Xkernel: Principled Performance Tunability of Operating System Kernels

Zhongjie Chen, Wentao Zhang, Yulong Tang
Ran Shu, Fengyuan Ren, Tianyin Xu, Jing Liu

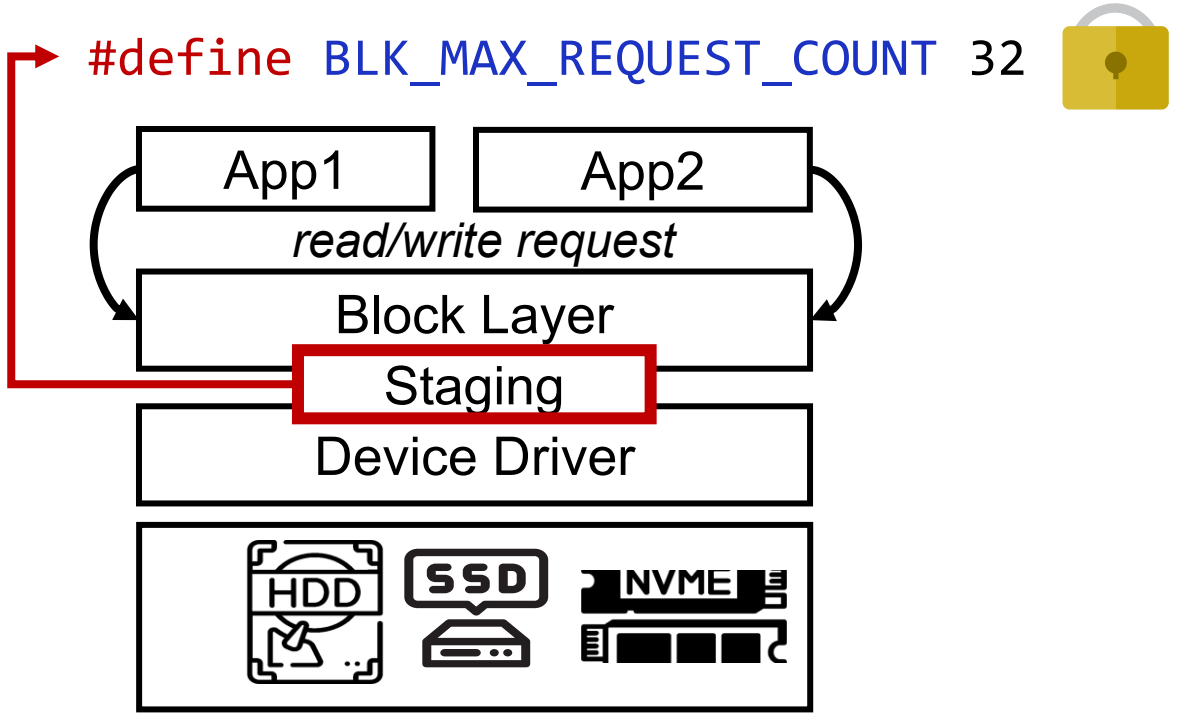






The Tale of a Macro

- Batch size of merging requests in block layer
- Used everywhere and for every I/O
- Fixed at compile time



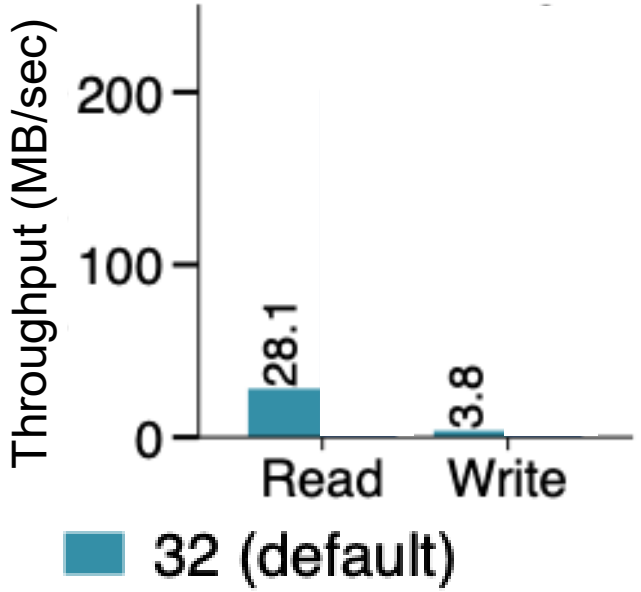
The Tale of a Macro

- The default is suboptimal across hardware and workloads

Sequential I/O access




```
#define BLK_MAX_REQUEST_COUNT 32
```



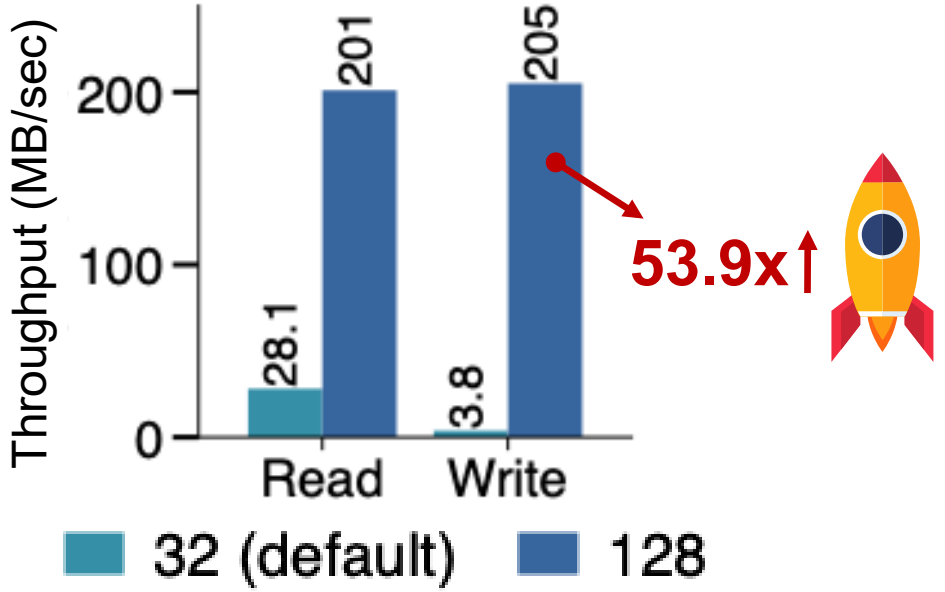
The Tale of a Macro

- The default value is suboptimal across hardware and workloads

Sequential I/O access




```
// #define BLK_MAX_REQUEST_COUNT 32
#define BLK_MAX_REQUEST_COUNT 128
```

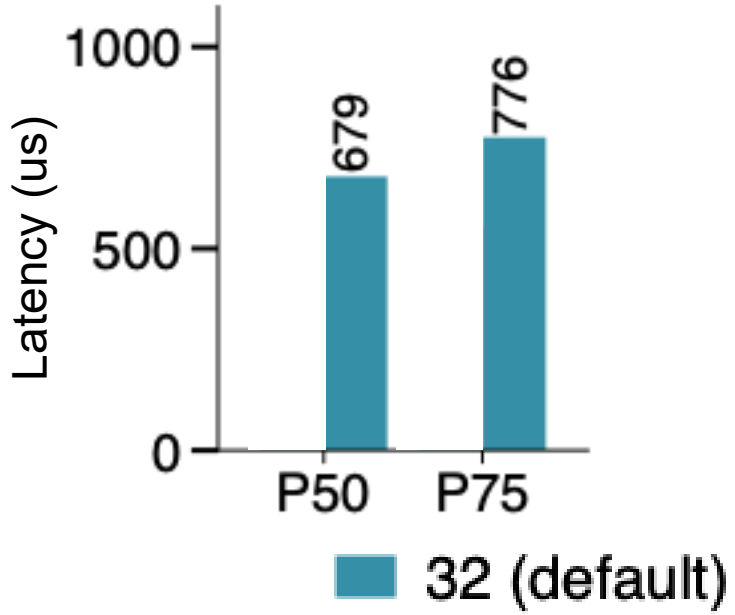


The Tale of a Macro

- The default value is suboptimal across hardware and workloads




```
#define BLK_MAX_REQUEST_COUNT 32
```

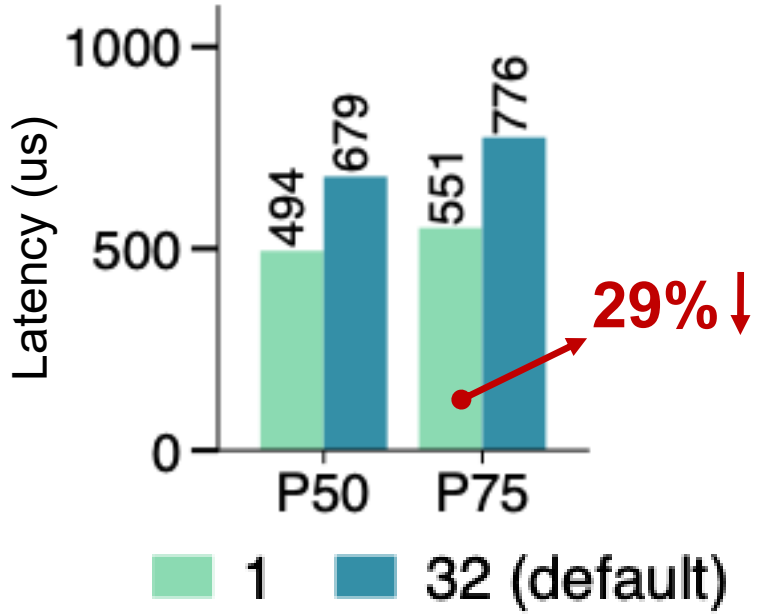


The Tale of a Macro

- The default value is suboptimal across hardware and workloads

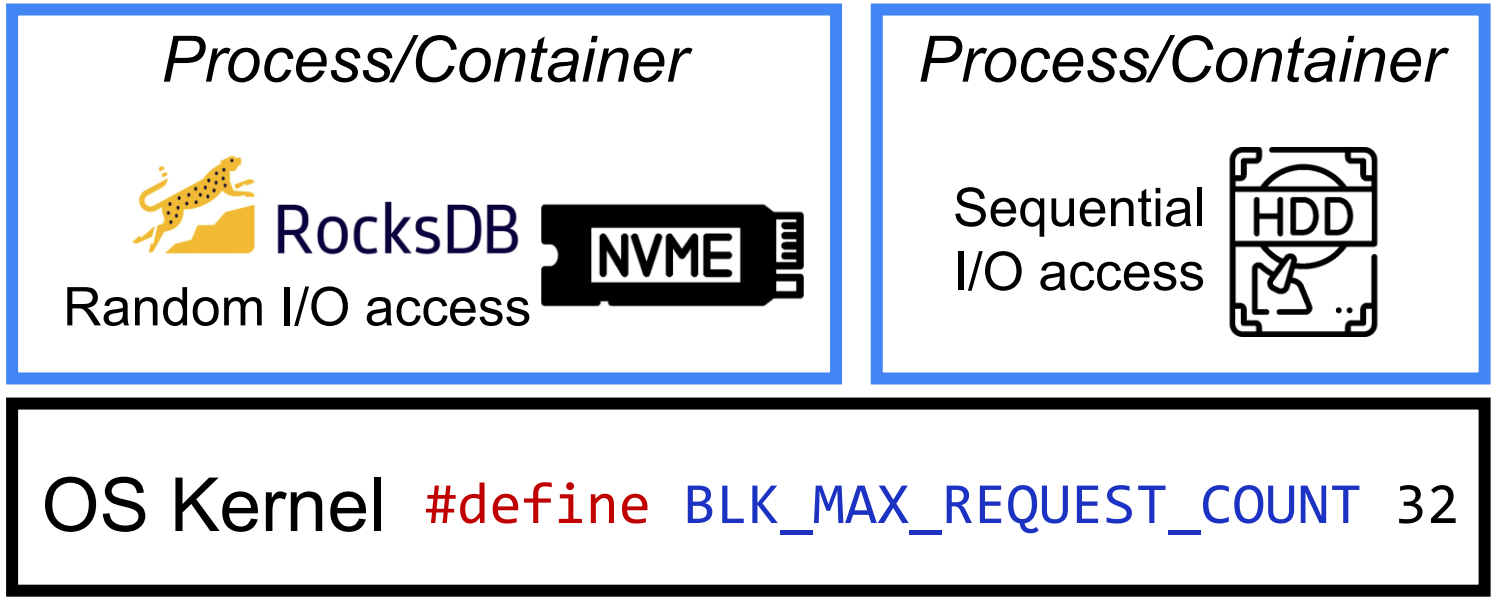



```
// #define BLK_MAX_REQUEST_COUNT 32
#define BLK_MAX_REQUEST_COUNT 1
```



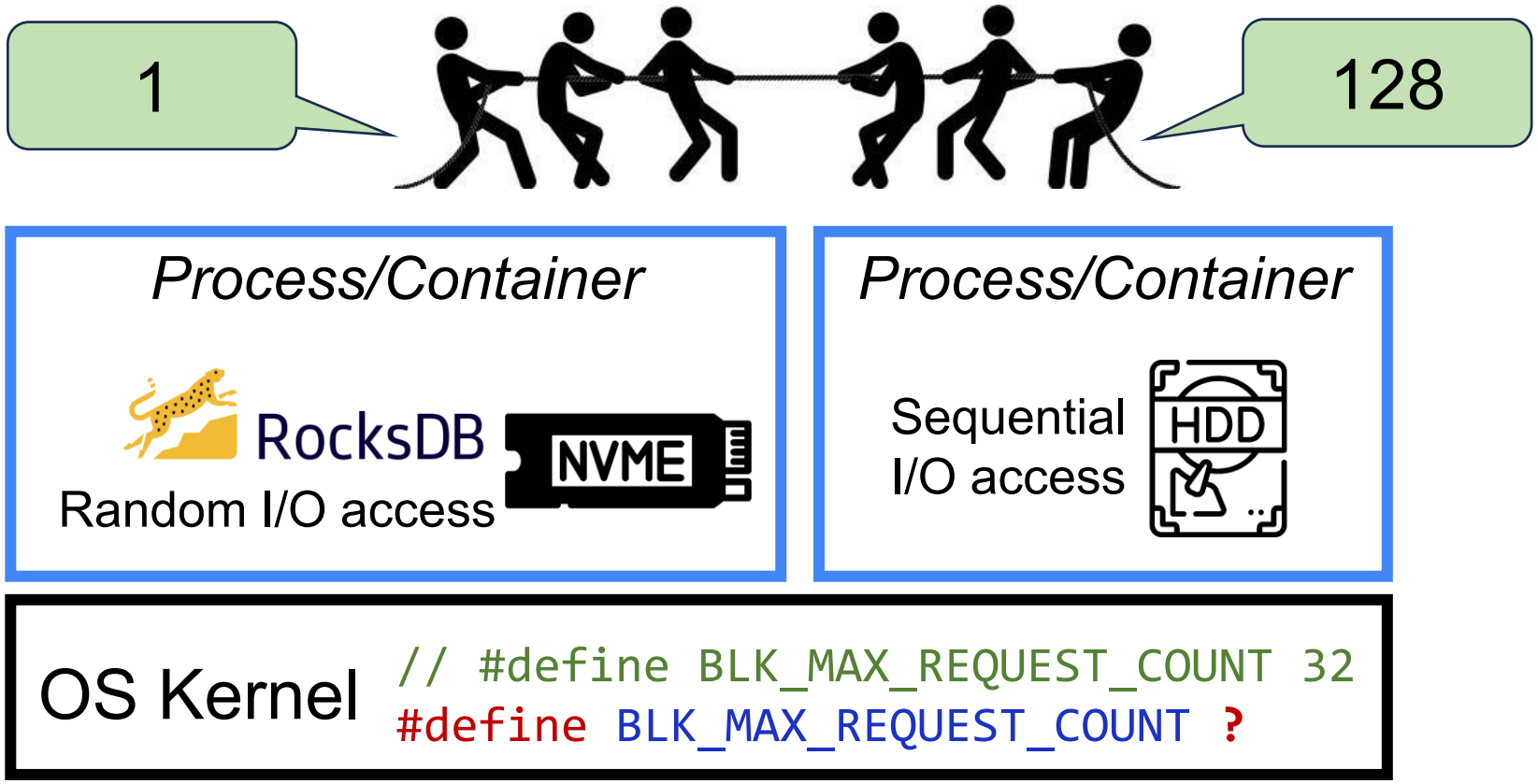
The Tale of a Macro

- Different workloads and hardware run on the same server
  - common in cloud scenarios



The Tale of a Macro

- *Cannot* customize for each simutaneously



The Tale of a Macro

- How was its value chosen?
- ~~It was chosen very carefully by developers…~~

> I thought we'd better have a threshold for requests. When a threshold reaches,
> this means there is no request merge and queue lock contention isn't severe
> when pushing per-task requests to queue, so the main advantages of blk plug
> don't exist. We can force a plug list flush in this case.
> With this, my test throughput actually increases and almost equals to small
> I/O depth. Another side effect is irq off time decreases in blk_flush_plug_list()
> for big I/O depth.
> The BLK_MAX_REQUEST_COUNT is choosen "arbitarily", but 16 is efficiently to
> reduce lock contention to me. But I'm open here, 32 is ok in my test too.

commit: 55c022b

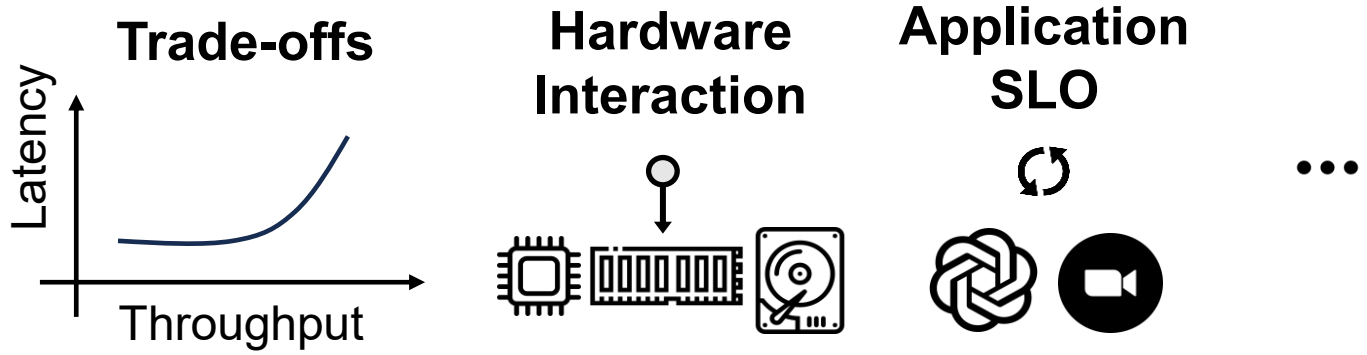
Performance-critical Constants in OS kernels

perf-consts

- They are frozen at compile time

Category	Example
Threshold	#define MAX_SOFTIRQ_RESTART **10**
Interval	#define IPVS_SYNC_SEND_DELAY (HZ/**50**)
Batch Size	static const int bfq_back_max = **16 * 1024**;
Scaling Factor	delta *= **4**;

- They shape system performance

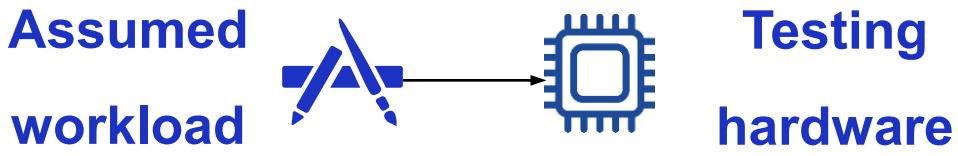


Inherent Mismatch



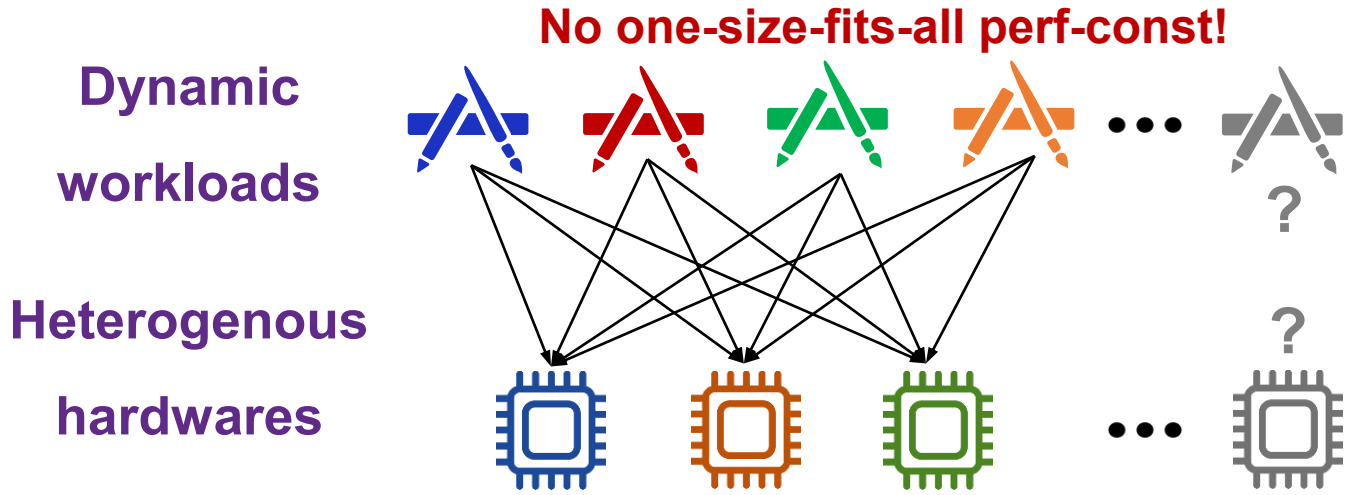
Kernel Developers

- In-house decision and heuristics
- Limited testing and benchmarking





Cloud and datacenter



Tunability Needs to Be Done Right

Flexible heuristics

- e.g., adapt to sequential and random I/O access pattern

Variable-grained policies

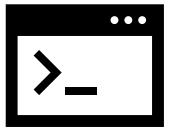
- Per process, per device, per network connection, …

Fast update

- e.g., runtime update without rebooting at the millisecond level

Existing Solutions are Inflexible, Unsafe, and Slow

- **Modify source code** to expose it as a **runtime knob**
  - sysctl/sysfs/syscall, etc.



```
# echo 128 > /proc/sys/kerel/batch_size
```

```
#define BATCH_SIZE 32

if (num >= BATCH_SIZE)
  io(BATCH_SIZE);
```

Source modification

```
static unsigned int global_batch_size=32;

if (num >= global_batch_size)
  io(global_batch_size);
```

Getter

```
int batch_size_setter(unsigned int val)
{
  global_batch_size = val;
}
```

Setter

Existing Solutions are Inflexible, Unsafe, and Slow

```
#define BATCH_SIZE 32

if (num >= BATCH_SIZE)
  io(BATCH_SIZE);
```

Source modification

```
static unsigned int global_batch_size=32;

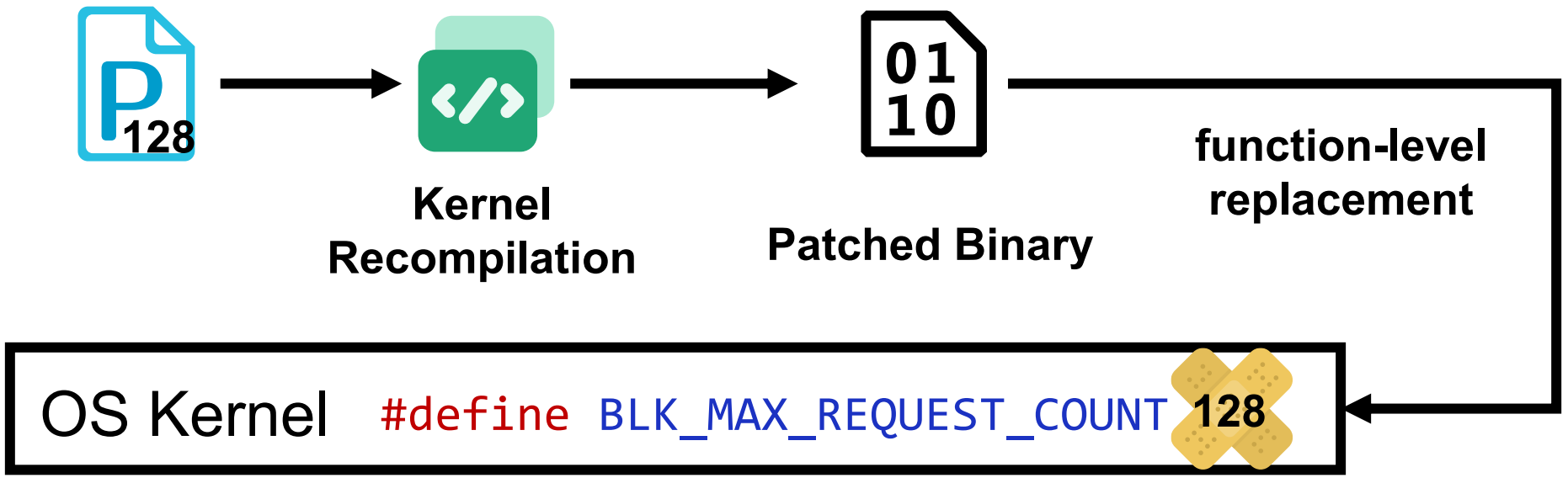
if (num >= global_batch_size)
  io(global_batch_size);
```
Getter

```
int batch_size_setter(unsigned int val)
{
  global_batch_size = val;
}
```
Setter

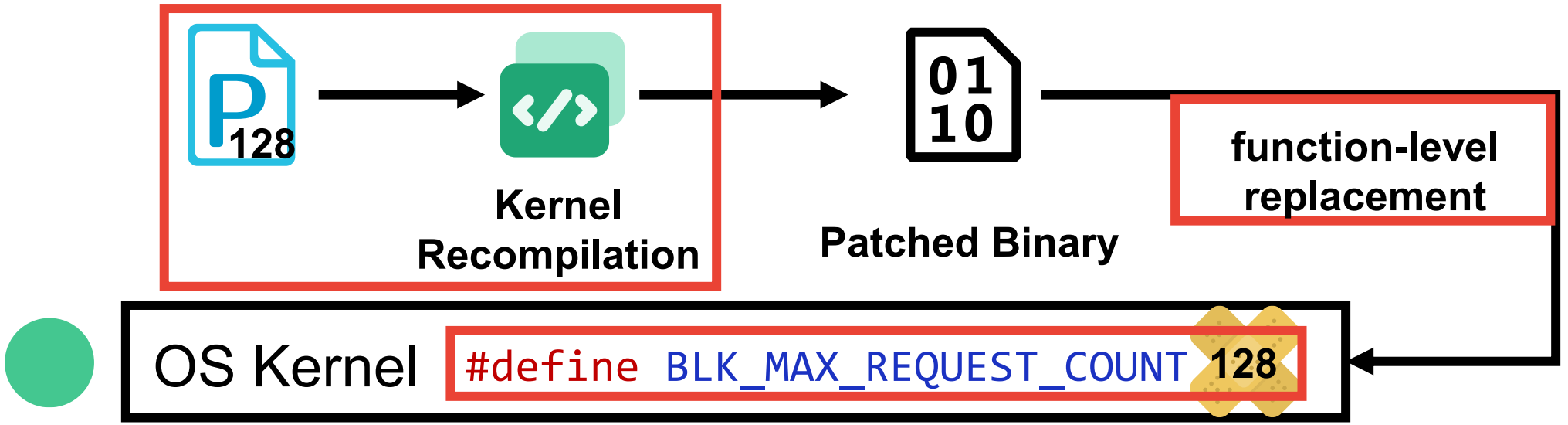
- 👎 Recompilation and reboot
- 👎 No variable-grained policy
- 👎 No side-effect safety
- 👎 Hard to upstream

Existing Solutions are Inflexible, Unsafe, and Slow

- Kernel Live Patching
  - Linux KLP, Kpatch, Ksplice, etc.

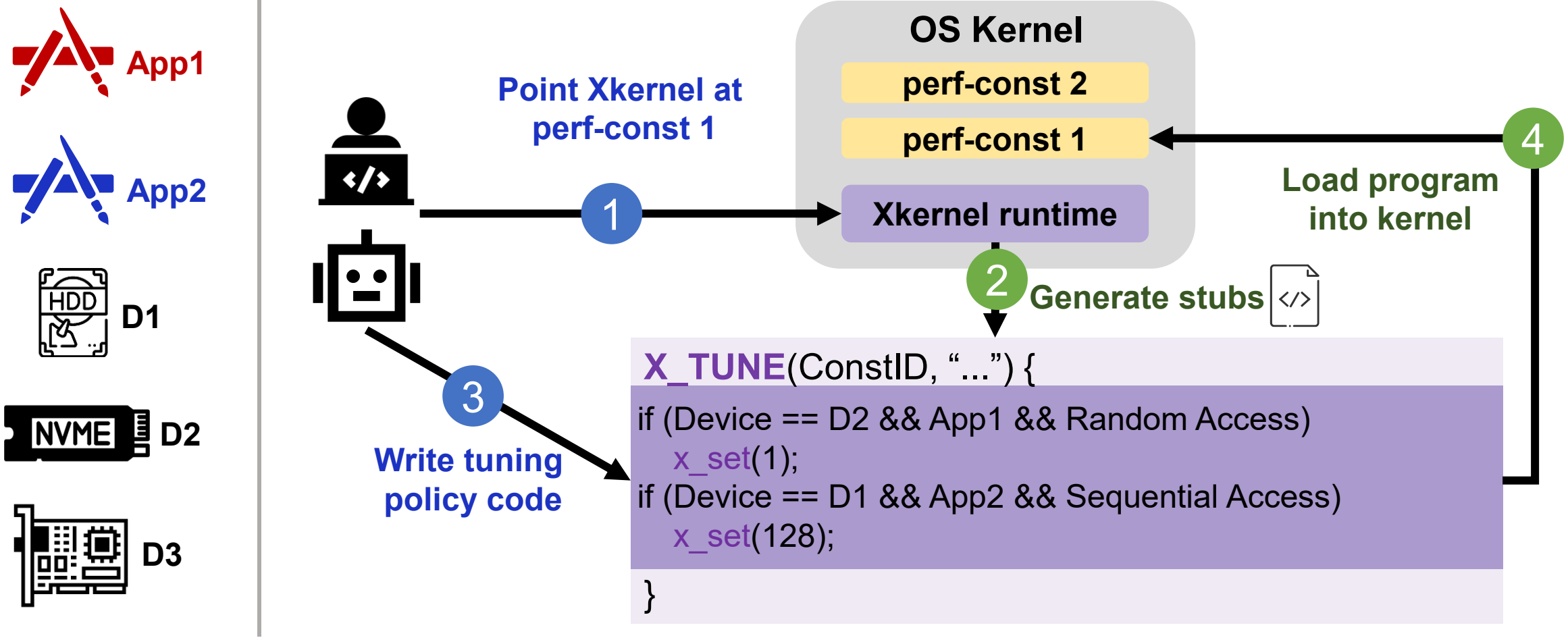


Existing Solutions are Inflexible, Unsafe, and Slow



- No reboot
- No variable-grained policy
- No side-effect safety
- Recompile for every value

New Paradigm of OS Performance Tuning



Xkernel 

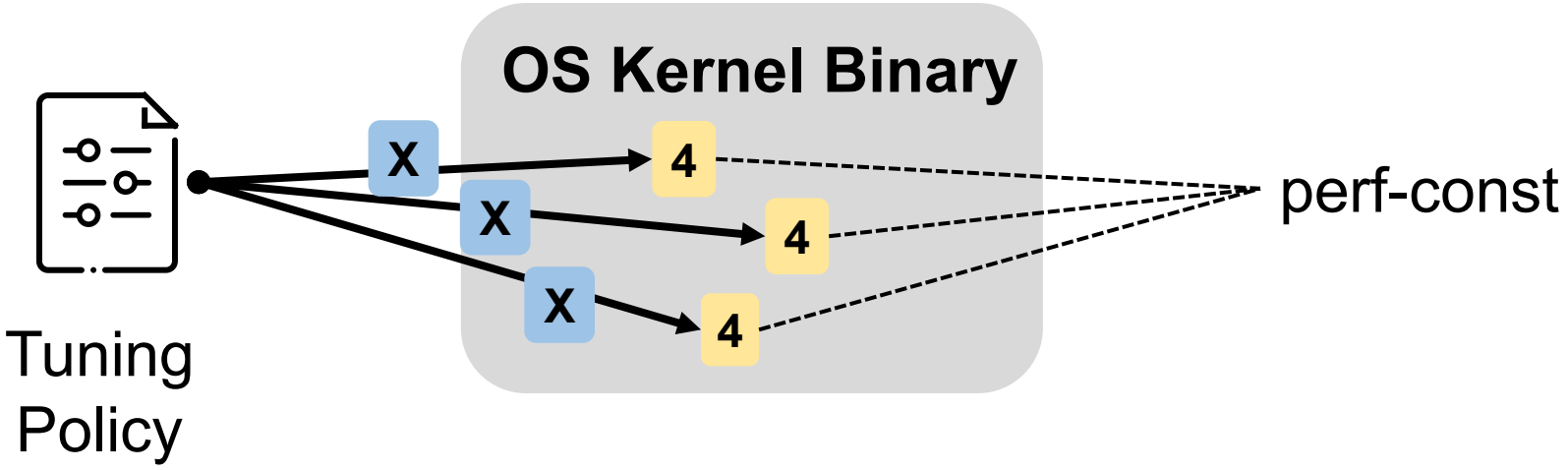
- The first novel mechanism to support **principled tunability**
  - Scoped Indirect Execution (SIE)
- Safe and fast ***in-situ*** **tuning** with programmable policies
  - No recompilation, no reboot
- Deployed directly in an out-of-the-box Linux kernel
- Comprehensive case studies and evaluation
  - Numerous case studies across several OS subsystems
  - Significant performance gains (up to 53.9x)
  - Enable new tuning policies (per-device, heuristic, application-informed)

Outline

- Introduction
- **Challenges**
- Design
- Evaluation

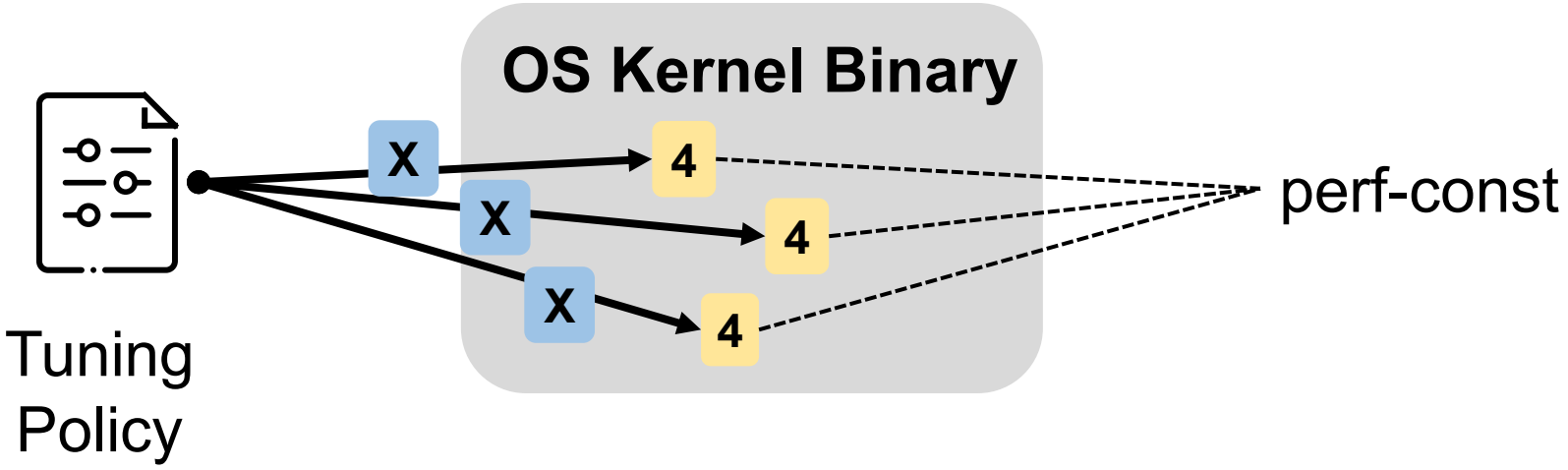
Challenges

- How to locate perf-const in the running OS kernel binary?
- How to generate instructions for new value and policy?



Challenges

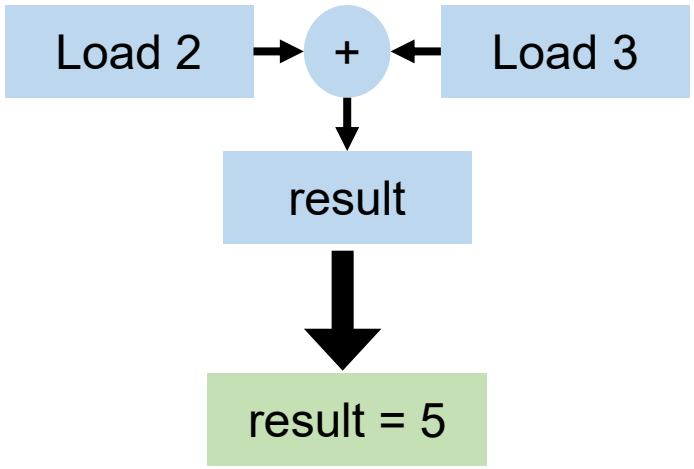
- How to locate perf-const in the running OS kernel binary?
- How to generate instructions for new value and policy?
- How to safely change the original value?



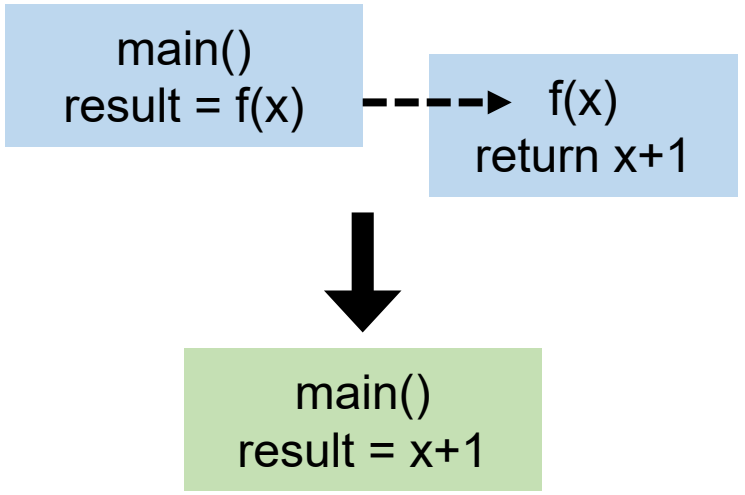
Challenge#1
How to locate instructions for perf-consts?

- Due to **compiler optimization**, you *can not* find instructions of perf-consts directly

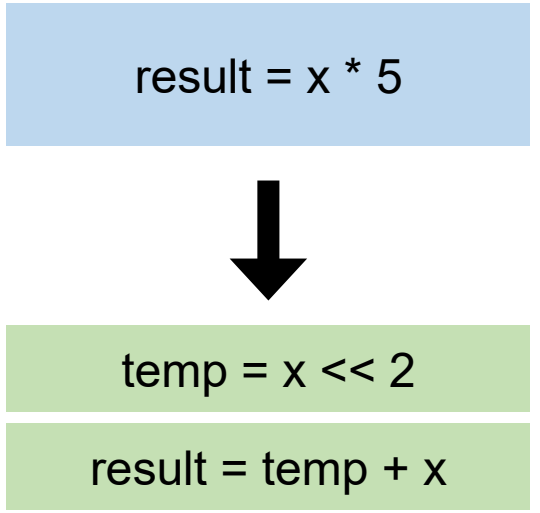
Constant folding



Function inlining

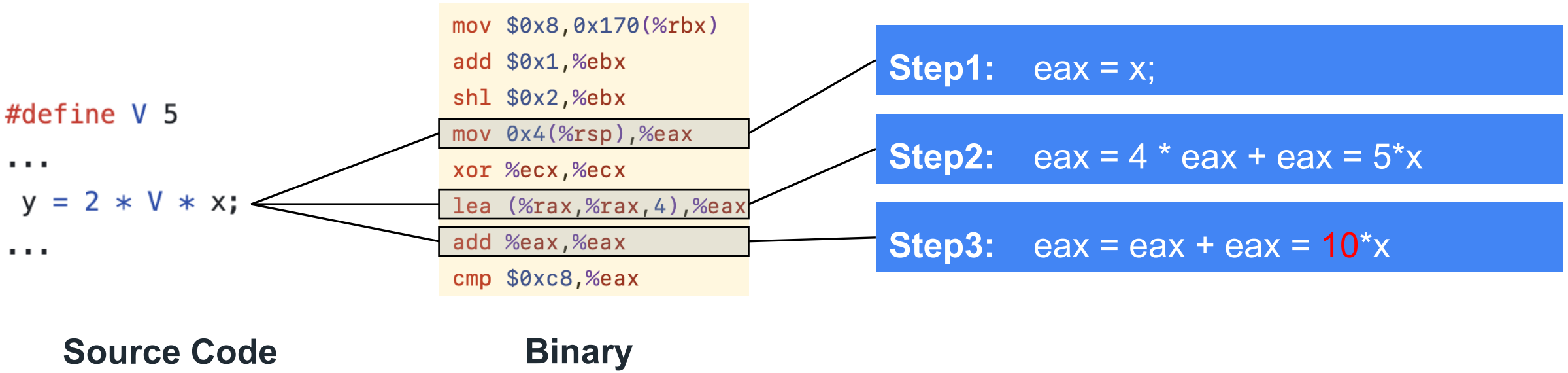


Strength reduction



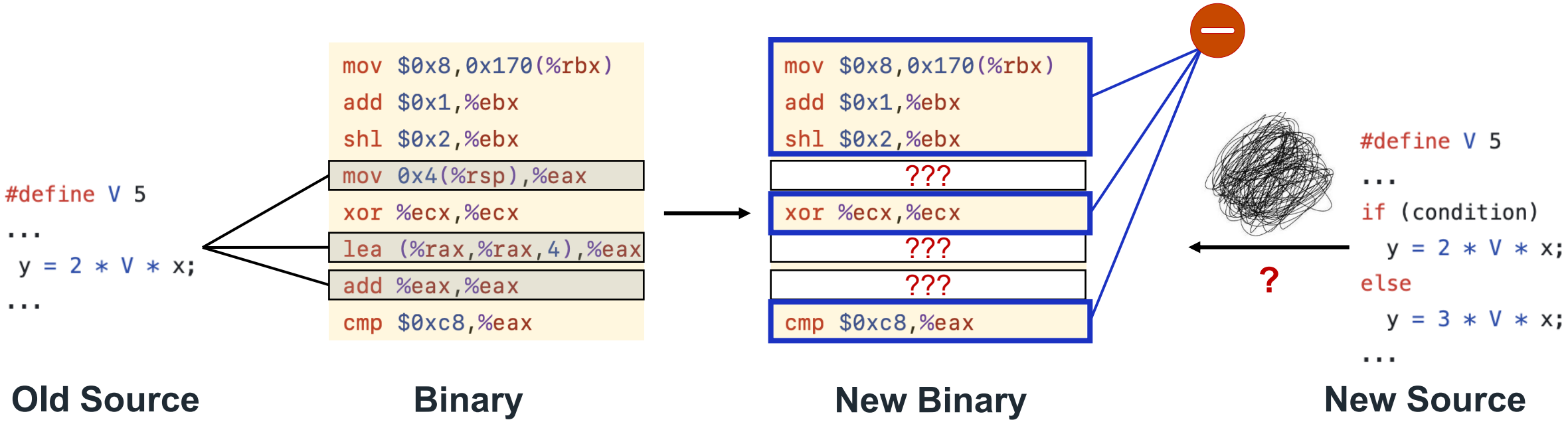
Challenge#1
How to locate instructions for perf-consts?

- Due to **compiler optimization**, you *can not* find instructions of perf-consts directly



Challenge#2
How to generate instructions for value and policy?

- We *can not* corrupt irrelevant register or memory
- We *can not* recompile the kernel for each new value and policy



Challenge#3
How to safely change the original value?

- All occurrences of perf-consts should be changed at the same time
- It's ***not safe*** to switch when the effects of the original value still exist in system states

```
#define PERF_CONST 32
f() {
……
// Use PERF_CONST
          32
……    32 → 128
// Use PERF_CONST
    ⚠   128
}
```

Dependency

Dependency

```
// At the same time,
it runs concurrently
g() {
……    32 → 128
// Use PERF_CONST
    ⚠   128
……
}
```



System crash!

Intra-thread inconsistency

Inter-thread inconsistency

Outline

- Introduction
- Challenges
- **Design**
- Evaluation

Scoped Indirect Execution (SIE)

Key Insight

- The point where a perf-const affects runtime states is *well-scoped,* and can be expressed symbolically:

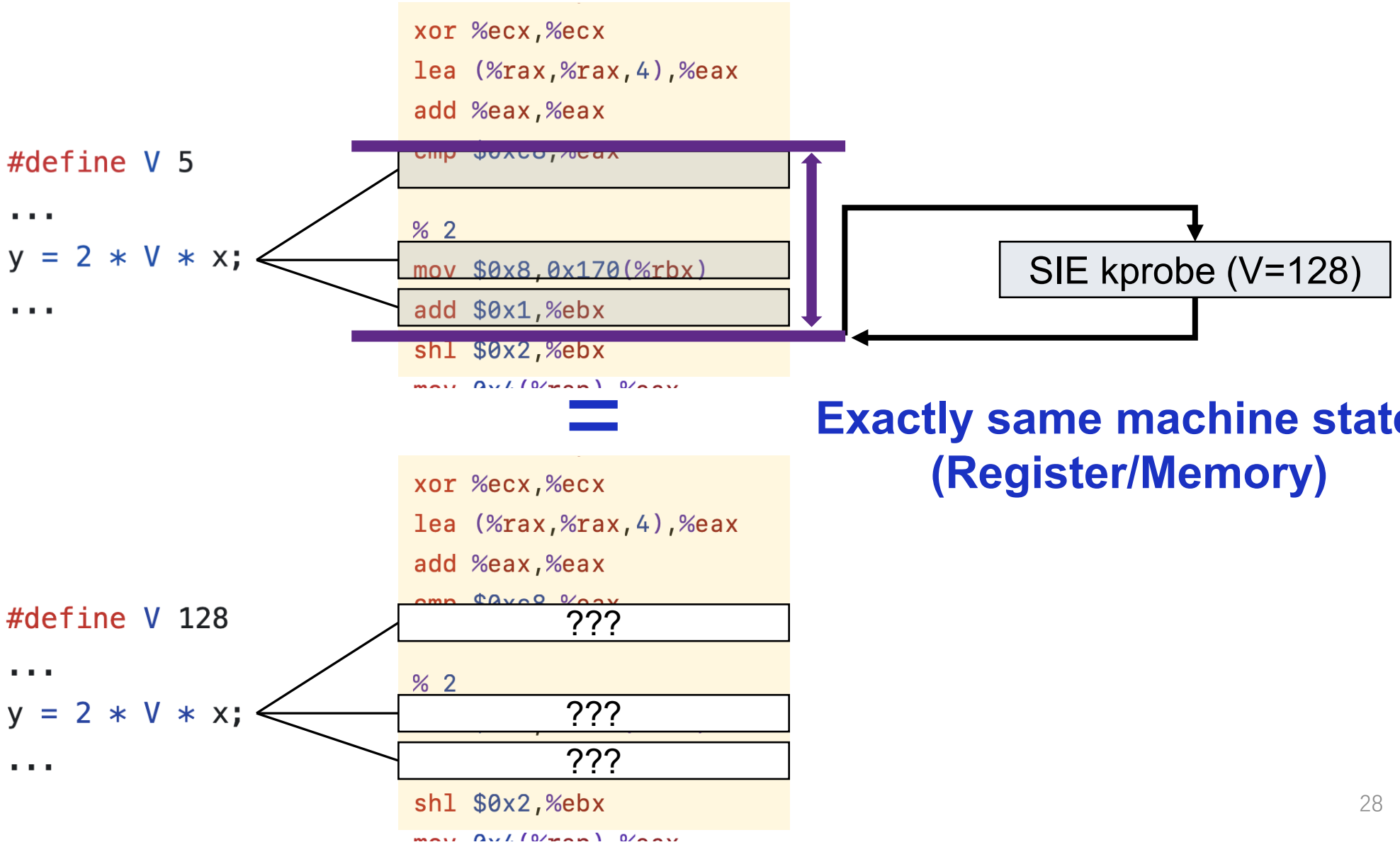
$$\boldsymbol{Reg/Mem \leftarrow f(Reg/Mem,\ V)}$$

- We can synthesize and insert instructions at the **scope**, to modify relevant registers and memory.

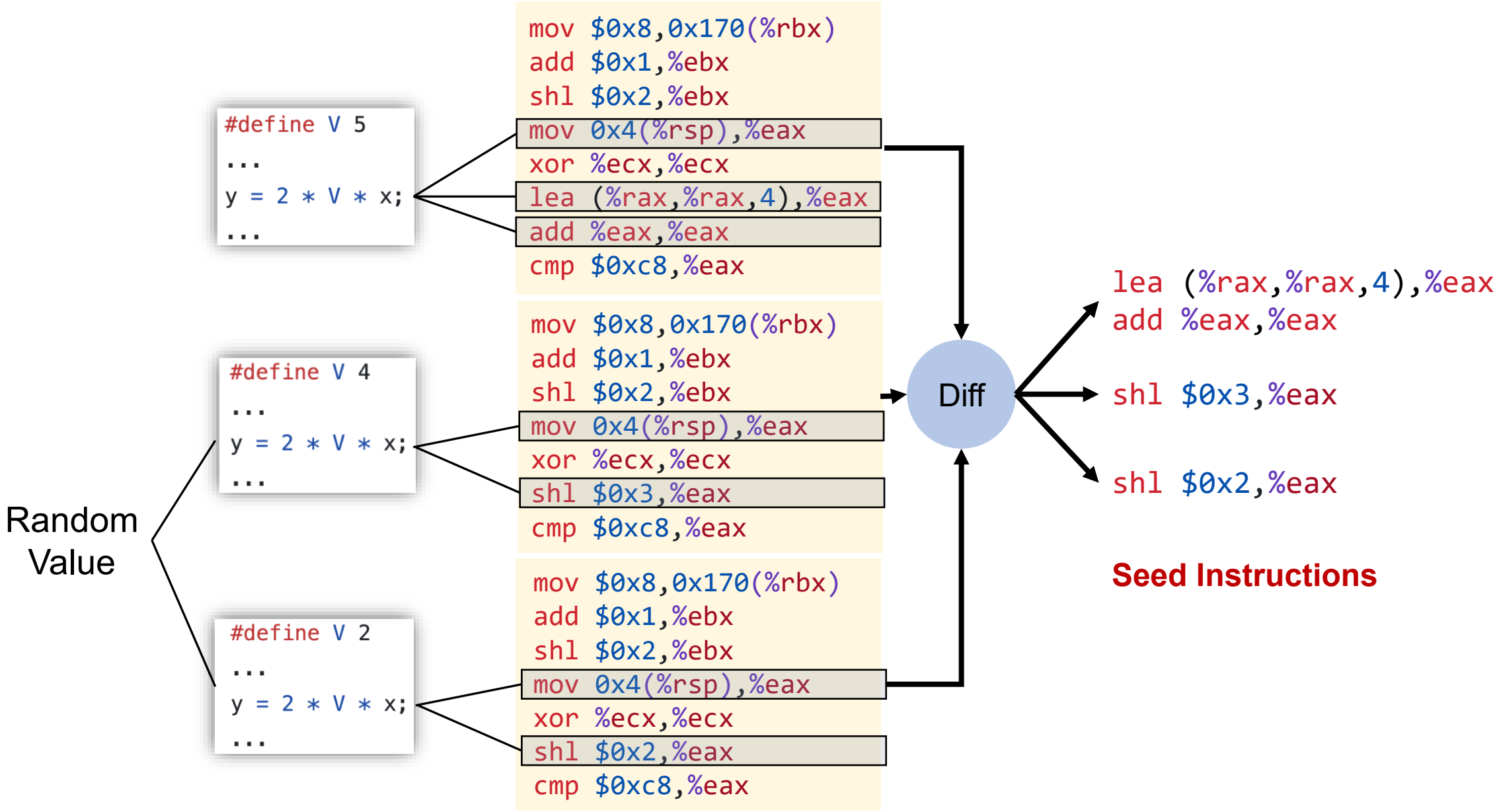


Critical Span (CS)

Scoped Indirect Execution (SIE)



Locate Perf-const and Construct CS



Locate Perf-const and Construct CS

Basic block starts

Binary	Symbolic State
`mov $0x8,0x170(%rbx)`	`Mem1=0x8`
`add $0x1,%ebx`	`Mem1=0x8,ebx=ebx+1`
`shl $0x2,%ebx`	`Mem1=0x8,ebx=4*(ebx+1)`
`mov 0x4(%rsp),%eax`	`Mem1=0x8,ebx=4*(ebx+1),eax=Mem2`
`xor %ecx,%ecx`	`Mem1=0x8,ebx=4*(ebx+1),eax=Mem2,ecx^=ecx`
Seed `lea (%rax,%rax,4),%eax`	`Mem1=0x8,ebx=4*(ebx+1),eax=5*Mem2,ecx^=ecx`
`add %eax,%eax`	`Mem1=0x8,ebx=4*(ebx+1),eax=10*Mem2,ecx^=ecx`
`cmp $0xc8,%eax`	`Mem1=0x8,ebx=4*(ebx+1),eax=10*Mem2,ecx^=ecx`

Basic block ends

*Basic block: single-entry, single-exit span

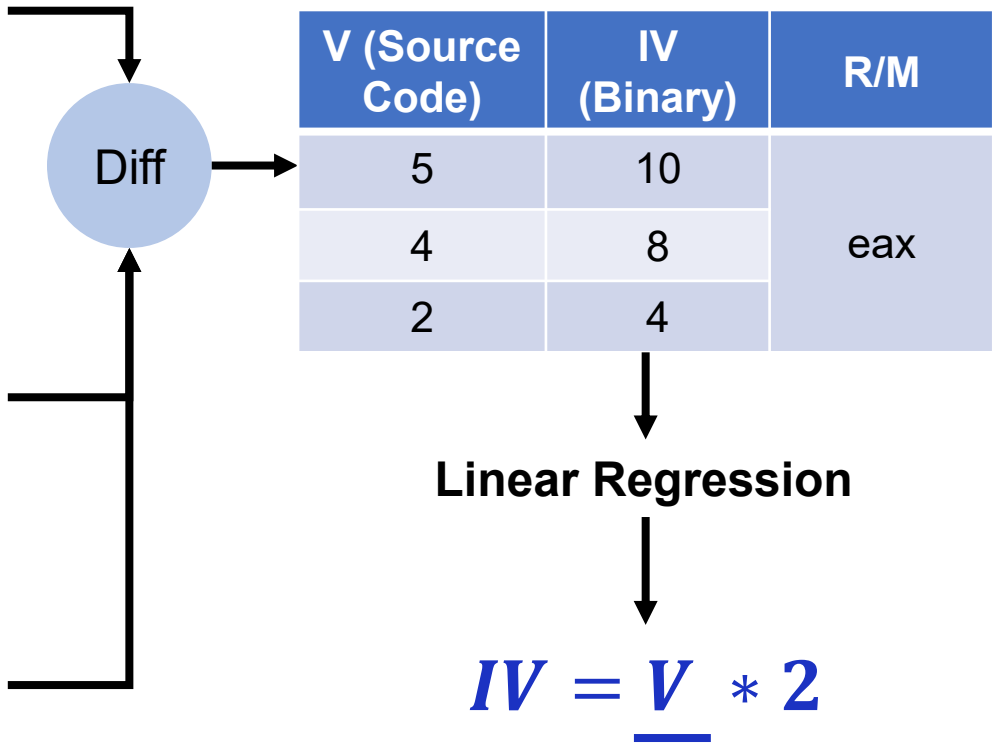
Locate Perf-const and Construct CS

Symbolic States

```
……
Mem1=0x8,ebx=4*(ebx+1),eax=Mem2
Mem1=0x8,ebx=4*(ebx+1),eax=Mem2,ecx^=ecx
Mem1=0x8,ebx=4*(ebx+1),eax=5*Mem2,ecx^=ecx
Mem1=0x8,ebx=4*(ebx+1),eax=10*Mem2,ecx^=ecx

……
Mem1=0x8,ebx=4*(ebx+1),eax=Mem2
Mem1=0x8,ebx=4*(ebx+1),eax=Mem2,ecx^=ecx
Mem1=0x8,ebx=4*(ebx+1),eax=8*Mem2,ecx^=ecx

……
Mem1=0x8,ebx=4*(ebx+1),eax=Mem2
Mem1=0x8,ebx=4*(ebx+1),eax=Mem2
Mem1=0x8,ebx=4*(ebx+1),eax=4*Mem2,ecx^=ecx
```



Locate Perf-const and Construct CS

Basic block starts

Binary	Symbolic State
`mov $0x8,0x170(%rbx)`	`Mem1=0x8`
`add $0x1,%ebx`	`Mem1=0x8,ebx=ebx+1`
`shl $0x2,%ebx`	`Mem1=0x8,ebx=4*(ebx+1)`
`mov 0x4(%rsp),%eax`	`Mem1=0x8,ebx=4*(ebx+1),eax=Mem2`
`xor %ecx,%ecx`	`Mem1=0x8,ebx=4*(ebx+1),eax=Mem2,ecx^=ecx`
`lea (%rax,%rax,4),%eax`	`Mem1=0x8,ebx=4*(ebx+1),eax=5*Mem2,ecx^=ecx`
`add %eax,%eax`	`Mem1=0x8,ebx=4*(ebx+1),eax=10*Mem2,ecx^=ecx`
`cmp $0xc8,%eax`	`Mem1=0x8,ebx=4*(ebx+1),eax=10*Mem2,ecx^=ecx`

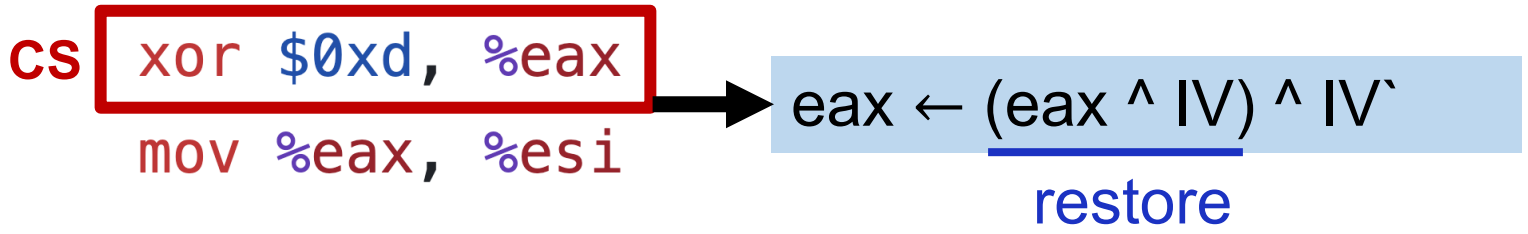
CS

Basic block ends

Synthesizing Indirections

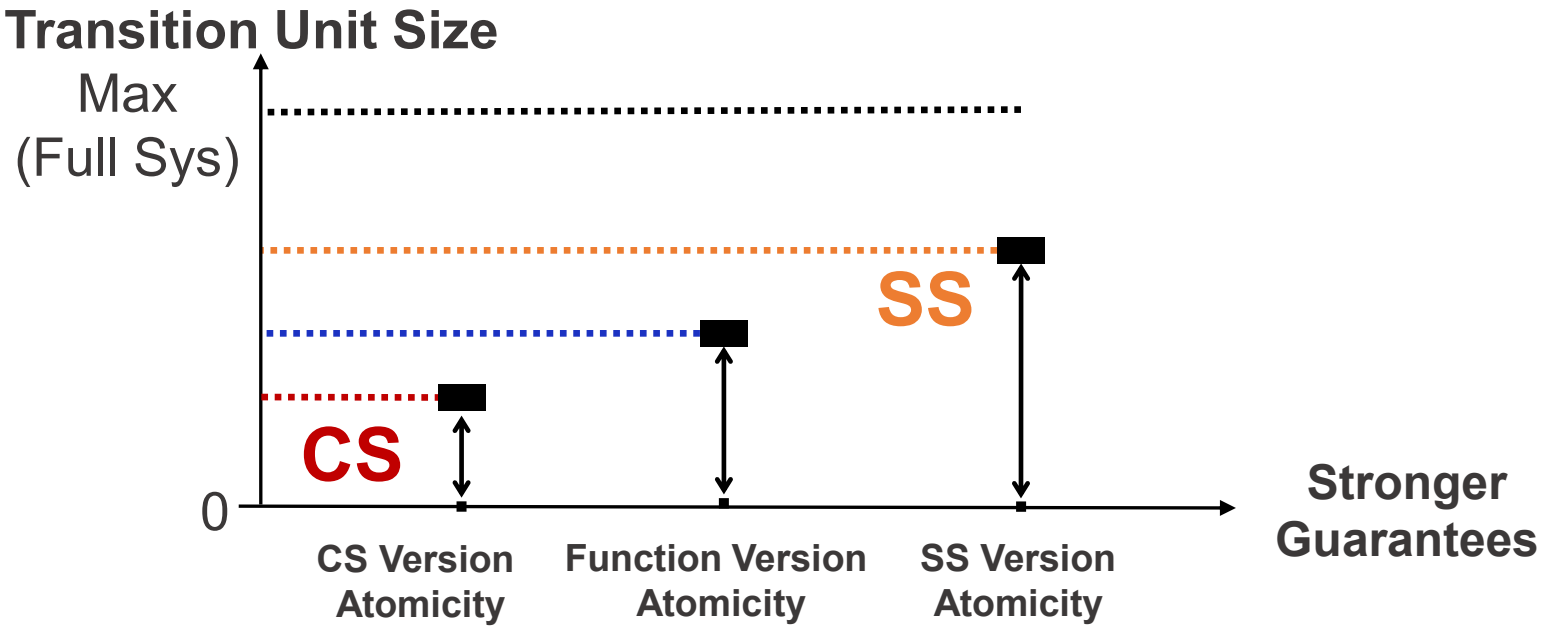
Recovered Expression | **Binary** | **Synthesized Instructions**

eax ← eax ^ IV



Safe Transition

- SIE extends CS to **Safety Span (SS)** using *dependency analysis* and *forward slicing* to encapsulate the propagation of perf-const.
- SIE enables **Side Effect Safety** using **SS-level version atomicity**.



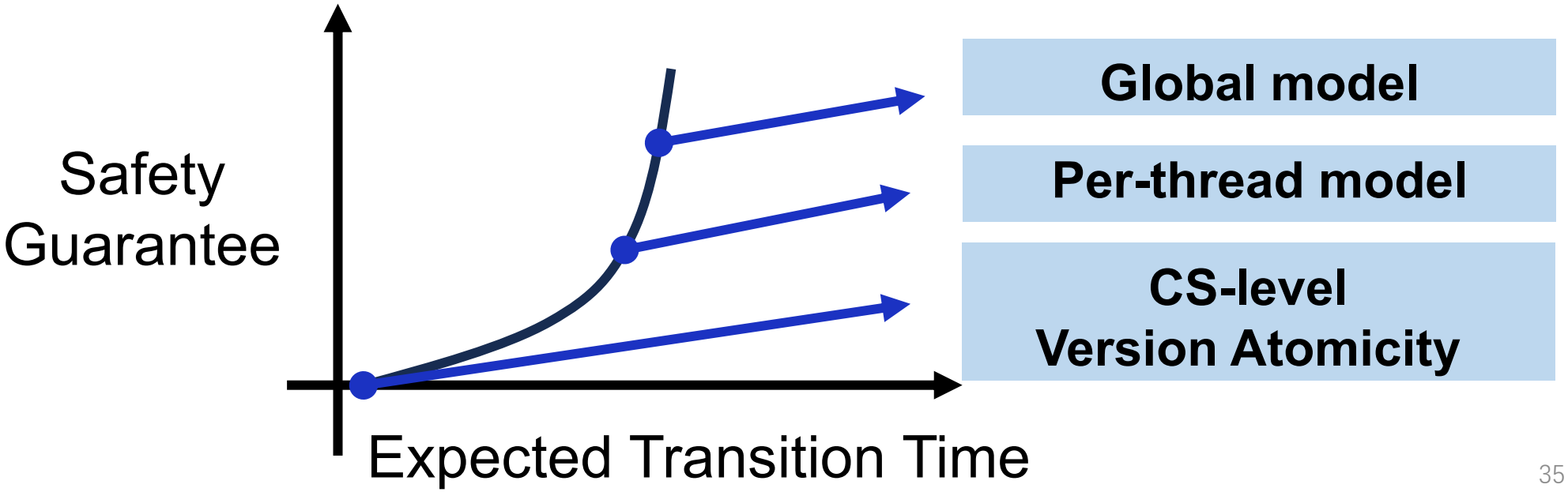
Safe Transition

- **Safe transition timing**

$$\forall t \in T, PC(t) \notin SS \Rightarrow \text{transition}(v \rightarrow v') \text{ is safe}$$

where **T** is thread set, **PC(t)** is the program counter of thread **t**.

- **Three models**



Practical Integration with Linux Kernel

- SIE is built on ***kernel-native*** techniques without kernel modification
- **Kprobe** (Kernel Probe)
  - Support probing to almost any location in the kernel
  - Kernel function + offset
- **eBPF** (**e**xtend **B**erkeley **P**acket **F**ilter)
  - Safe kernel programming
  - Kernel states monitoring
  - Constraint kernel memory modification with **BPF kfuncs**

Programmable Policy Plane

- **Xkernel User APIs**

Definition of context and user probe

```
typedef const struct xk_ctx * xk_handle_t;
#define XK_TUNE(unique_name, perfconst_id, args...)
```

```
long xk_set(xk_handle_t xk_ctx, u64 val);
bool xk_transition_done(xk_handle_t xk_ctx);
```

The set and transition API

Programmable Policy Plane

See more examples in paper

```
XK_TUNE(blk_add_rq_to_plug, “block/blkmq.c:L1377:32:0") {
// 1. Safety Guard (Mandatory)
if (!xk_transition_done(xk_ctx)) return 0;
// 2. User policy logic
struct request *rq = PT_REGS_PARM2(ctx);
char name[32];
if (bpf_probe_read_kernel(name,
    sizeof(name), &rq->disk_name) < 0)
return 0;
if (bpf_strncmp(name, 3, "nvme") == 0 &&
    is_random_access())
    xk_set(xk_ctx, 1);
else if (bpf_strncmp(name, 3, "sdb") == 0 &&
    is_sequential_access())
    xk_set(xk_ctx, 128);
return 0;
} /* BLK_MAX_REQUEST_COUNT.bpf.c */
```

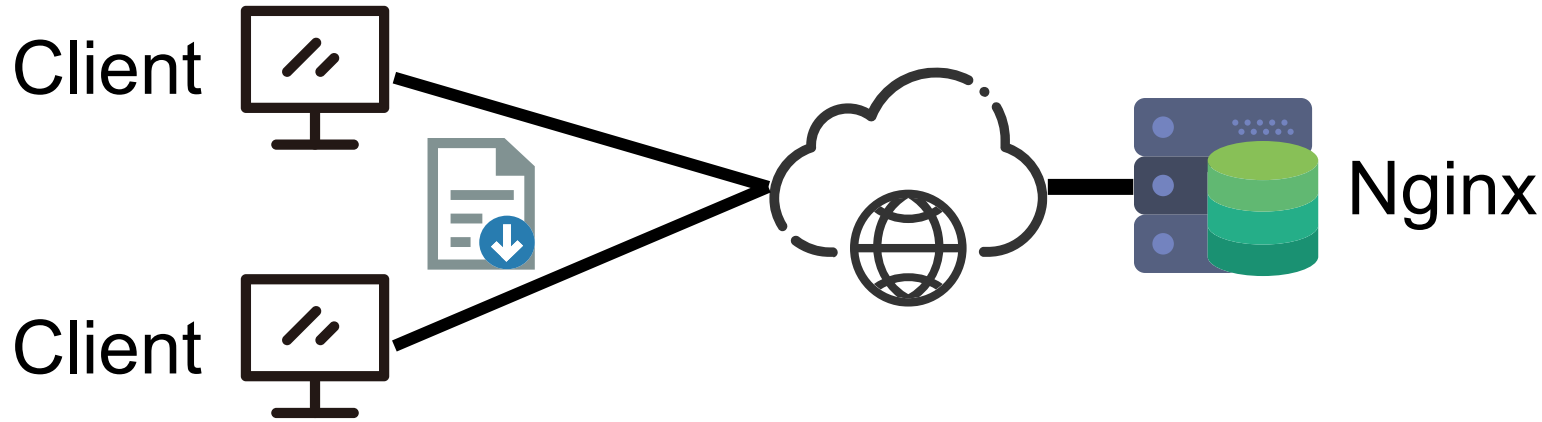
eBPF programming

Application hint with eBPF maps (e.g., tid/pid)

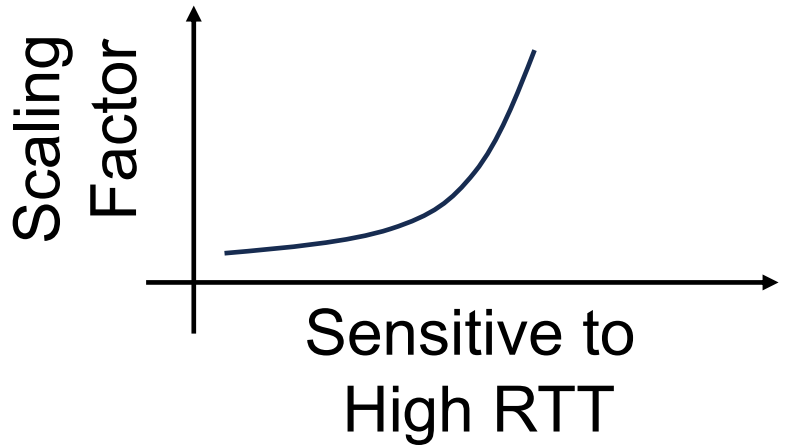
Outline

- Introduction
- Challenges
- Design
- **Evaluation**

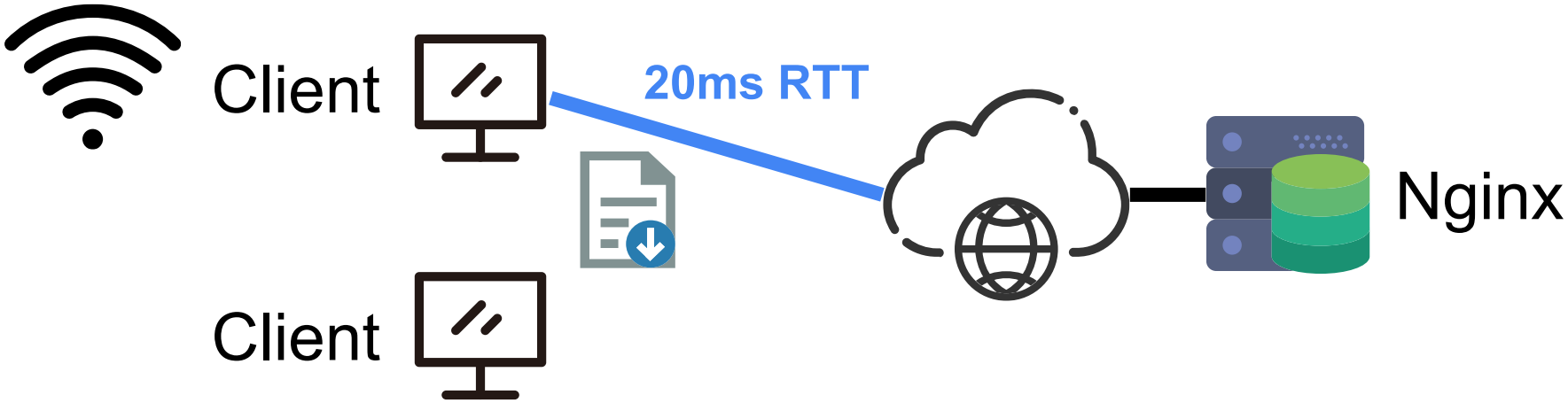
You can get the best of both worlds with Xkernel

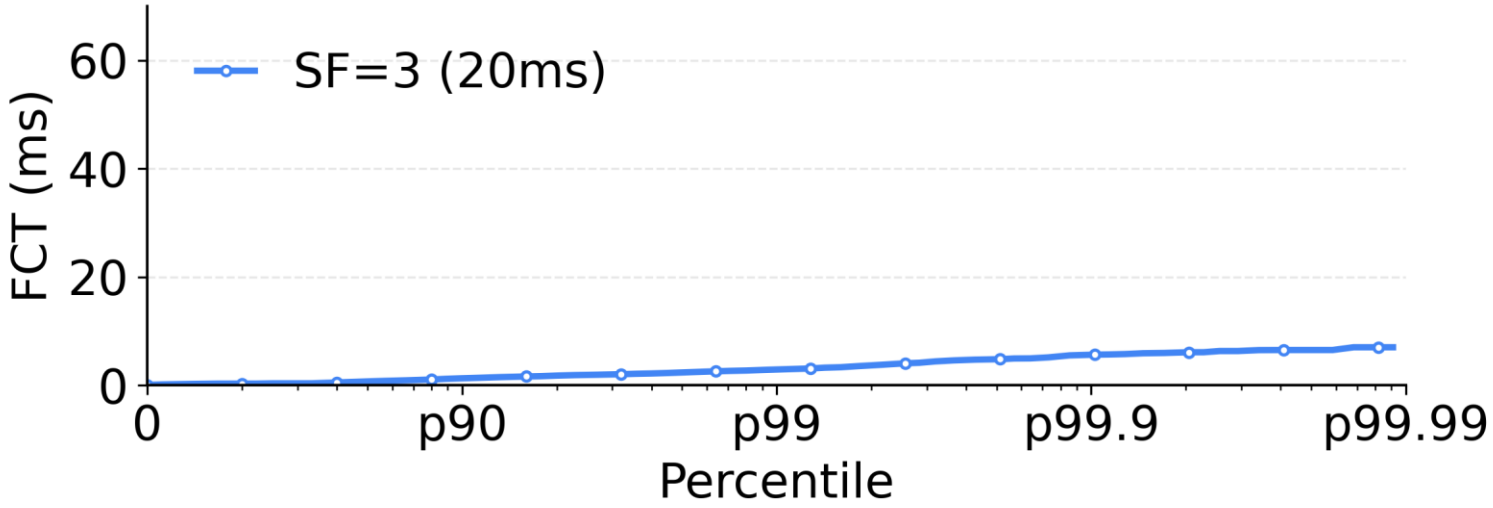


Scaling Factor (SF) in TCP Cubic: `ca->delay_min >> 3`

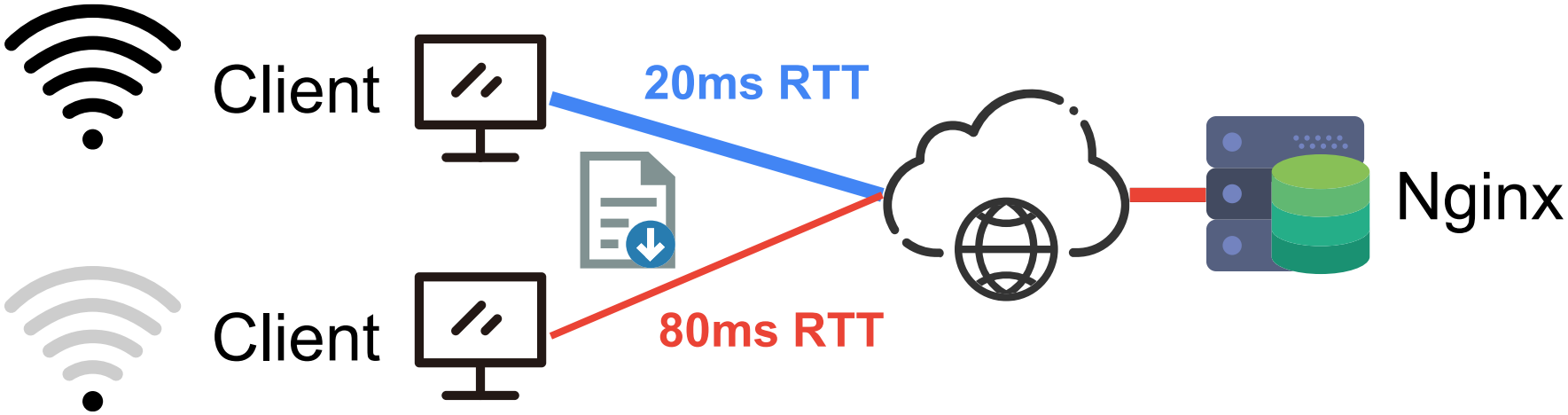


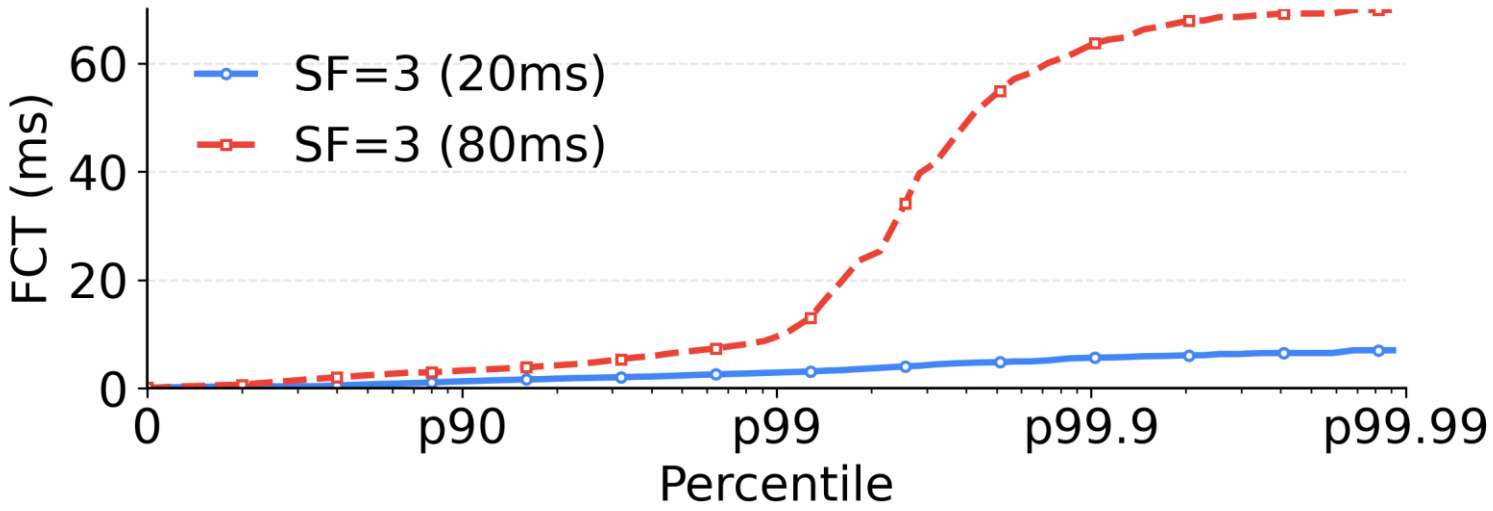
You can get the best of both worlds with Xkernel



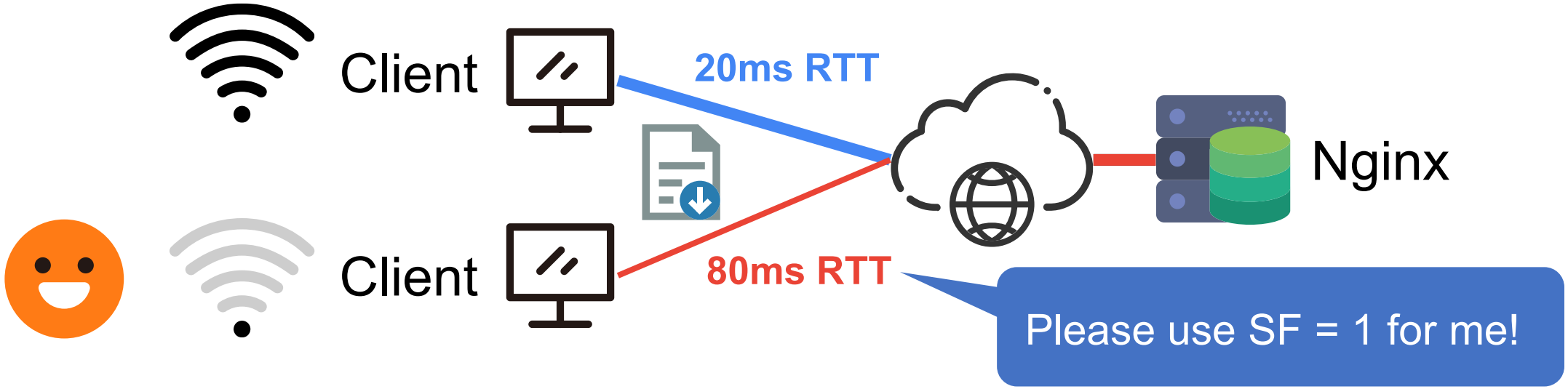


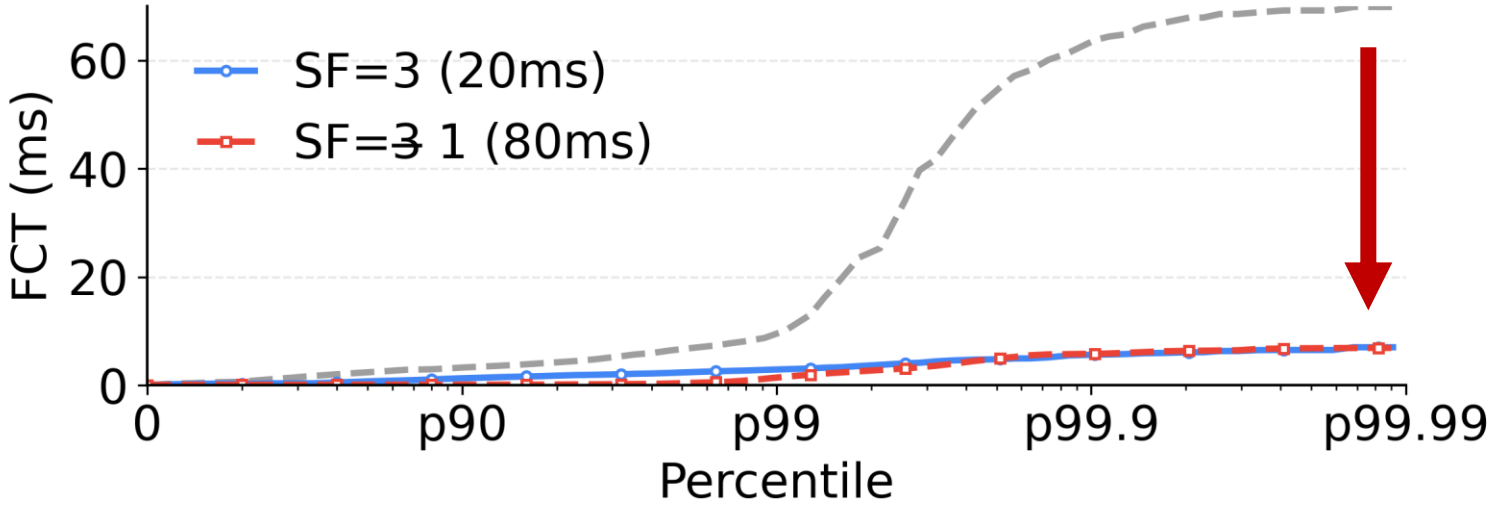
You can get the best of both worlds with Xkernel



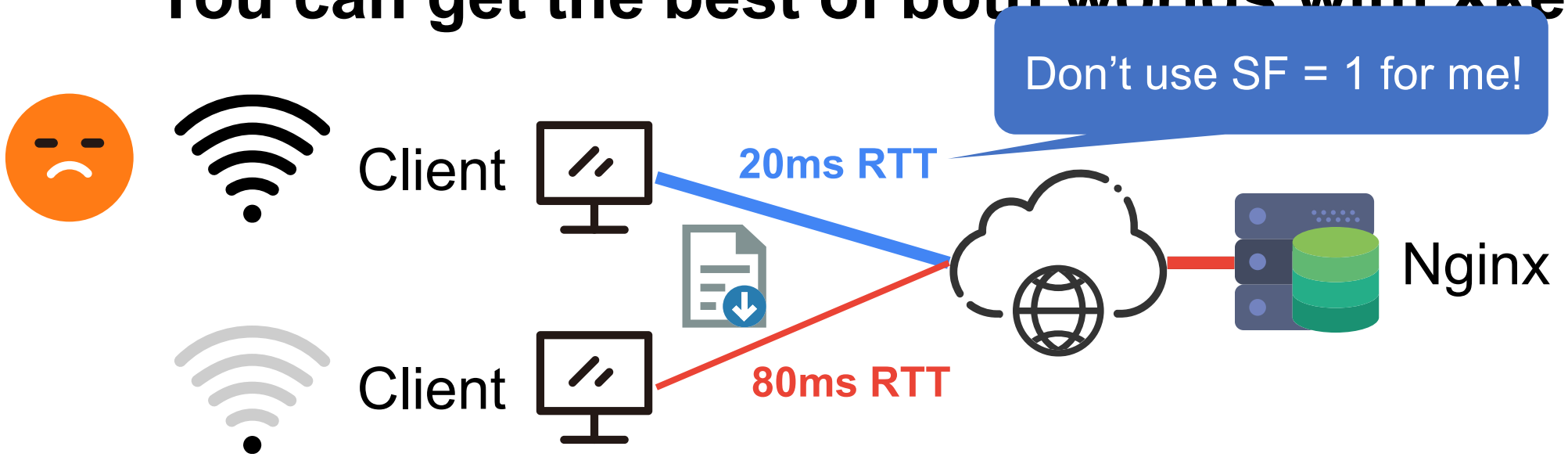


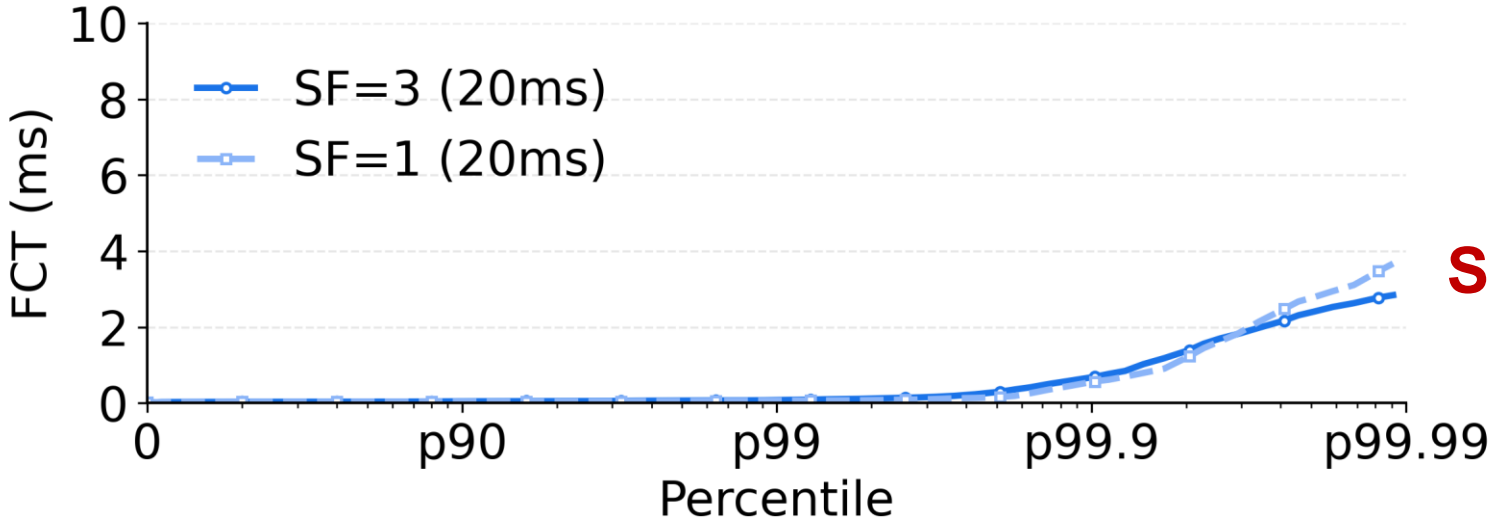
You can get the best of both worlds with Xkernel





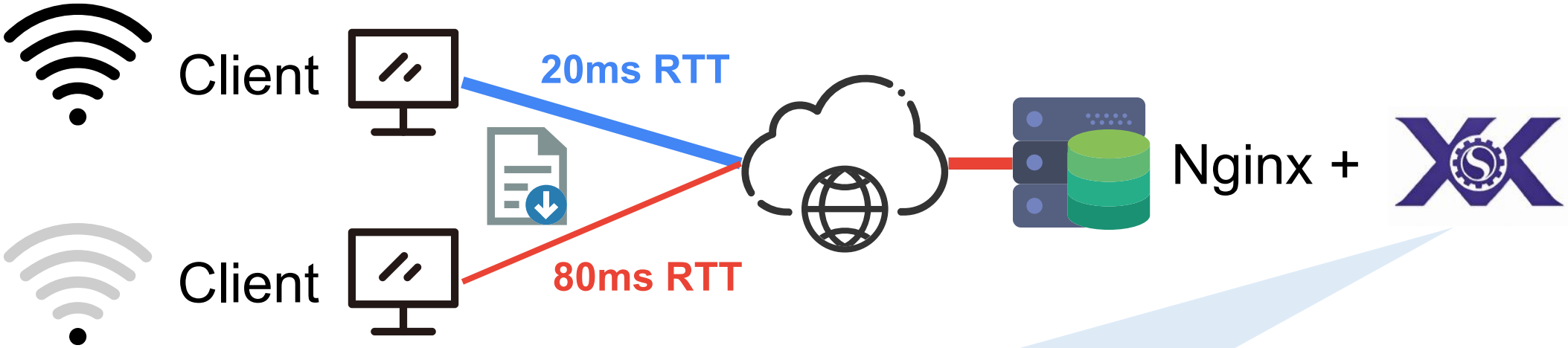
You can get the best of both worlds with Xkernel





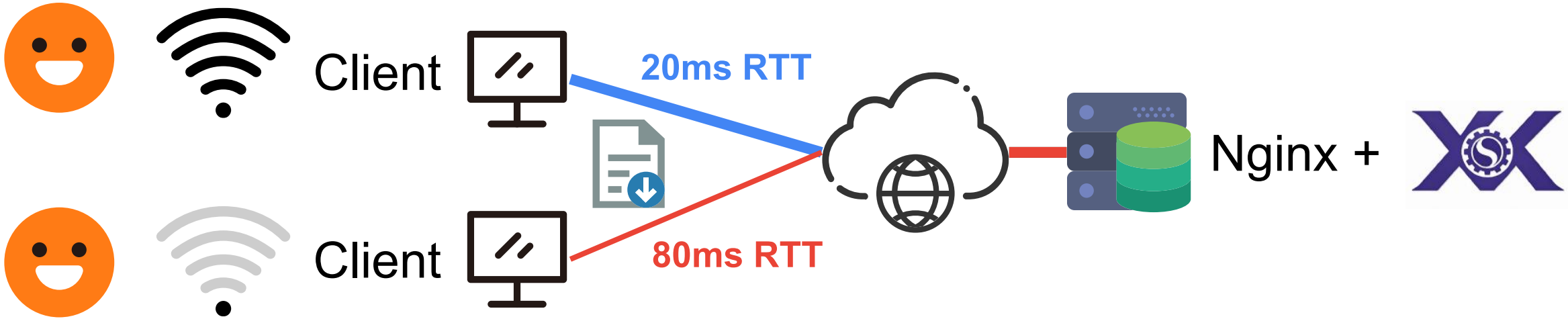
Slightly worse

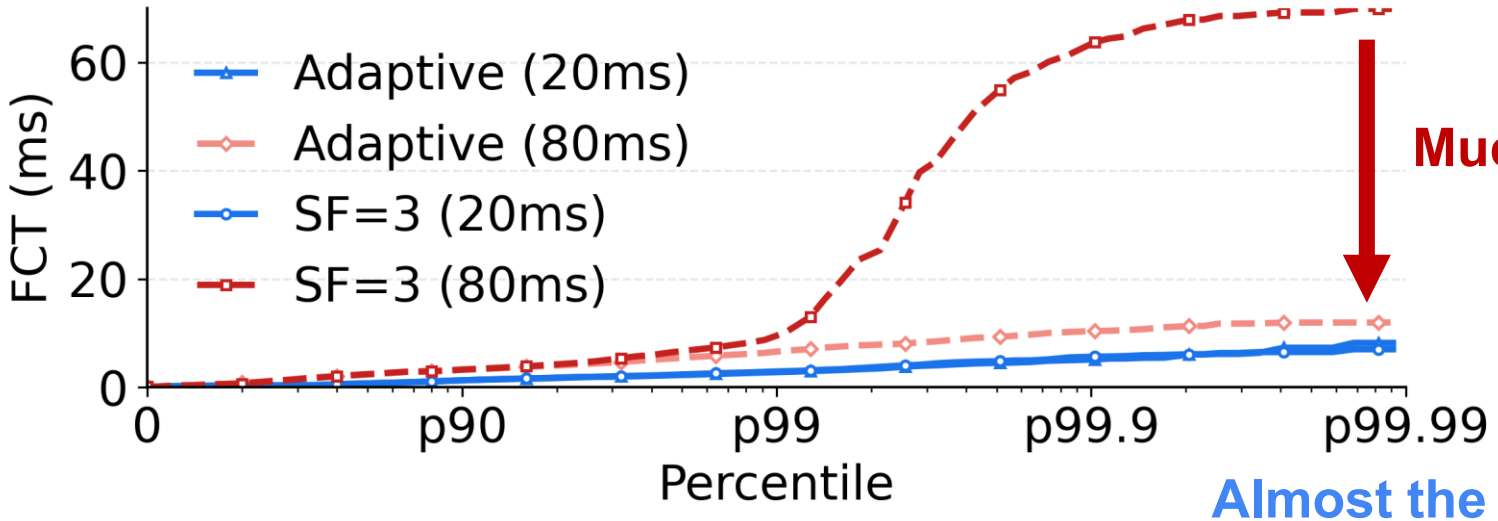
You can get the best of both worlds with Xkernel




```
// 2. User policy logic
// - Obtain the current network flow's RTT
struct sock *sk = (struct sock *)PT_REGS_PARM1(ctx);
struct bictcp *ca = inet_csk_ca(sk);
u32 cur_rtt = BPF_CORE_READ(ca, curr_rtt);
// - Set the value to 1 only when RTT > 80ms
if (cur_rtt >= 80000) xk_set(xk_ctx, 1);
return 0;
```

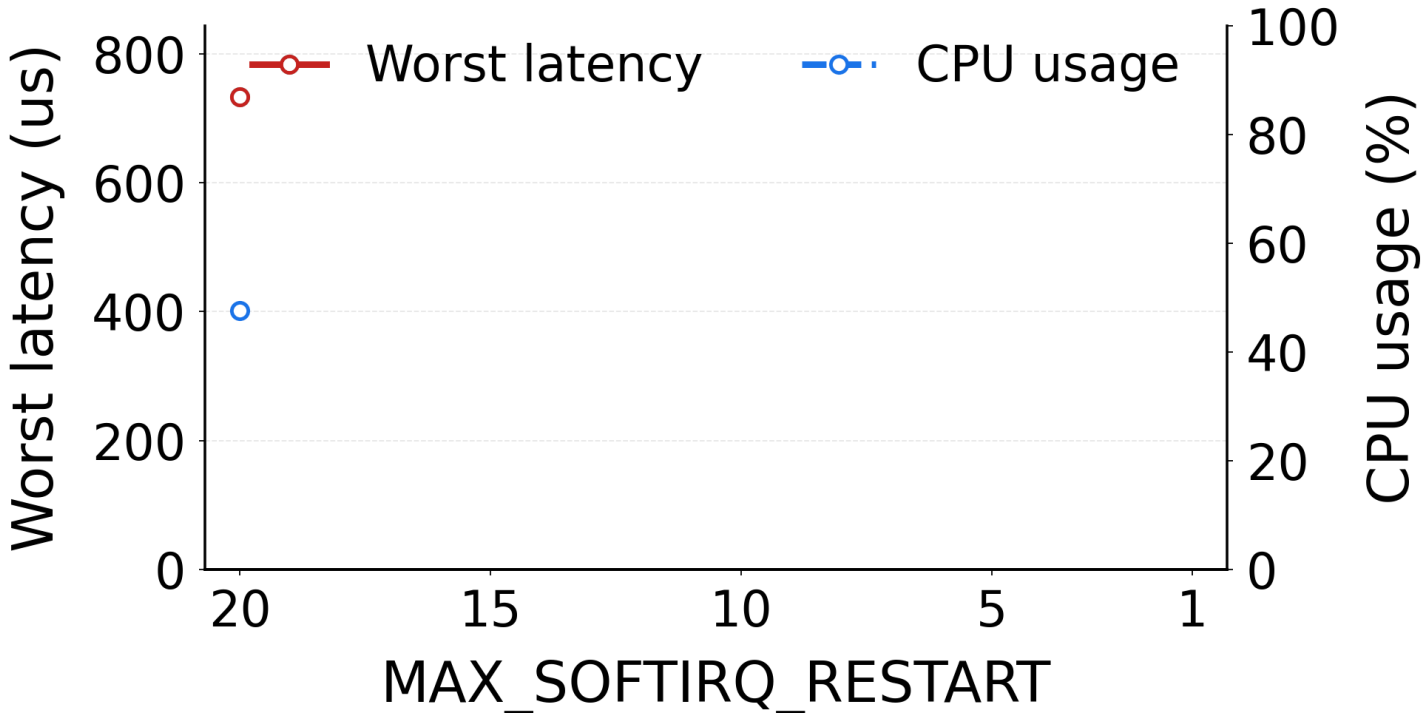
You can get the best of both worlds with Xkernel





Xkernel Benefits Many OS Subsystems

- **CPU scheduling subsystem**
  - `MAX_SOFTIRQ_RESTART`
  - Reduce the worst latency at the cost of CPU usage

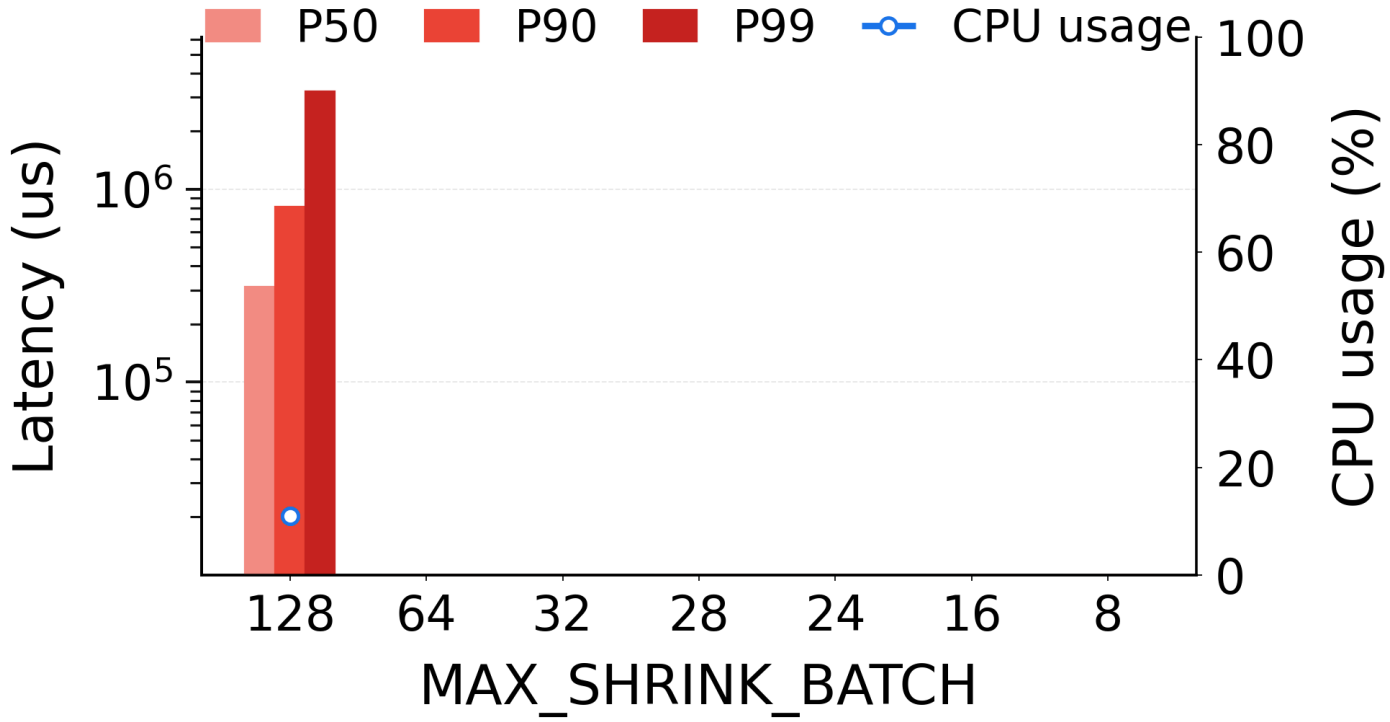


Xkernel Benefits Many OS Subsystems

- **Memory subsystem**

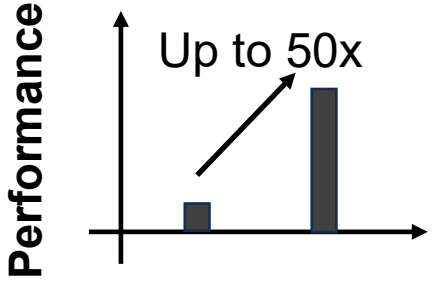
See more examples in paper

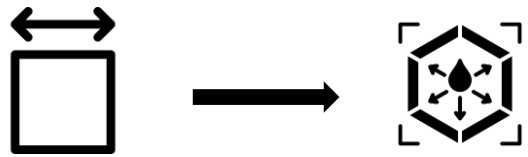
  - `SHRINK_BATCH`
  - Reduce the reclaim batch size to increase the data reuse opportunity



Conclusion

- **Principled performance tunability**
  - Flexible, safe, and fast in-situ OS tuning
- **Scoped Indirect Execution (SIE)**
  - Symbolic Execution, Critical Span, Safety Span
- **General, efficient, and low overhead**
  - Tested on a dataset of 140 perf-consts (comparable to sysctl scale)
  - Policy update time: < 542 milliseconds
  - Low overhead in common-path





Substantially larger policy space



New path of system performance
Many new research questions to answer

Thank you!

Xkernel is open-source available



github.com/xkernel-org/Xkernel

zhongjie@microsoft.com
jingliu3@microsoft.com