Efficient Scheduler Live Update for Linux Kernel with Modularization

2023/07/17　システム情報専攻
江　松穎

Information

Authors

- Alibaba Group

  Teng Ma　　Shanpei Chen　　Yihao Wu

  Erwei Deng　　Zhuo Song

- Shanghai Jiao Tong University

  Quan Chen　　Minyi Guo

Purposed Conference

- ASPLOS 2023 —

  28th ACM International Conference on Architectural Support for Programming languages and Operating Systems

Outline

1. Background
2. Prior Researches
3. System Design
4. Evaluation
5. Summary

Background – The Linux Scheduler

The scheduler is one of the most critical and complicated subsystem in the OS.

- Depends on the synchronization primitives
  → E.g., RCU (Read-Copy-Write) / Interrupts
- Interacts with other subsystems
  → E.g., Memory Management
- Sizeable
  → >27 KLOC, also >60 Files

Direct impacts many metrices —
Scheduling latency, CPU Utilization, etc

Background — Problem of Schedulers

Workload Has Different Features ➡ Difficult to Develop One-Fit-All Scheduler

CFS Scheduler, default for current Linux kernel,

→ Uses ~7.6% of CPU cycles
when ~1000 container co-run on single node in cloud scenario

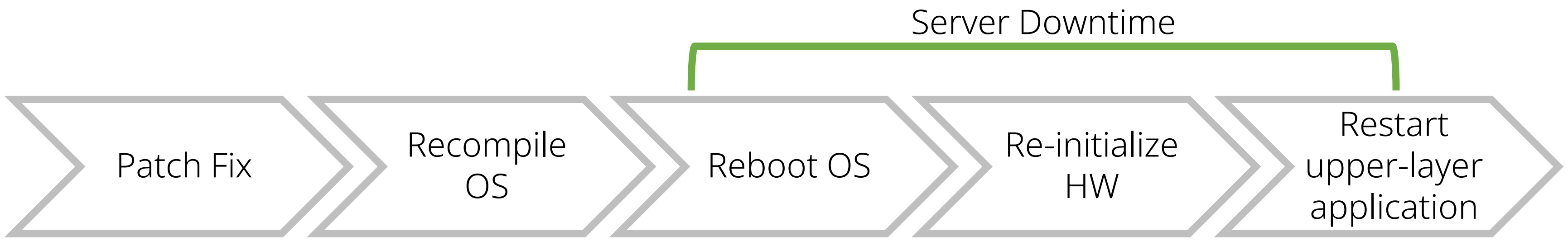
Using Workload-Specific Schedulers Helps:

- Tableau Scheduler
  → Web service applications, improved throughput by 1.6x
- Calandan Scheduler
  → Network request, improved tail-latency by 11,000x

However, updating scheduler is costly

Background — Replacing Schedulers

Traditrional Straight Forward Update Steps



VM-Migration technique enables live-update & speeds things up, but…

- Migration (with network) time grows as thread number increases
- Having >10,000 thread on recent server now is normal

  → **Minutes** of downtime while migration

**Minute-level downtime is unacceptable,
aim for 10ms or lower**

Prior Researches

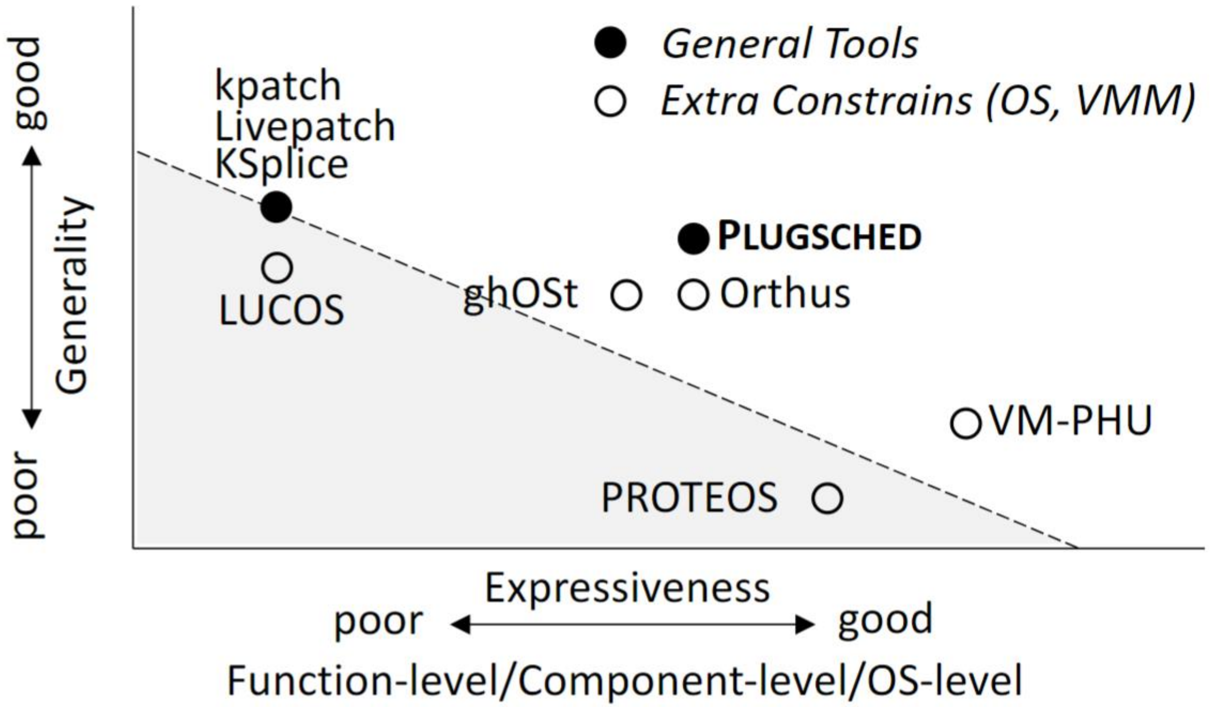


Figure 1: Comparisons between potential solutions.

PROTEOS

- Only for microkernel

ghOSt

- Scheduler update in user space
- Server code modification required
- Only for scheduler class

VM-PHU

- Only for guest kernel in VM

PLUGSCHED

System proposed by paper

System Design — Design Goals

Adequate expressiveness

- Support component-level live update
- Data state should also be migratable

High generality

- Can be applied to Linux servers without constraints
- No need to change kernel code

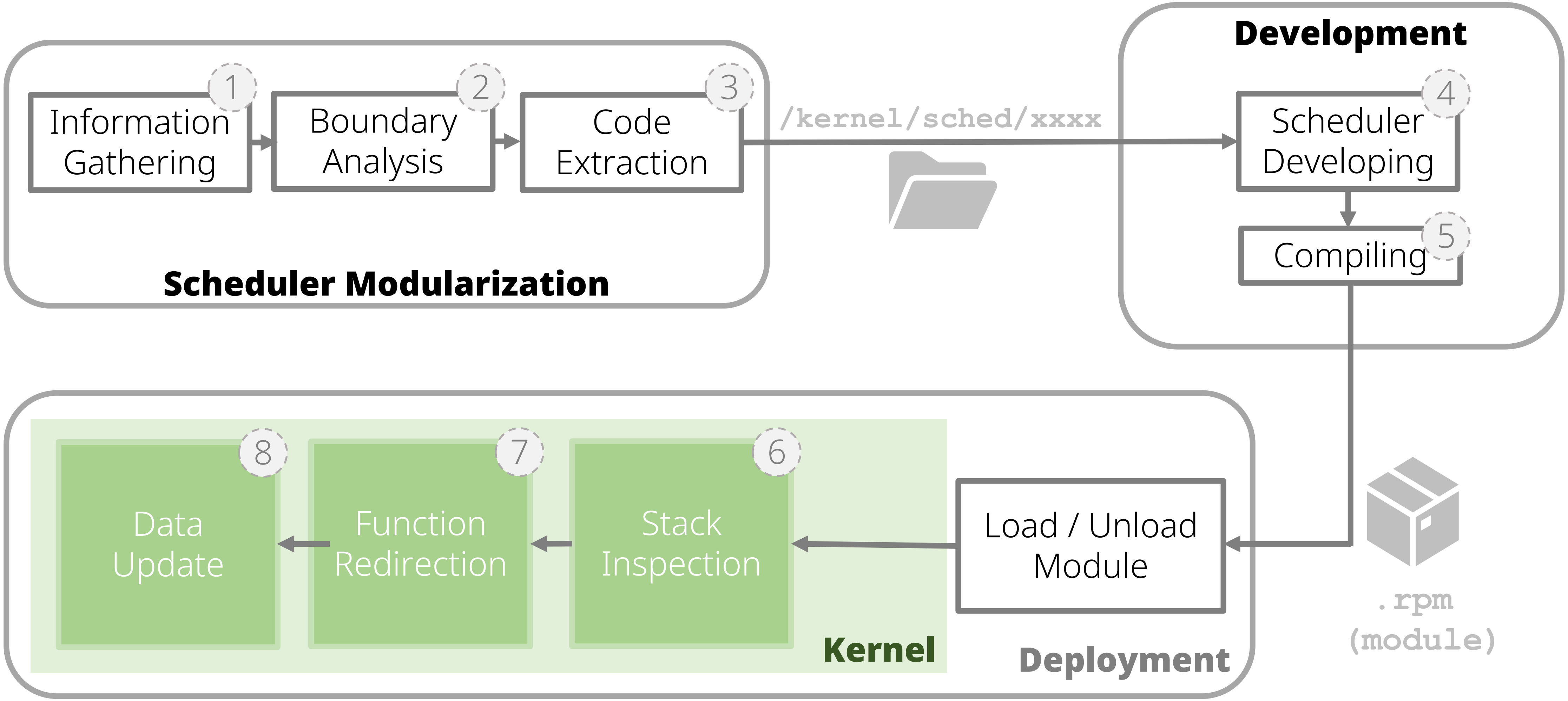
Achieve both short downtime and safety

- Minimum downtime
- No damage to system — clear boundary and atomic update

Easy-to-use

- Provide common live-update tools
- Upgrade / Rollback should be done by install/uninstall scheduler module

System Design — Overview



System Design — Scheduler Modularization①

Scheduler related files are under `/kernel/sched` but...

- Not all function / data are relevant
- We cannot move these out since <u>we cannot modify kernel code</u>

⇨ **Function Based Boundary**

Classify function / data to different classes

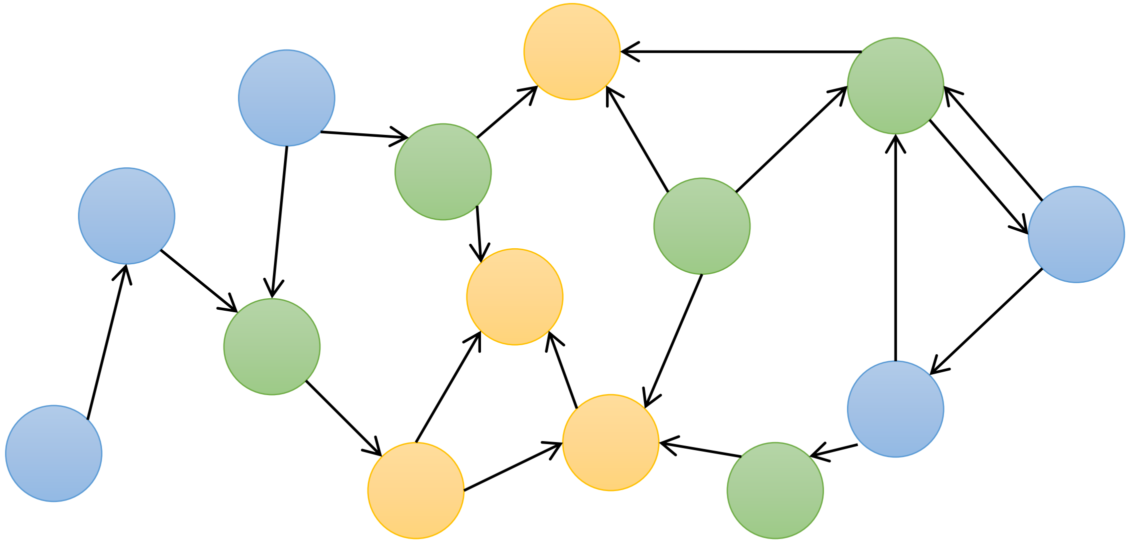
Data Type	Used by
$D_{Internal}$	Scheduler only
$D_{external}$	All over the kernel

Function Type	Caller of	Callee of
$F_{Internal}$	$F_{Internal}$	$F_{Internal}$ $F_{Interface}$
$F_{Interface}$	$F_{Internal}$	$F_{external}$
$F_{external}$	$F_{Interface}$ $F_{external}$	$F_{external}$

System Design — Scheduler Modularization②

Boundary Analysis

- $\boldsymbol{F_{Internal}}$ as input provided by developers
- Kernel Call Graph ($\boldsymbol{G}$) provided by GCC Plugin



Algorithm 1: Boundary Analysis.

(1) Initialization:
Load user-defined configurations ($F_{interface}$) and compiler generated information (F_{mod}, G)
(2) Inflect:
$F' \leftarrow F_{mod} - F_{interface}$
$F'_{external} \leftarrow \varnothing$
while *True* **do** /* Coverage */
  $F^* \leftarrow \varnothing$
  foreach $e_{f_i, f_j} \in G$ **do**
    if $f_i \notin F' \wedge f_i \notin F_{interface} \wedge f_j \in F'$ **then**
      $F^* \leftarrow F^* \cup \{f_j\}$
    end
  end
  if $F^* = \varnothing$ **then**
    break
  end
  $F'_{external} \leftarrow F'_{external} \cup F^*, F' \leftarrow F' - F^*$
end
(3) Save Results:
$F_{internal} \leftarrow F', F_{external} \leftarrow F'_{external}$

System Design — Scheduler Modularization③

Scheduler code extracted into `/kernel/sched/...` for development

- Transform external function/data into declarations
  - → $\boldsymbol{F_{external}}$ — Delete function body and add semi-colon "`;`" behind
  - → $\boldsymbol{D_{external}}$ — Referencing `VarDecl::str_decl` from GCC Plugin

Classification result for Linux 4.19

Type	$F_{Internal}$	$F_{Interface}$	$F_{external}$
Count	786	94	96

System Design — Stack Inspection

Uses technique similar to other approaches (kpatch), just better

Compare call stack with old functions to make sure that, after update, threads will not execute text from old scheduler.

1. `stop_machine()` to halt system
2. Parallel Checking: kpatch uses only one CPU Core, PLUGSCHED uses the whole CPU
3. When traversing function list, use binary search

Table 1: Different potential solutions for scheduler update. (*: This approach does not exist for kernel subsystem.)

Approach	Stack Inspection	Expressive	w/o Kernel Changes	w/o Extra Overhead	Data Update	Granularity
ebpf [3]	-	👎	✘	✔	✔	Function
kpatch [8]	$O(T * D * N)$	👎	✘	✔	✔	Function
gHost [26]	-	👍	✔	✘	✘	Component
kernel module*	-	👍	✔	✔	✔	Component
PLUGSCHED	$O(\frac{T}{C} * D * logN)$	👍	✔	✔	✔	Component

System Design — Function Redirection

Achieved by replacing instructions

- First instruction of old function is replaced by `JMP` instruction, towards the new function

Only exception, `__schedule()`

- Sleeping thread returns to old `__schedule()`
- All scheduling decision is done before `context_switch()`
  → New function only does the upper part, then jump to old `context_switch()`

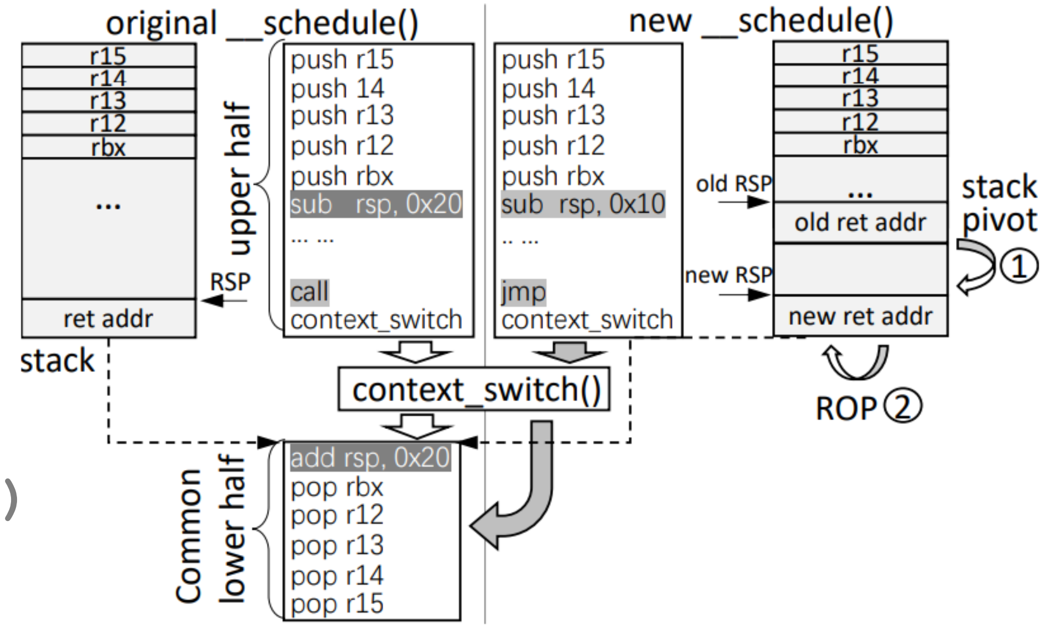


Figure 3: Workflow of handling __schedule()

System Design — Data Update

Many data are "stateful", and should remain the state

- E.g., running task should remain running state
- $D_{External}$ — Developers are not allowed to make change
- $D_{Internal(Private)}$ — New memory allocated in new scheduler
- $D_{Internal(Shared)}$ — Use memory allocated in the old scheduler

Internal data also has differences

- Critical data must be rebuilt from the ground up. E.g., `struct cfs_rq`

Type	Critical	Non-Critical
$D_{External}$	Inherit	
$D_{Internal(Private)}$	Data Rebuild	Re-Initialization
$D_{Internal(Shared)}$	Data Rebuild	Inherit

Evaluation — Experiment Setup

Platforms

- x86_64
- ARM64

Implementation

- Python — ~2,000 LOC
- C — ~6,000 LOC

Linux Kernel

- v4.19.91

CPU

- Intel Xeon(R) Platinum 8163 CPU @ 2.50GHz
- Dual-socket, 48 core, 32MB L3 Cache

Memory

- 192GB

Evaluation — Downtime on Different Cases

3 Types of Update

- Patch —
  Minor changes / fixes
- Feature —
  Add component to current scheduler
- New Scheduler —
  Entire scheduler revamp

Update size does not affect downtime

Table 3: Different cases to use PLUGSCHED. p: patch, f: feature, n: new scheduler.

ID	Type	Commit ID/Feature	Files	LOC	Data
1	p(opt)	aa93cd53bc1b91	1	24	✘
2	p(opt)	11f10e5420f6ce	1	3	✘
3	p(opt)	45da7a2b0af8fa	1	4	✘
4	p(bug)	b9c88f75226838	1	9	✘
5	f	Remove Bandwidth Control	1	1258	✔
6	f	Add Group Identity	7	2592	✔
7	n	Tiny Scheduler	10	3845	✔

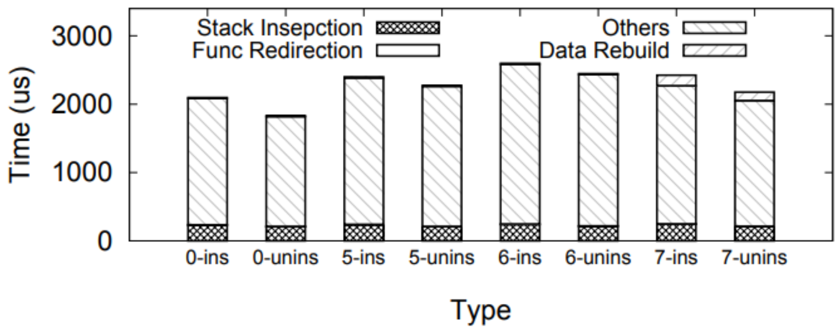


Figure 6: The downtime breakdown when using PLUGSCHED to update the scheduler with different cases.

Evaluation — General Downtime Breakdown

- On test environment with no workload no parallel optimization, **downtime ~2ms for both install and uninstall**
- On real environment with workloads with parallel optimization, almost all **downtime is less then 1ms**
- **98% time is for initialization**, causes no effect on downtime

Table 4: Breakdown of total time in PLUGSCHED. (*: without parallel optimization)

Phase	Upgrade	Rollback
Total Time	141*ms*	93*ms*
Initialization	137*ms*	85*ms*
Down Time	2309μs	2160μs
*Stack Inspection**	1585μs	1054μs
Stack Inspection	224μs	201μs
Function Redirection	7μs	6μs
*Data Rebuild**	135μs	112μs
Data Rebuild	51μs	51μs
Other	2027μs	1902μs

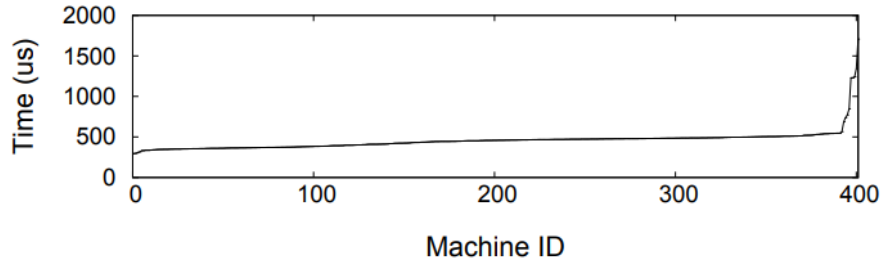


Figure 9: The downtime distribution due to the scheduler update of 400 servers in our production cloud.

Evaluation — Scalability

- Speed goes up until ~24 cores, **but 6 core is sufficient**
- Suboptimal scalability due to
  1. Workloads are not perfectly balanced
  2. Starting SI (Stack Inspection) costs milliseconds, negates the tiny improvement for more core

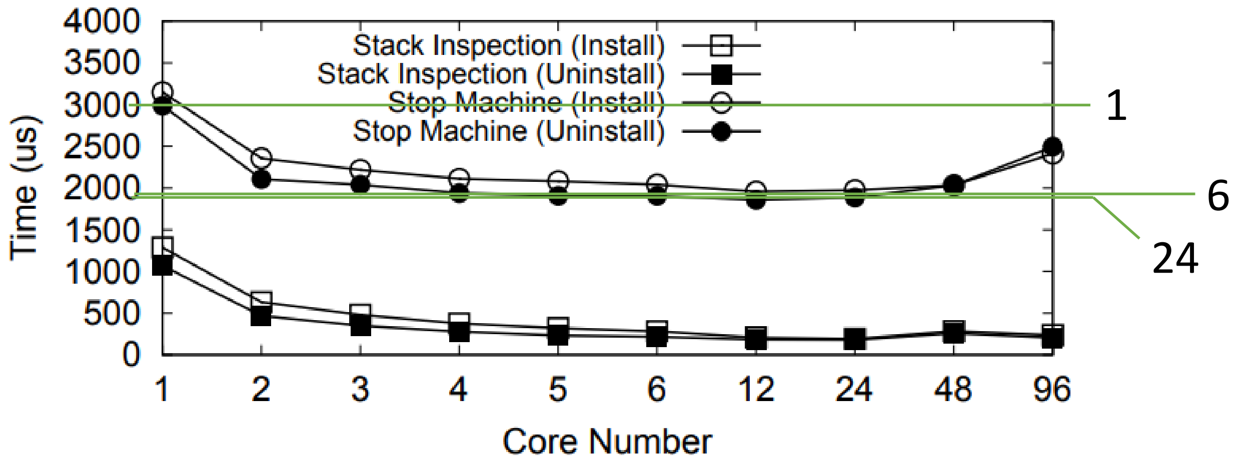


Figure 10: The scalability of PLUGSCHED.

Evaluation — Parallel SI(Stack Inspection) / DR (Data Rebuild)

- Test with <u>no parallel optimization / parallel using 6 cores</u>
- **SI** — **Linear time** for both scenario
- **DR** — With parallel optimization, **constant time** while running thread number grow

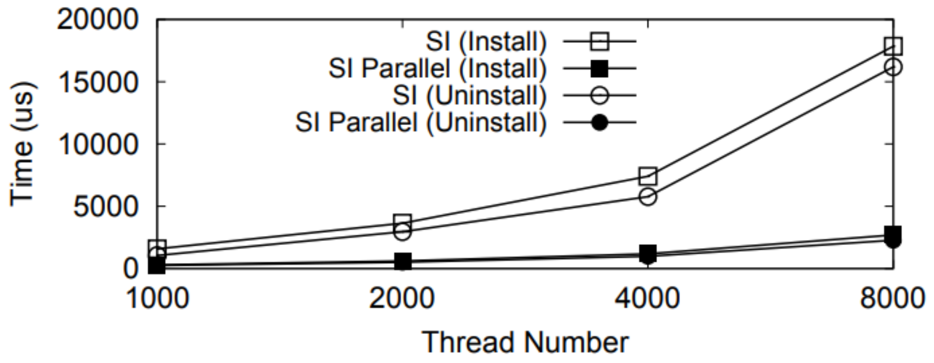


Figure 11: The time of stack inspection (SI) with different number of active threads in the previous scheduler.

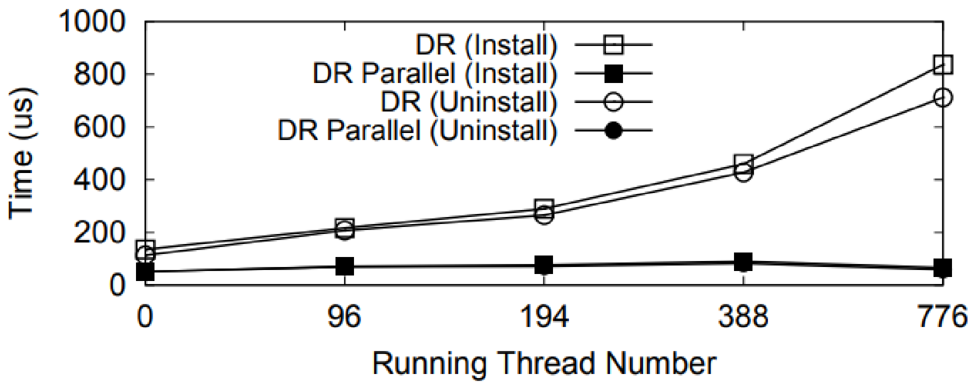


Figure 12: The time of data rebuild (DR) with different number of active threads in the previous scheduler.

Summary

- Sometimes, we need to update the scheduler to fit workloads
- We must do it with minimal downtime

- **PLUGSCHED** enables scheduler live-update with good generality
- With function/data analysis, scheduler code can be extracted to develop update
- Update into module provides easy install
  - Just inspect stack, redirect function, and update data

- Same modularization technique could be used on other components for future work