinement



Crash refinement definition

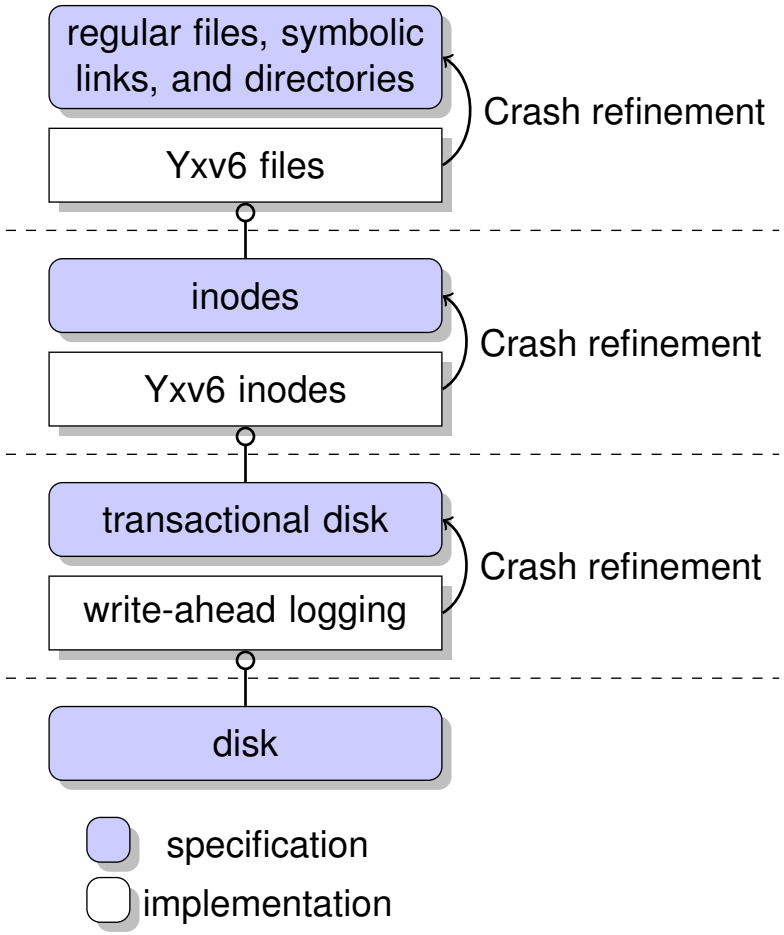
- Model FS as a state machine
- Augment each op with an explicit crash schedule: op(disk, inp, sched) → disk
- For each FS op, prove:

$$\forall\, \text{disk}, \text{inp}, \text{sched}_{\text{impl}}.\ \exists\, \text{sched}_{\text{spec}}.$$
$$\text{op}_{\text{spec}}(\text{disk}, \text{inp}, \text{sched}_{\text{spec}}) =$$
$$\text{op}_{\text{impl}}(\text{disk}, \text{inp}, \text{sched}_{\text{impl}})$$

- Z3 is good at solving this form

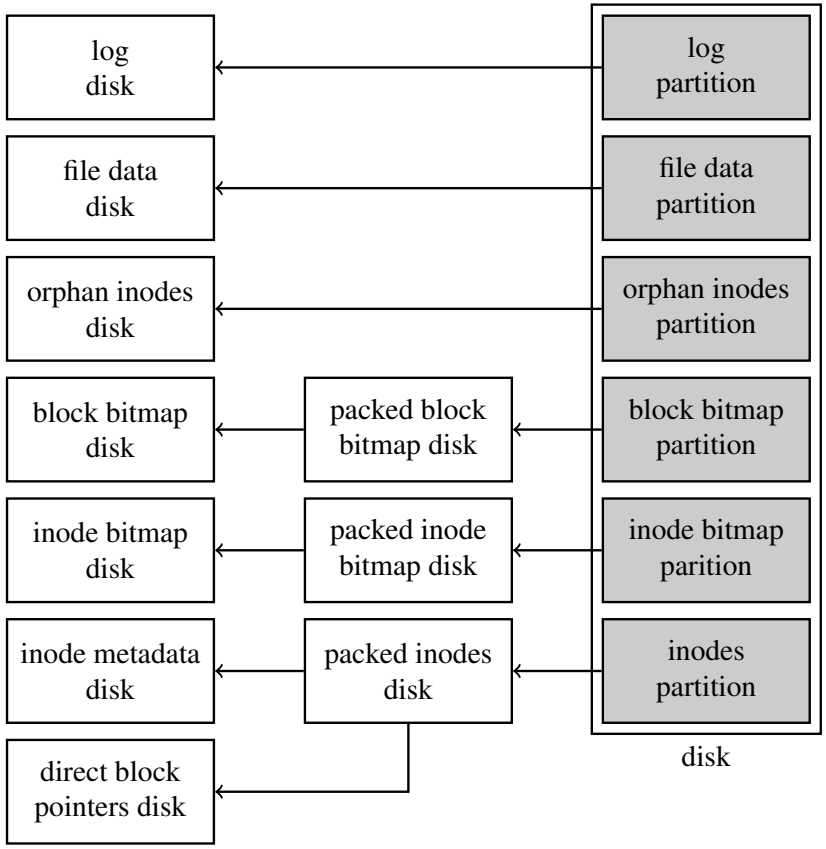
Stack of layered abstractions

- Each layer has a spec
- Each layer builds upon a lower layer spec
- Limit verification to a single layer at a time



Separate refinement of layout

- Start with multiple disks & inefficient layout
- Gradually refine to optimized layout
- Separate reasoning of correctness from layout



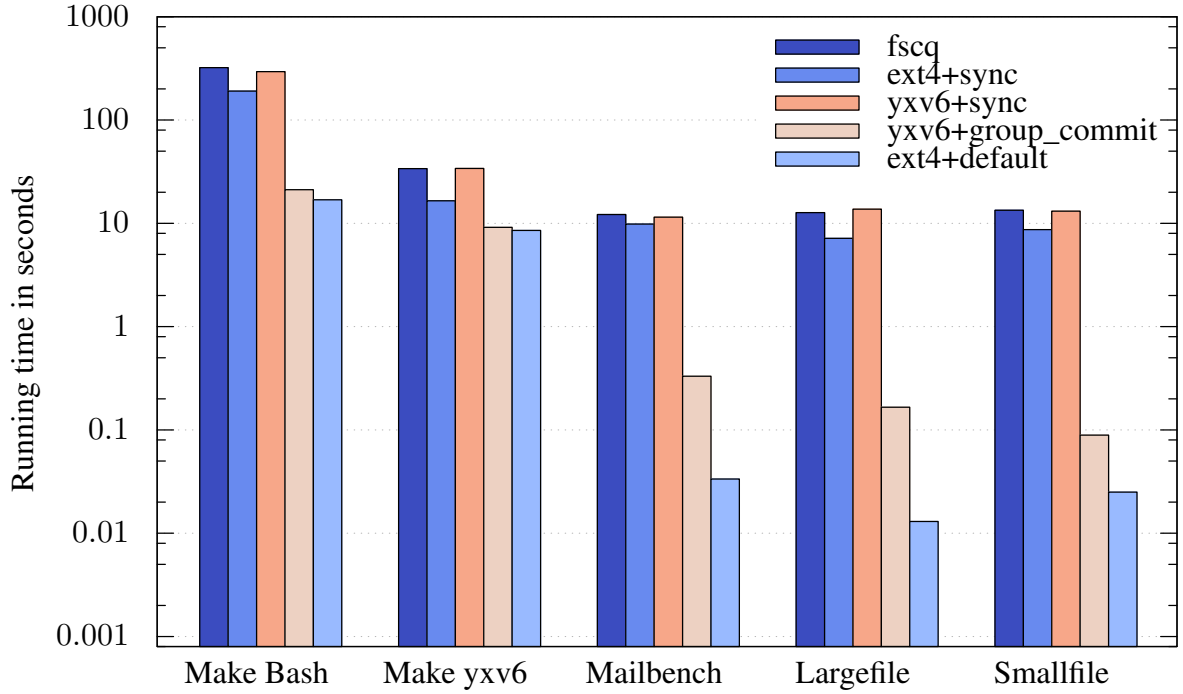
Implementation w/ Python & Z3

- Two Yxv6 variants
  - Yxv6+sync: similar to xv6, FSCQ and ext4+sync
  - Yxv6+group_commit: an optimized Yxv6+sync
- verified: 1.6 hours w/ 24 cores - no manual proofs!

	spec	impl	consistency inv.
Yxv6	250	1,500	5
infrastructure	--	1,500	--
FUSE stub	--	250	--

Run-time performance

- 3–150× faster than ext4+sync
- Within 10× of ext4+default



Summary of Yggdrasil

- Push-button verification is feasible for FS
  - No manual proofs on implementation
  - New FS correctness definition: crash refinement

- FS design for verification
  - Model FS as a state machine
  - Verify each operation using crash refinement
  - Verify each layer independently

Hyperkernel [SOSP'17]

- The OS Kernel is a critical component
  - Isolation is essential for application security
  - Kernel bugs can compromise the entire system
- Manual verification is costly
- Goal: OS design for automated SMT verification

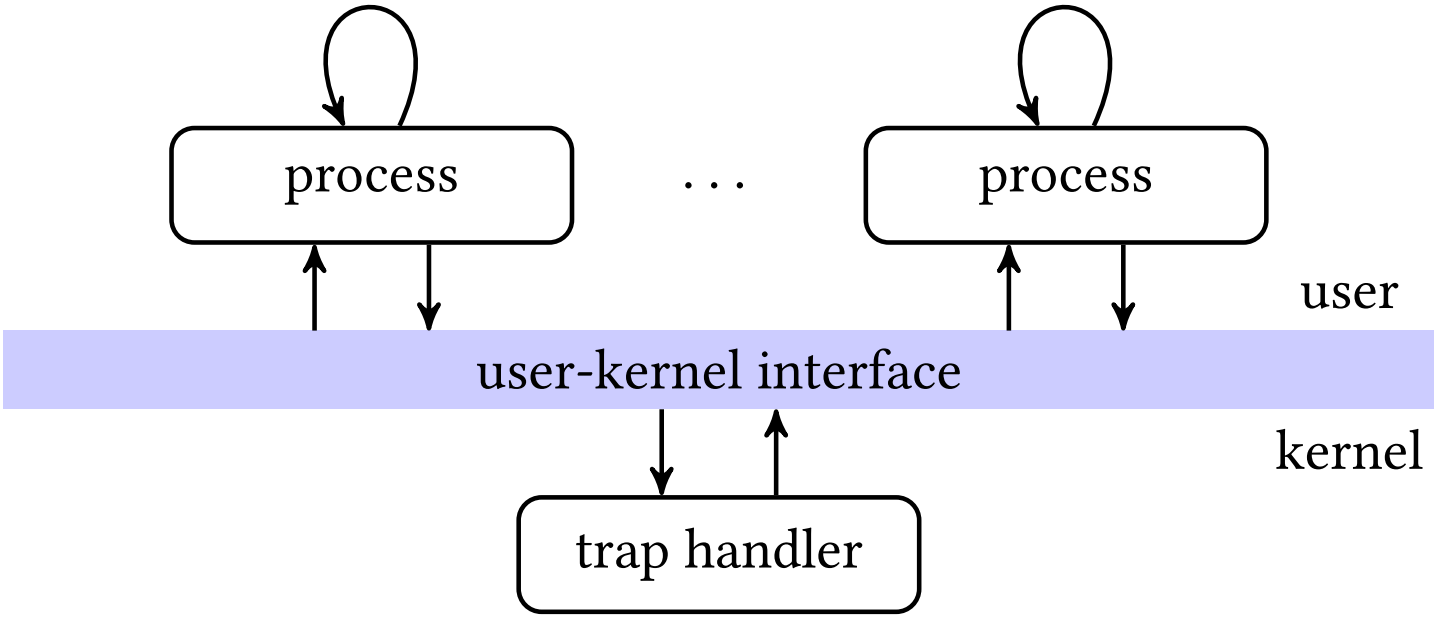
Design challenges

- Kernel API must be amenable to SMT reasoning
- Kernel pointers are difficult to reason about
  - kernel runs under virtual memory
  - kernel also manipulates the mapping
  - the mapping is often non-injective
- C is known to be difficult to model

Ideas: design to scale verification

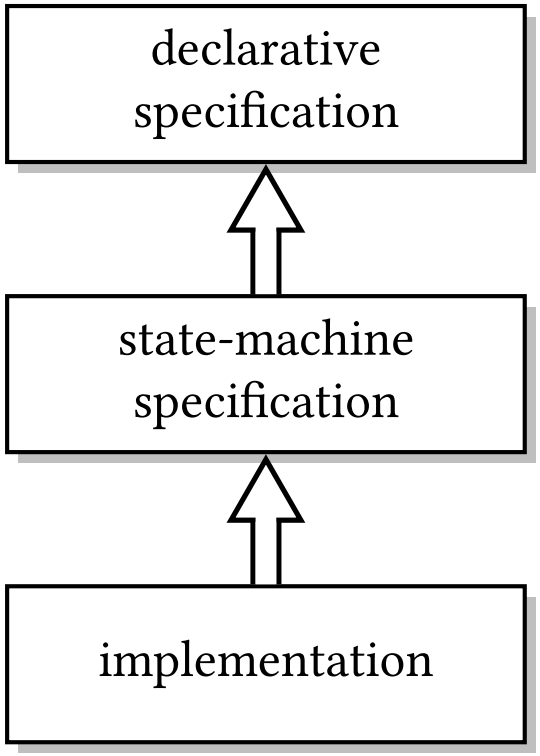
- Finite interface: no loops/interrupts in kernel
  - Use SMT-friendly data structures (e.g., bitmaps)
  - Use validation whenever possible
- Identity mapping in kernel
  - Separate address spaces in kernel and user
  - “Abuse” virtual machine instructions
- Verification using LLVM IR instead of C
- SMT encodings for reference counting, etc.

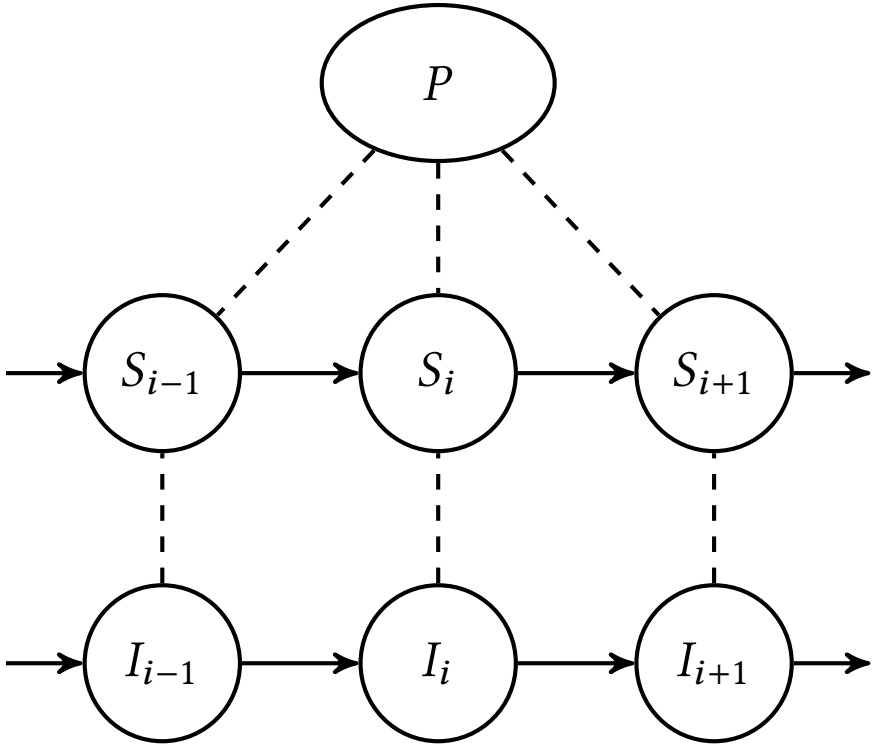
Model OS as a state machine



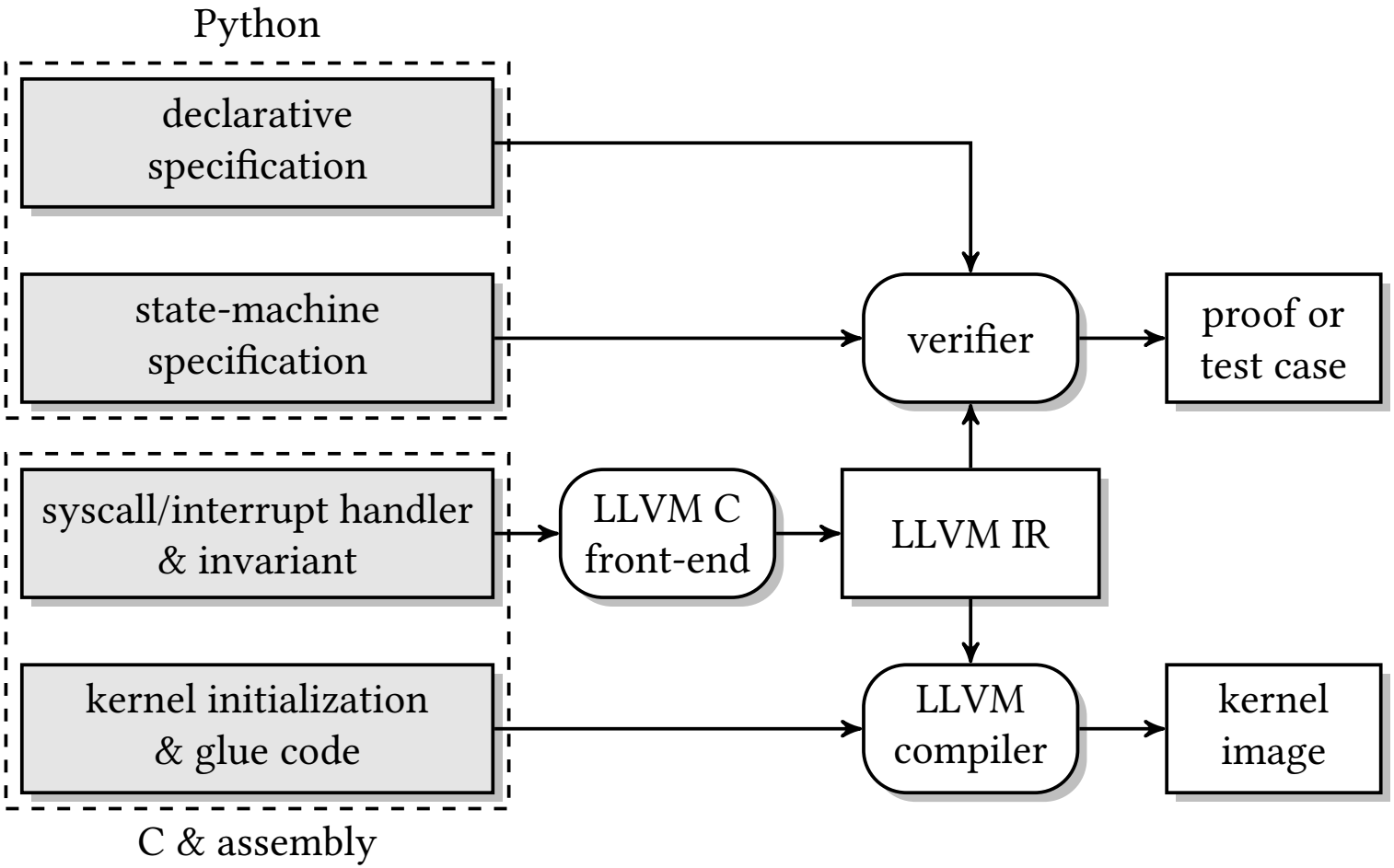
- Assume a uniprocessor system
- Assume initialization and glue code correct

Main theorems





Workflow


Python
declarative specification
state-machine specification
verifier
proof or test case
syscall/interrupt handler & invariant
LLVM C front-end
LLVM IR
kernel initialization & glue code
LLVM compiler
kernel image
C & assembly

Demo

- Workflow
- Virtual memory management

Summary of Hyperkernel

- Feasible to verify a simple Unix-like OS kernel
  - Make interface finite, “exokernel”-y
  - Leverage advances in HW and formal methods
- Starting point for verifying applications+OS

Lessons learned

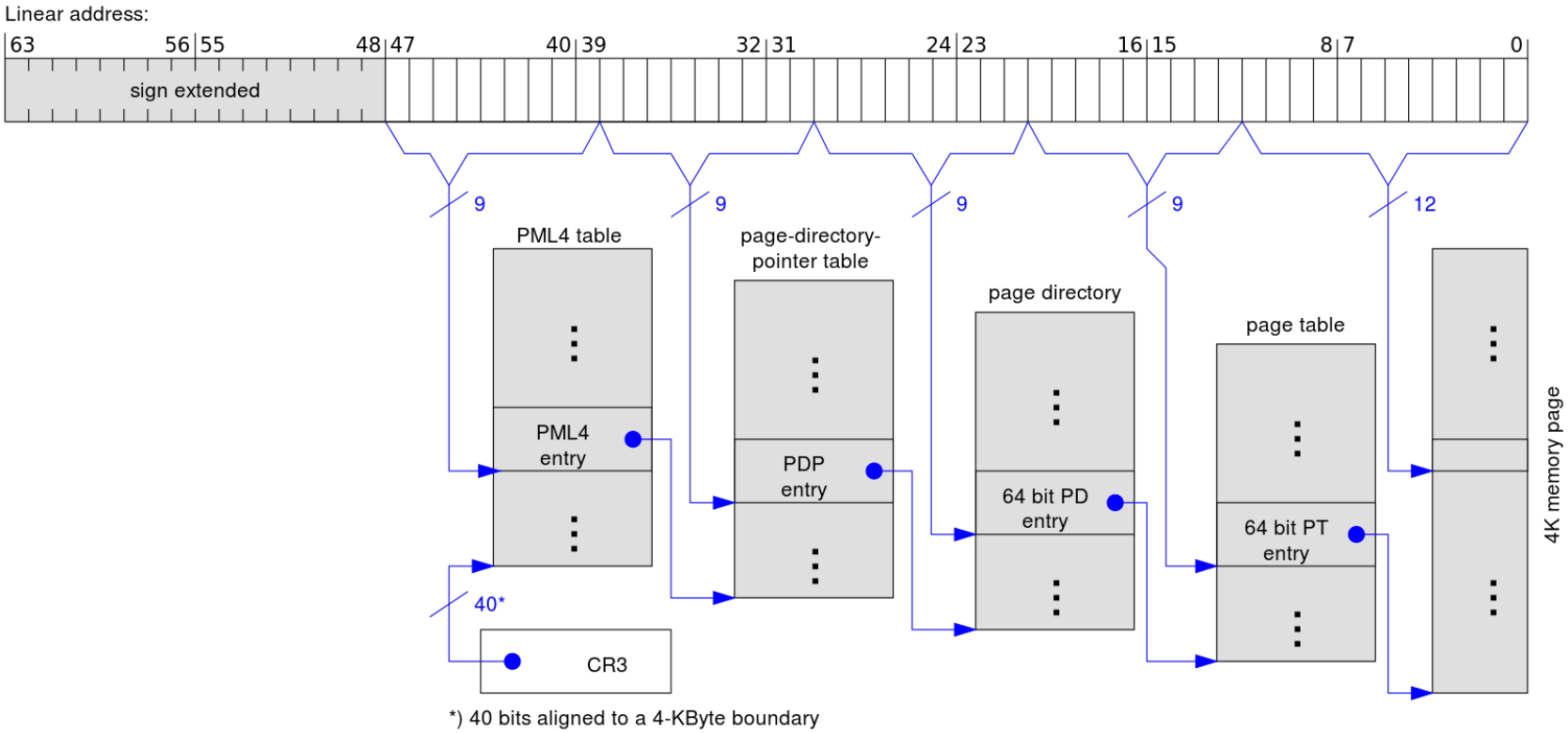
- Event-driven systems
  - A set of “atomic” handlers
  - Encode finite handlers in SMT
  - Add layers (if needed) to scale up verification

- Co-design systems w/ SMT
  - Use effectively decidable theories whenever possible
  - Restricted use of quantifiers

Conclusion

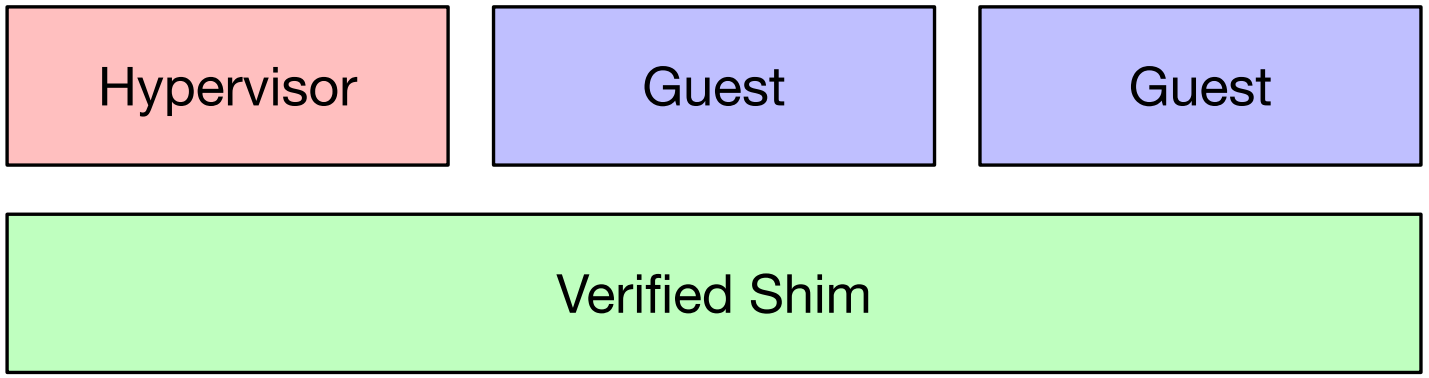
- Push-button verification
  - Examples: file system, OS kernel
  - Reusable design patterns and toolchains

- Verifiability as a first-class system design concern


Linear address:
63
56
55
48
47
40
39
32
31
24
23
16
15
8
7
0
sign extended
9
9
9
9
12
PML4 table
page-directory-pointer table
page directory
page table
4K memory page
PML4 entry
PDP entry
64 bit PD entry
64 bit PT entry
40*
CR3
*) 40 bits aligned to a 4-KByte boundary

Deployability

- Run a hypervisor as a guest on a verified shim.
- Enforce memory is protected from other guests and from hypervisor.
- Rely on hypervisor for device and policy implementation.



Key design ideas

- Explicit resource allocation and reclamation
  - Require user space to make decisions about resources, eliminating need for allocators or garbage collectors in kernel
- Finitize system call interface
  - Should complete in constant time, independent of parameters, eliminating need to reason about loops or long-running system calls