第 5 讲：The Interface of OS

第五节：Interface for Performance

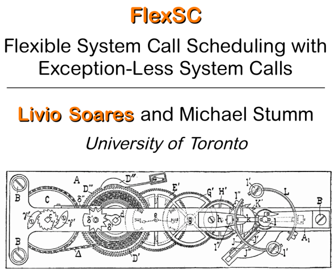
陈渝

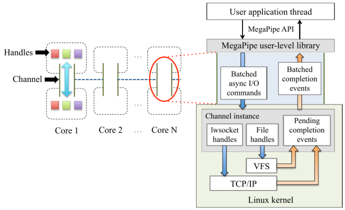
清华大学计算机系

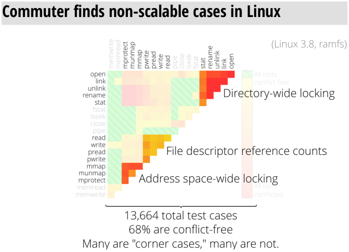
yuchen@tsinghua.edu.cn

2020 年 3 月 15 日

Introduction







Reference:

Flexsc: Flexible System Call Scheduling with Exception-Less System Calls,OSDI 2010

MegaPipe: A New Programming Interface for Scalable Network I/O, OSDI 2012

The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors, SOSP 2013.

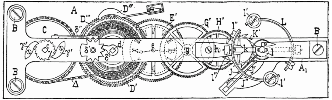
Introduction

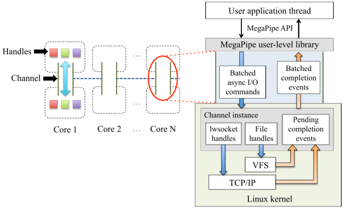
FlexSC

Flexible System Call Scheduling with Exception-Less System Calls

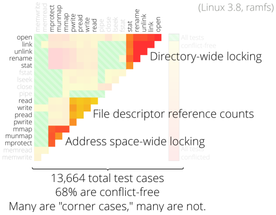
Livio Soares and Michael Stumm

University of Toronto





Commuter finds non-scalable cases in Linux



Improve performance:

- synchronmous syscall is a legacy
- syscall is not for high-speed net
- syscall is not for multicore arch

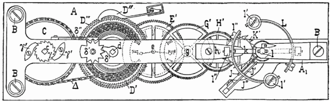
The **synchronous** system call interface is a legacy from the single core era

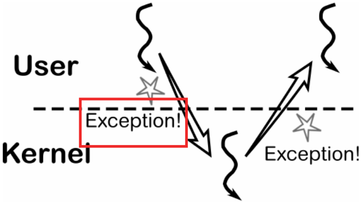
FlexSC

Flexible System Call Scheduling with Exception-Less System Calls

Livio Soares and Michael Stumm

University of Toronto





Expensive! Costs are:

- **direct**: mode-switch
- **indirect**: processor structure pollution

FlexSC implements **efficient and flexible** system calls for the multicore era

Introduction – FlexSC

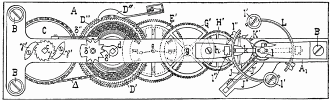
FlexSC

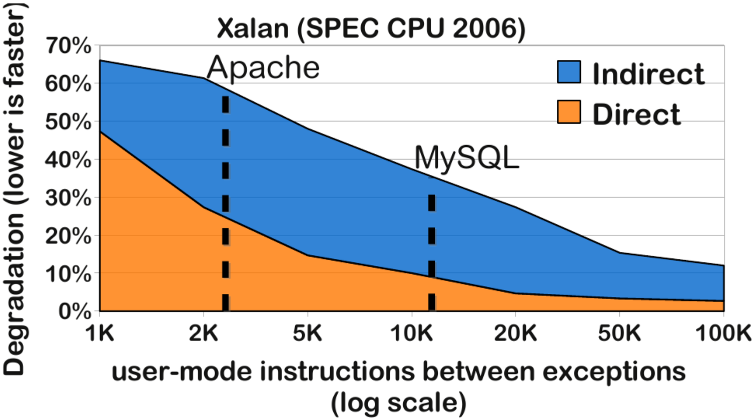
Flexible System Call Scheduling with Exception-Less System Calls

---

Livio Soares and Michael Stumm

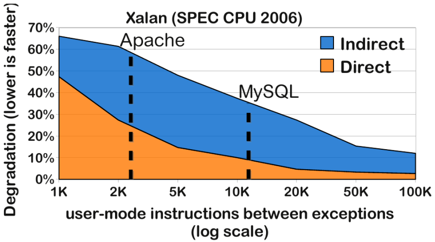
University of Toronto





System calls can **half** processor efficiency; **indirect** cause is major contributor

Introduction – FlexSC

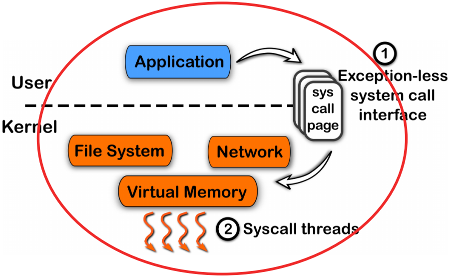


System calls can **half** processor efficiency; **indirect** cause is major contributor

Key source of performance impact

- Traditional system calls are synchronous and use exceptions to cross domains
- Kernel performance equally affected. Processor efficiency for OS code is also cut in half
- On a Linux write() call: up to 2/3 of the L1 data cache and data TLB are evicted

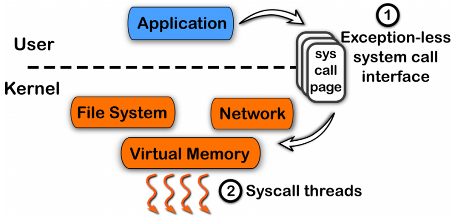
Introduction – FlexSC



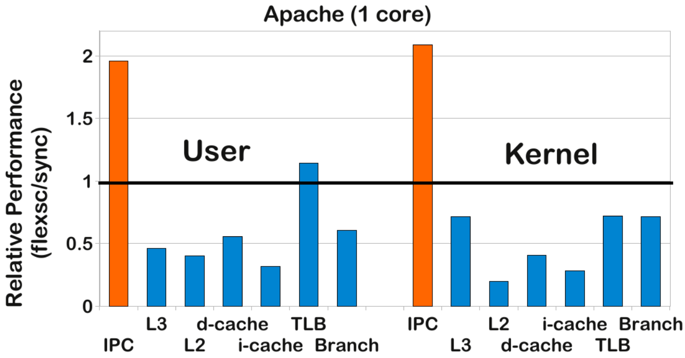
Exception-less syscalls

- Remove synchronicity by decoupling invocation from execution
- Allow for batching, reduce indirect costs, fewer mode switches
- Allow for dynamic multicore specialization

Introduction – FlexSC



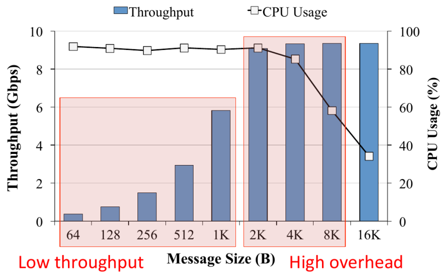
Apache processor metrics

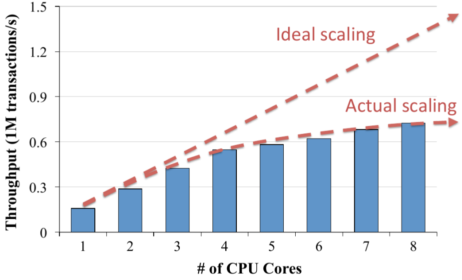


Processor efficiency doubles for kernel and user-mode execution

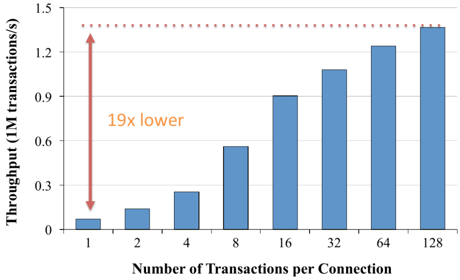
Contributions: Exception-less syscalls & FlexSC-Threads

Introduction – MegaPipe





BSD



Socket API Performance Issues

```
n_events = epoll_wait(…);    // wait for I/O readiness
for (…) {
    …
    new_fd = accept(listen_fd); // new connection
    …
    bytes = recv(fd2, buf, 4096); // new data for fd2
```

Introduction – MegaPipe

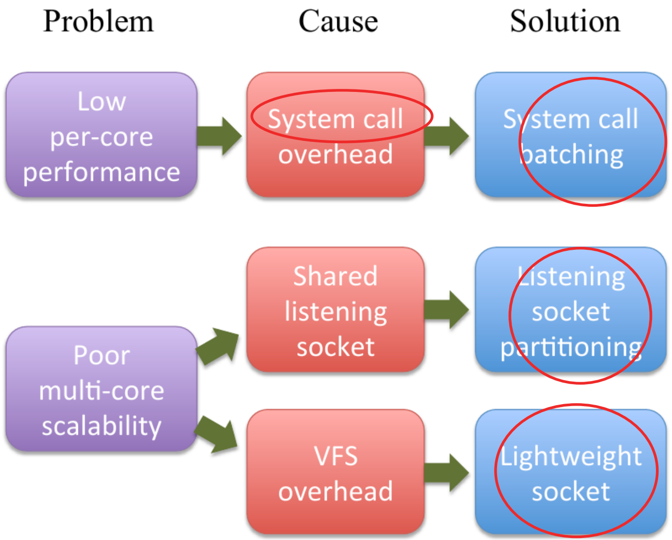
BSD Socket API Performance Issues

```
n_events = epoll_wait(…);    // wait for I/O readiness
for (…) {
    …
    new_fd = accept(listen_fd); // new connection
    …
    bytes = recv(fd2, buf, 4096); // new data for fd2
```

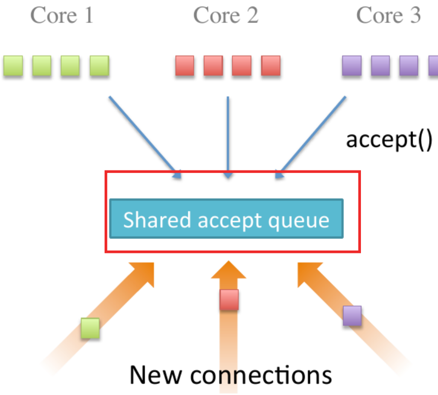
Issues with message--oriented workloads

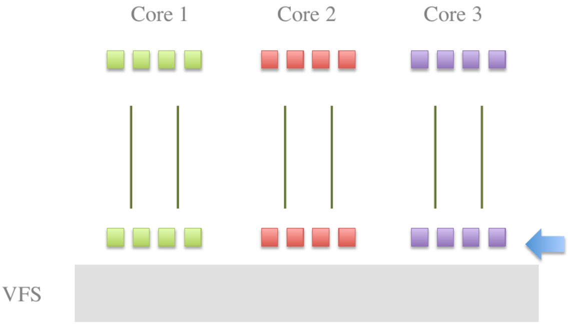
- System call overhead
- Shared listening socket
- File abstraction overhead

Introduction – MegaPipe



Introduction – MegaPipe





Completion Notification Model

- BSD Socket API
  - Wait-and-Go (Readiness model)

```
epoll_ctl(fd1, EPOLLIN);
epoll_ctl(fd2, EPOLLIN);
epoll_wait(…);
…
ret1 = recv(fd1, …);
…
ret2 = recv(fd2, …);
…
```

☹

- MegaPipe
  - Go-and-Wait (Completion notification)

```
mp_read(handle1, …);
mp_read(handle2, …);
…
ev = mp_dispatch(channel);
…
ev = mp_dispatch(channel);
…
```

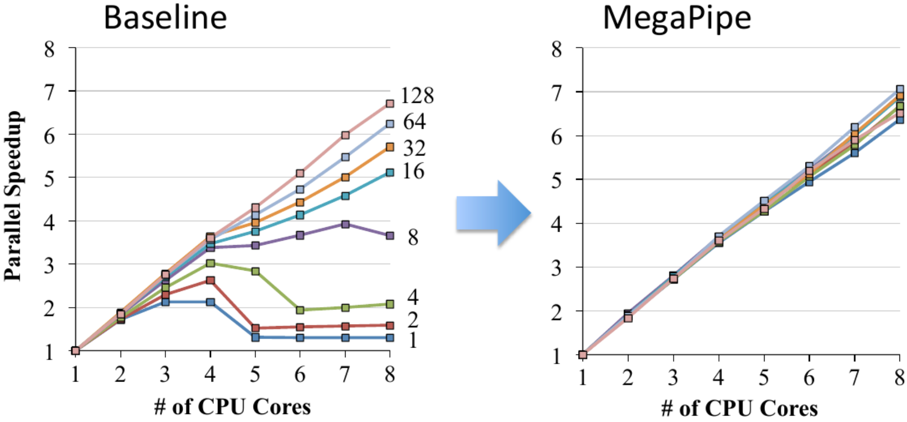
☺

☺ Batching

☺ Easy and intuitive

☺ Compatible with disk files

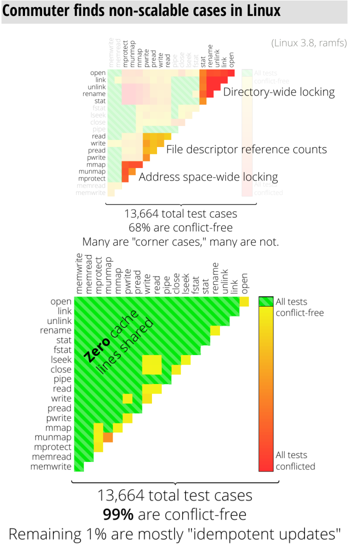
Multi-core scalability (# of transactions)



Key abstraction: per--core channel

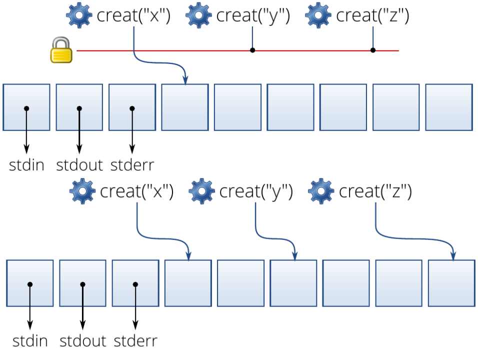
Enabling 3 optimizations: Batching, partitioning, lwsocket

Introduction – Commuter



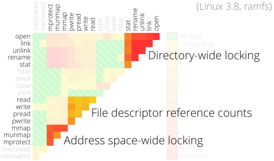
scalable software development

The real bottlenecks may be in the interface design

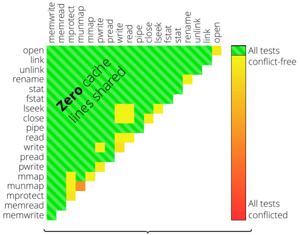


Introduction – Commuter

Commuter finds non-scalable cases in Linux



13,664 total test cases
68% are conflict-free
Many are "corner cases," many are not.



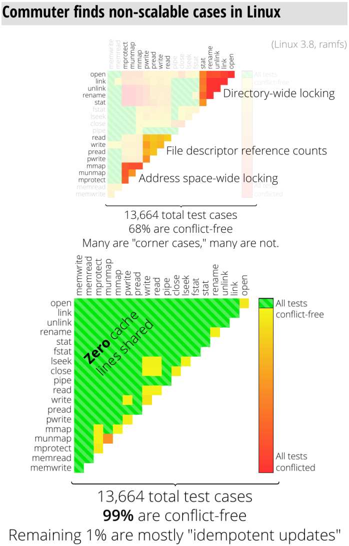
13,664 total test cases
99% are conflict-free
Remaining 1% are mostly "idempotent updates"

The scalable commutativity rule

Whenever $\boxed{\text{interface operations}}$ commute, they can be implemented in a way that scales.

	Commutes		Scalable implementation exists
creat with lowest FD	✗		
creat with any FD	✓	$\xrightarrow{\text{rule}}$	✓

Introduction – Commuter



The scalable commutativity rule

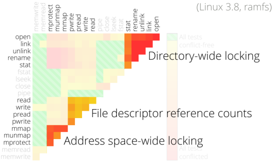
Whenever interface operations commute, they can be implemented in a way that scales.

The rule enables reasoning about scalability throughout the software design process

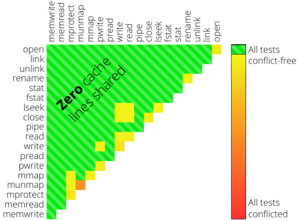
- Design: Guides design of scalable interfaces
- Implement: Sets a clear implementation target
- Test: Systematic, workload-independent scalability testing

Introduction – Commuter

Commuter finds non-scalable cases in Linux



13,664 total test cases
68% are conflict-free
Many are "corner cases," many are not.



13,664 total test cases
99% are conflict-free
Remaining 1% are mostly "idempotent updates"

The scalable commutativity rule

Commutativity is sensitive to operations,arguments, and state

Formalizing the rule

Y *SI-commutes* in $X \,||\, Y :=$

$$\forall\, Y' \in \text{reorderings}(Y),\ Z: X \,||\, Y \,||\, Z \in \mathscr{S} \Leftrightarrow X \,||\, Y' \,||\, Z \in \mathscr{S}.$$

Y *SIM-commutes* in $X \,||\, Y :=$

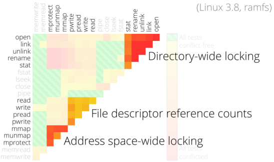
$$\forall\, P \in \text{prefixes}(\text{reorderings}(Y)): P \text{ SI-commutes in } X \,||\, P.$$

An *implementation* m is a step function: $\mathit{state} \times \mathit{inv} \mapsto \mathit{state} \times \mathit{resp}$.

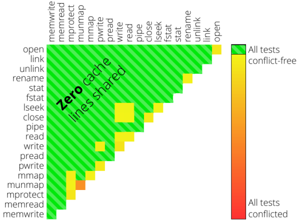
Given a specification $\mathscr{S}$,
a history $X \,||\, Y$ in which Y SIM-commutes,
and a reference implementation M that can generate $X \,||\, Y$,
$\exists$ an implementation m of $\mathscr{S}$ whose steps in Y are conflict-free.

Introduction – Commuter

Commuter finds non-scalable cases in Linux



13,664 total test cases
68% are conflict-free
Many are "corner cases," many are not.



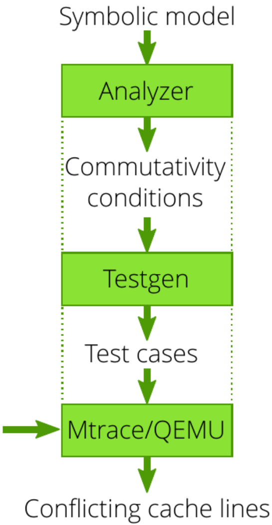
13,664 total test cases
99% are conflict-free
Remaining 1% are mostly "idempotent updates"

The scalable commutativity rule

Commutativity is sensitive to operations,arguments, and state

	Commutes	Scalable implementation exists
P1: creat P1: creat	✗	
P1: creat("/tmp/x") P2: creat("/etc/y")	✓	✓ ~~(Linux)~~
P1: creat("/x") P2: creat("/y")	✓	✓
P1: creat("x", O_EXCL) P2: creat("x", O_EXCL)		
Same CWD	✗	
Different CWD	✓	✓

Introduction – Commuter

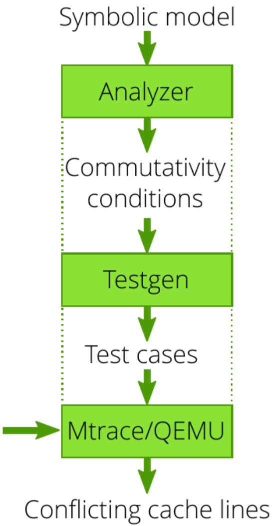



```
@symargs(src=SymFilename, dst=SymFilename)
def rename(self, src, dst):
  if src not in self.fname_to_inum:
    return (-1, errno.ENOENT)
  if src == dst:
    return 0
  if dst in self.fname_to_inum:
    self.inodes[self.fname_to_inum[dst]].nlink -= 1
  self.fname_to_inum[dst] = self.fname_to_inum[src]
  del self.fname_to_inum[src]
  return 0
```

rename(a, b) and rename(c, d) commute if:

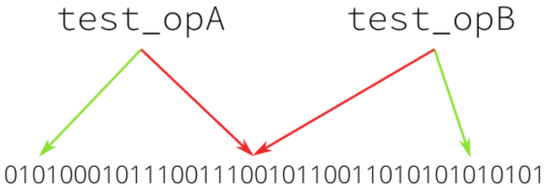
- Both source files exist and all names are different
- Neither source file exists
- a xor c exists, and it is not the other rename's destination
- Both calls are self-renames
- One call is a self-rename of an existing file and a != c
- a & c are hard links to the same inode, a != c, and b == d

Introduction – Commuter

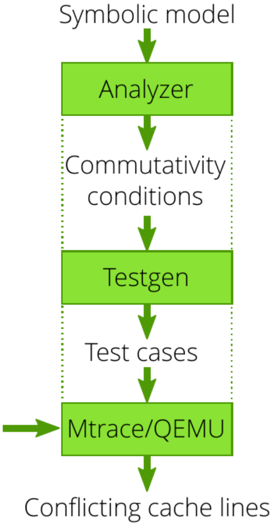


Using the rules to build the scalable OS

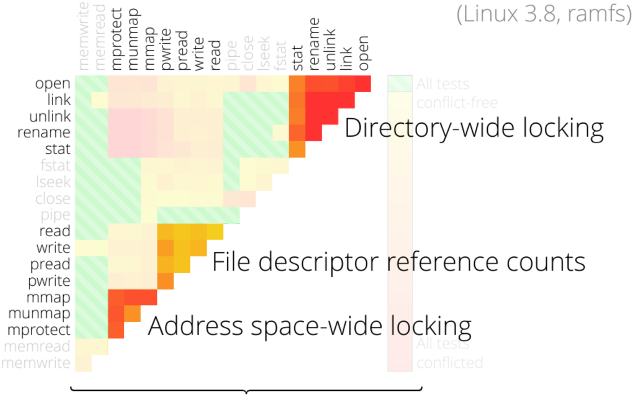
```c
void setup() {
    close(creat("f0", 0666));
    close(creat("f2", 0666));
}
void test_opA() { rename("f0", "f1"); }
void test_opB() { rename("f2", "f3"); }
```



Introduction – Commuter

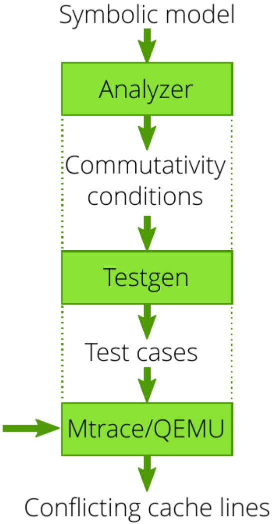


Using the rules to build the scalable OS

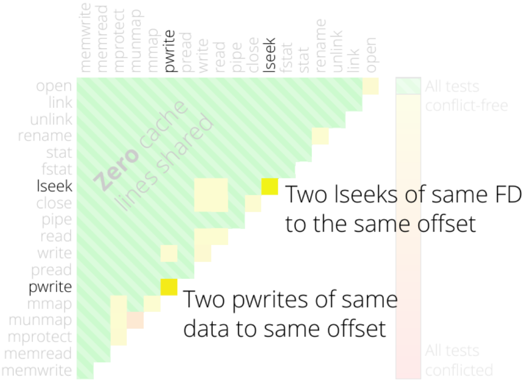


13,664 total test cases
68% are conflict-free
Many are "corner cases," many are not.

Introduction – Commuter



Using the rules to build the scalable OS

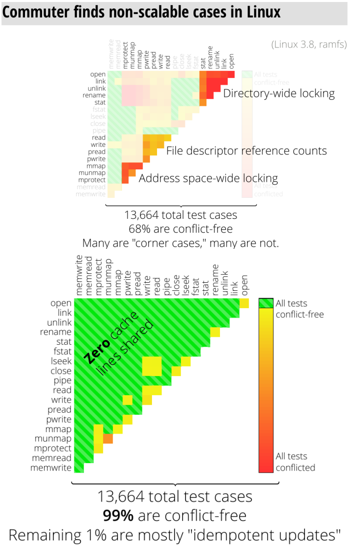


13,664 total test cases

99% are conflict-free

Remaining 1% are mostly "idempotent updates"

Introduction – Commuter



Refining POSIX with the rule

- Lowest FD versus any FD
- stat versus xstat
- Unordered sockets
- Delayed munmap
- fork+exec versus posix_spawn