

Optimistic Incremental Specialization: Streamlining a Commercial Operating System

Oregon Graduate Institute of Science & Technology

Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan,
Jon Inouye, Lakshmi Kethana, Jonathan Walpole, Ke Zhang

Presented by: David Tam

Introduction

Conventional OS:

- Very general, handles all possible events
- Lots of “interpretation”
 - Determine system state
 - Check permissions
 - Acquire locks
- Sometimes, lots of overhead to perform little “real” work

Use Specialization:

- Target: optimize where little “real” work done by OS

Incremental Specialization:

- Specialize when have enough information to do so

Optimistically Specialize:

- Conditions true most of the time but not all
 - Can revert to generic version

Paper Context

Authors' Motivation

Synthesis OS:

- Did most of work already
- Performance gains due to many factors
- Could not separate gains due to specialization
- Custom OS

New contributions:

Paper describes work done in the Synthetix project.

1. Specialization model
2. Incremental and optimistic
3. Demonstrate possible on traditional OS

Introduction

Specialization:

- a.k.a. partial evaluation
- Customizing code based on input
- Usually applied to the traditional compiler level
- Idea can be applied at a higher level too: OS

Code Generation:

- Costly
- Avoid at runtime
- Generate code templates to be filled at runtime

High-Level Model:

1. Optimistically & incrementally specialize
2. Put guards
3. If assumptions invalid, switch back to generic

Specializing HP-UX read

Overview:

- A representative OS system call
- Challenge: already highly optimized

Specialization Opportunities:

- At open time, many (quasi-)invariants:
 1. file
 2. type
 3. PID of caller
 4. FS block size
 5. inode memory address

Specializing HP-UX read

Control Flow

1. System call startup
2. Identify file & FS type, translate into inode #
3. Lock inode
4. Translate file offset into logical block #
5. Translate logical block # into physical block #,
get target block from buffer cache
- 6. Data transfer**
- yes 7. Do another block?
8. Unlock inode
9. Update file **offset**
10. System call clean up

Legend:

- interpretation, traversal
- locking
- “real” work

Invariants and Quasi-Invariants

<u>(Quasi-)Invariants</u>	<u>Description</u>
FS_CONSTANT	Invariant FS parameters
NS_FP_SHARE	No file pointer sharing
NS_HOLES	No holes in file
NO_INODE_SHARE	No inode sharing
NO_USER_LOCKS	No user-level locks
READ_ONLY	No writers
SEQ_ACCESS	Calls inherit previous offset

- Only true invariant is FS_CONSTANT
- Rest are quasi-invariants
- **Important:** Specialization at file open time

Optimistic Assumptions:

1. Regular files
2. Local FS with 8 KB blocks
3. Specific to caller process

Specializing HP-UX read

Elimination

Control Flow

1. System call startup

FS_CONSTANT

* 2. Identify file & FS type, translate into inode #

FS_INODE_SHARE

* 3. Lock inode

* 4. Translate file offset into logical block #

* 5. Translate logical block # into physical block #,
get target block from buffer cache

6. Data transfer

yes
* 7. Do another block?

FS_INODE_SHARE

* 8. Unlock inode

NO_FP_SHARE

9. Update file **offset**

10. System call clean up

SEQ_ACCESS*

Legend:

- interpretation, traversal
- locking
- “real” work

Specializing HP-UX read

Specialized Control Flow (is_read)

1. System call startup

2a. Same block?

4. Translate file offset into logical block #

5. Translate logical block # into physical block #,
get target block from buffer cache

6a. Data transfer

yes
7. Do another block?

6b. Data transfer

9. Update file offset

10. System call clean up

Legend:

- interpretation, traversal
- locking
- “real” work

yes

yes



Guards

Guards:

- For quasi-invariants
- If triggered, invoke “replugging” to unspecialize

Quasi-Invariant

NO_FS_SHARE

NO_HOLES

NO_INODE_SHARE

READ_ONLY

SEQ_ACCESS

System Calls that may Invalidate

creat, dup, dup2, fork, sendmsg

open

creat, fork, open, truncate

open

lseek, readv, is_read,

buffer cache block replacement

Replugging

Challenging:

1. Actively running kernel
2. Multi-processor

Handle Concurrency Between:

1. Replugger and process being replugged
2. Kernel threads invoking multiple concurrent repluggers

1st Piece of Infrastructure:

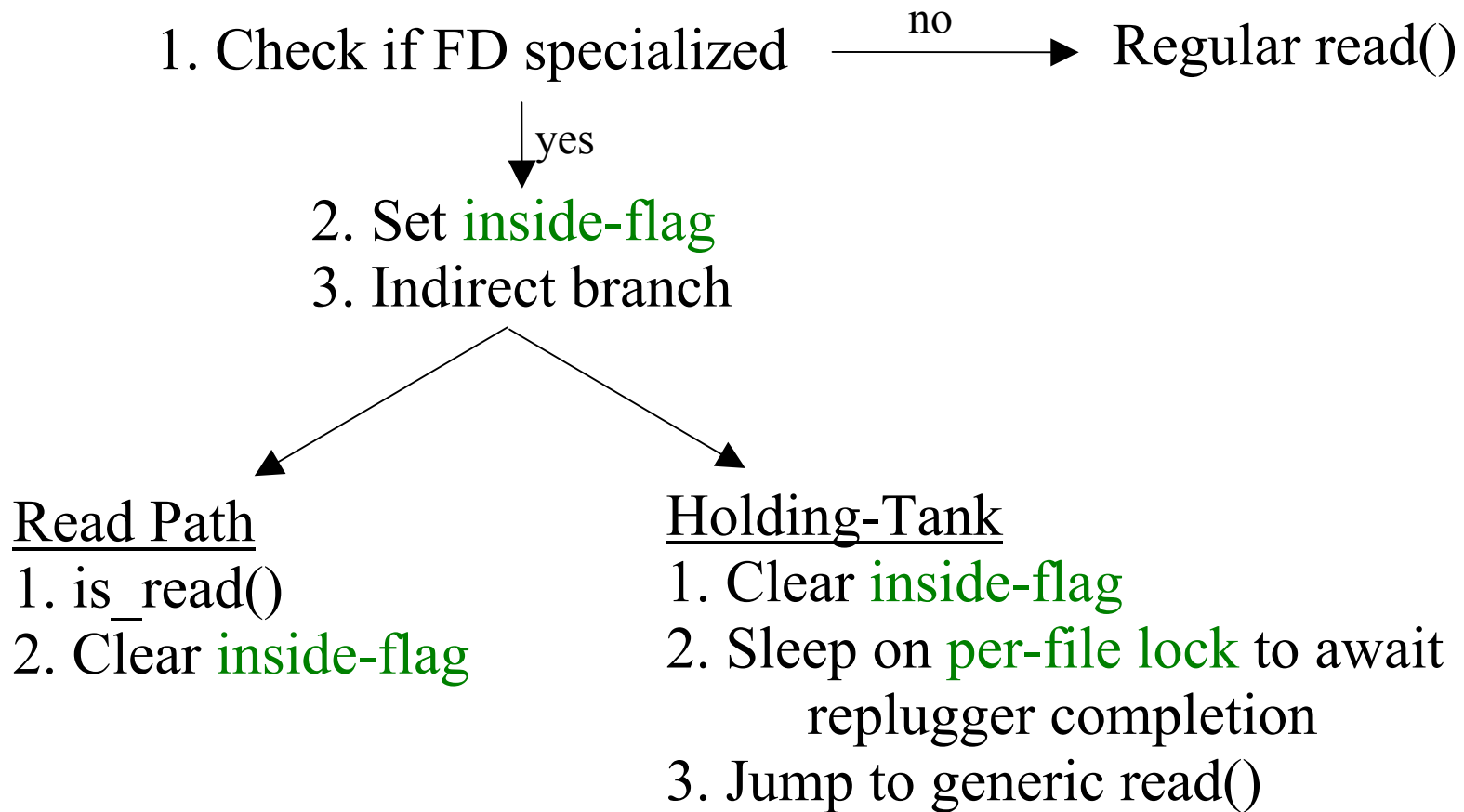
- Signify whether a thread is in `is_read()`
- In `is_read()`:
 1. Set “**inside-flag**” upon entry
 2. Clear “**inside-flag**” upon exit

2nd Piece of Infrastructure:

- Prevent entry during replugging
- Queue called the “**hold-tank**”

Replugging

read() System Call Actions:



Replugging

Replugger Actions: (for a Target FD)

1. Block other repluggers
(acquire **per-process lock**)
2. Block exit from holding-tank
(acquire **per-file lock**)
3. Change indirect branch target
(for target FD)
4. If necessary, wait for **inside-flag** to be cleared
5. Unspecialize
6. Set FD to unspecialized
7. Unblock holding-tank
(release **per-file lock**)
8. Unblock other repluggers
(release **per-process lock**)


Performance Results

Reducing read() Overhead:

Setup:

1. Warmed FS buffer cache
2. Warmed CPU data caches
3. Enabled optimized copyout()

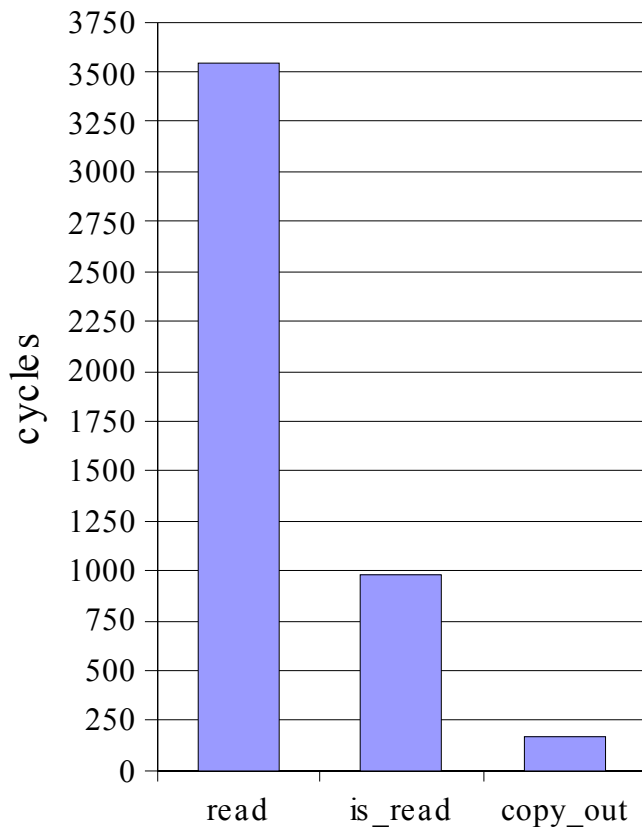
Workload:

- 
1. Open file
 2. Timestamp
 3. Read N bytes
 4. Timestamp
 5. Close file

Performance Results

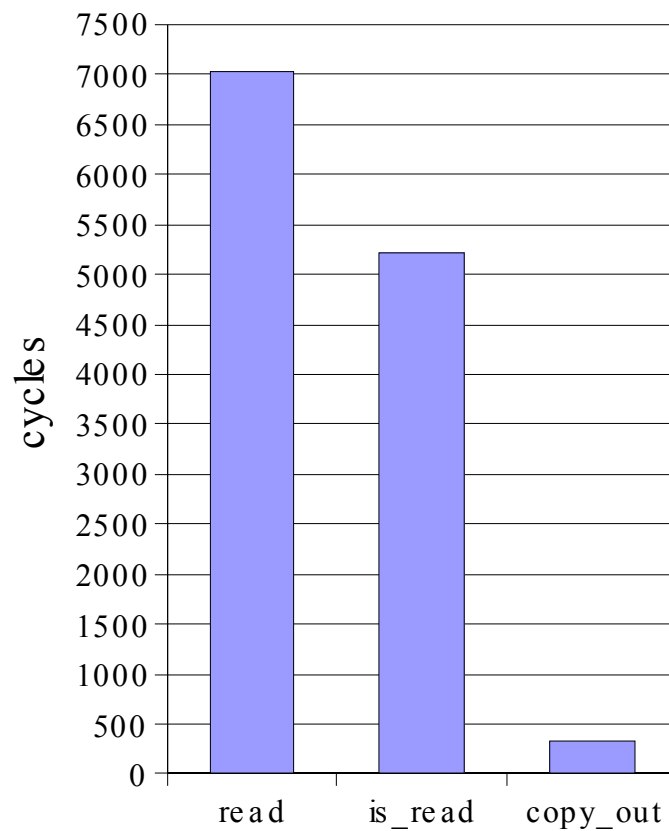
Reducing read() Overhead:

1 byte read()



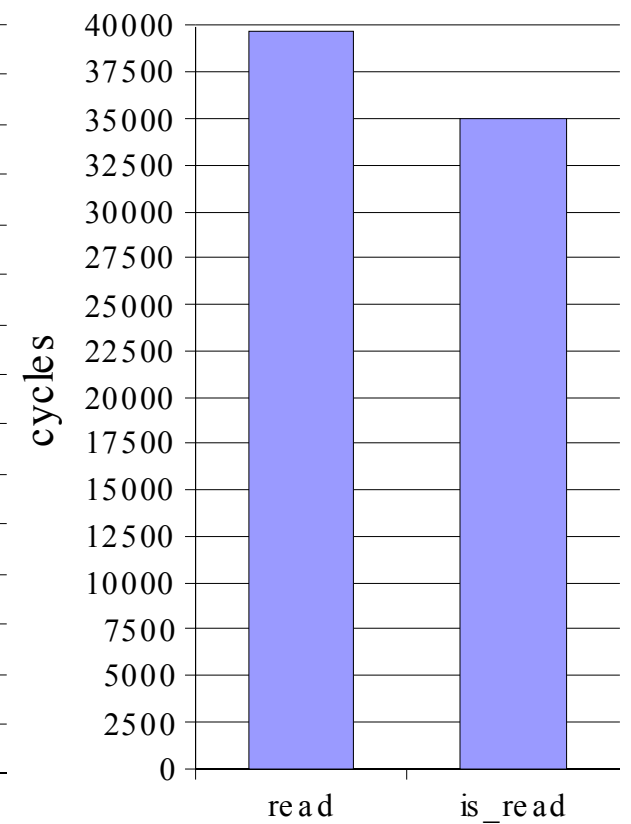
3.61x

8 kB read()



1.35x

64 kB read()



1.13x

Cost of Initial Specialization:

Overhead:

- Adds overhead to other parts of system

Most Significant Addition to open():

- Checks 8 (quasi-)invariants
 - 90 instructions
- 1 lock/unlock pair
- Extra kernel memory allocation (119 cycles)
- **Extra 355 cycles total**
(5582 vs 5227 cycles)

Addition to close():

- Free kernel memory if necessary (**138 cycles**)

Good Trade-Off:

- Even 1 specialized read() call results in savings

Cost of Nontriggered Guards

Two Types:

1. Check for specialized FDs
2. Check for specialized inodes

Inodes:

- Must acquire locks to check
- **145 cycles** for lock/unlock pair

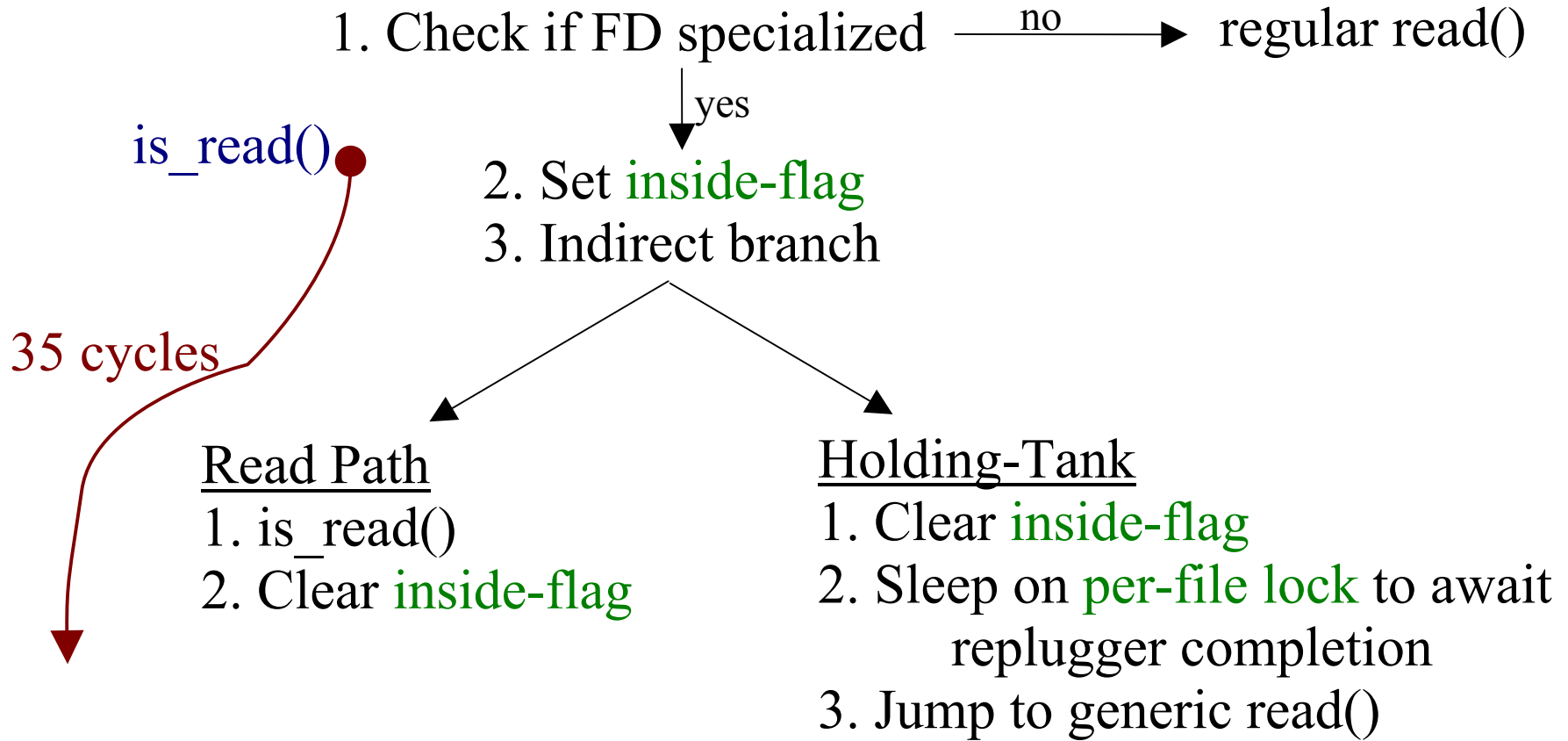
Detailed Costs Per Guard:

- 2 temporary registers
- 2 loads
- 1 add
- 1 compare
- = **11 cycles**

Critical Path:

- Guards are not in critical path
- Not in read() or write()

Cost of Replugging



Cost of Replugging

Unspecialization

- 1 mem location check
- 1 mem location store
- 4 stores
- 1 address generation

- 1 lock/unlock FS table
- 1 store offset in FS table

Replugger Actions: (for a target FD)

- * 1. Block other repluggers
(acquire **per-process lock**)
- * 2. Block exit from holding-tank
(acquire **per-file lock**)
- 3. Change indirect branch target
(for target FD)
- * 4. If necessary, wait for **inside-flag** to be cleared
- 5. Unspecialize
- 6. Set FD to unspecialized
- * 7. Unblock holding-tank
(release **per-file lock**)
- * 8. Unblock other repluggers
(release **per-process lock**)

Total = 535 cycles

Discussion

Missing Cycles: (1 byte read)

Lower bound = 235 cycles

Results = 979 cycles

Missing 744 cycles due legacy support

Without Legacy:

- 1 byte user-level `getc()` = 38 cycles
- 1 byte specialized read = 65 cycles

Claims:

- Not fully implemented
- Conservative results

Challenges:

1. Correctly placing guards
2. Policies to trigger specialization
 - what, when

Related Work

Micro-Kernel OS:

- Mach, V, QNX, Amoeba, Chorus
- Enables specialization of OS
- OS services implemented as user-level servers
- Performance problems

Loadable Kernel Modules:

- Linux, Chorus, Flex, SPIN
- Security issues

Object-Oriented OS:

- Choices, Spring, Tornado, K42
- Inheritance, invocation re-direction, meta-interfaces

Scout OS:

- Specialization for networking code

Conclusions

Contributions:

- Introduced a model for specialization:
(quasi-)invariants, guards, replugging
- Incremental and optimistic approach
- Demonstrated significant improvements possible:
 1. on a traditional system
 2. on an already highly-optimized system
 3. maintain system semantics
 4. transparent to application

Future Work:

- Specialize copyout()
- Automate guard placement
- Tools to do cost/benefit analysis