

LinuxCon Japan 2014  
(2014/5/22)

# Scalability Efforts for Kprobes

or: How I Learned to Stop Worrying and Love a Massive  
Number of Kprobes

**Masami Hiramatsu**

<masami.hiramatsu.pt@hitachi.com>

Linux Technology Research Center  
Yokohama Research Lab. Hitachi Ltd.,

Yokohama Research Lab.  
Linux Technology Center



- Masami Hiramatsu
  - A researcher, working for Hitachi
    - Researching many RAS features
  - A linux kprobes-related maintainer
    - Ftrace dynamic kernel event (a.k.a. kprobe-tracer)
    - Perf probe (a tool to set up the dynamic events)
    - X86 instruction decoder (in kernel)

## Background Story

- Kprobes Blacklist Improvement

- Testing the Blacklist

- Qualitative versus Quantitative

## Scalability Efforts

- Enlarge the Hash Table

- Cache the Hash List

- Reduce Redundancy

## Conclusion and Discussion

## Background Story

Kprobes Blacklist Improvement

Testing the Blacklist

Qualitative versus Quantitative

## Scalability Efforts

Enlarge the Hash Table

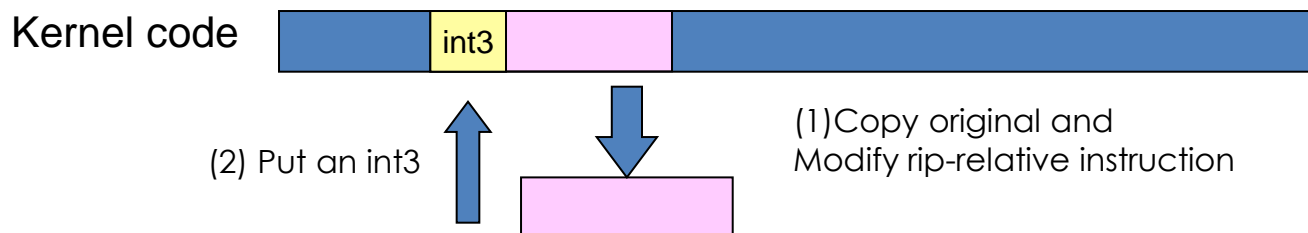
Cache the Hash List

Reduce Redundancy

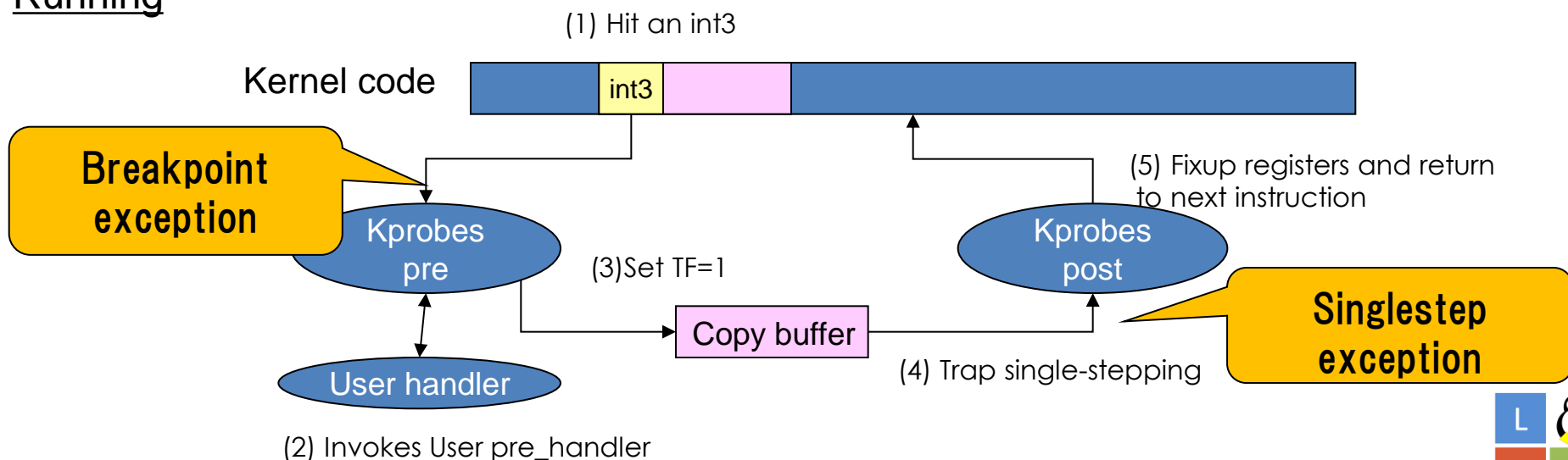
## Conclusion and Discussion

- Kprobes uses a breakpoint and a singlestep on copied code

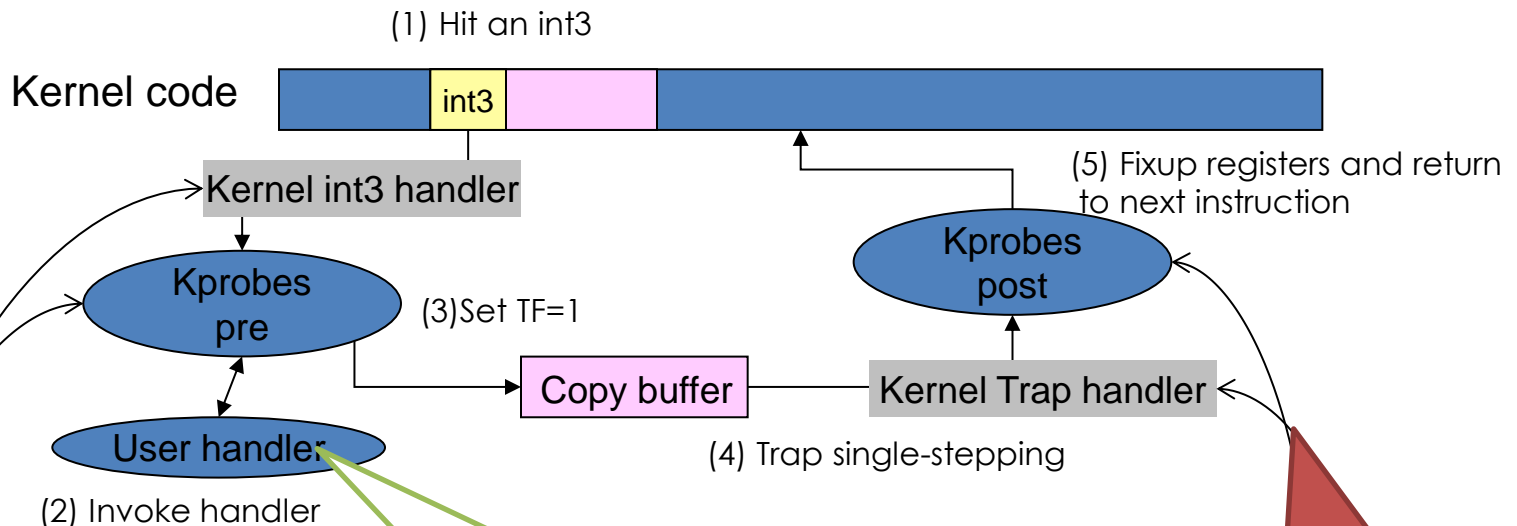
## Preparing



## Running



- It is dangerous to probe on some functions, that are called when a breakpoint/singlestep is executed



Probing here can cause endless loop on int3 handling

Probing here is safe, because kprobes can skip it (just do singlestep and return)

Probing here is also dangerous: kprobes can detect it, but cannot skip singlestep


⇒ These must be blacklisted with “**`_kprobes`**”

- Naming issue
  - \_\_kprobes means “prohibiting probes”, but it is misunderstood as “kprobes related functions”
  - “What? This function is not a part of kprobes.”
- Code cache fuzzing
  - \_\_kprobes == “\_\_attribute(section(“kprobes.text”))”  
This means moving the function another text area.
  - For the blacklisting, we don’t need to do that. Just need the function entry address and the size.
- No module support
  - In kernel module, adding \_\_kprobes doesn’t change anything.

- Represents correct meaning
  - Mark only symbols verified as black or gray
  - Similar usage to EXPORT\_SYMBOL()
- Do not fuzz code cache
  - Just save symbol address as data
  - Do not use separated text section
- Easy to support kernel module
  - It's just data
- User can now refer the blacklist via debugfs

➡ But which symbols are really black?



- Ingo's suggestion
  - The Blacklist is neither complete nor tested enough.  Um, right.
- How to test it?
  - One by one probe testing is not enough
  - To completely ensure the stability, we need to put kprobes on all functions in the kernel (and run them)
    - Usually, there are **30,000 - 40,000** functions in the kernel.

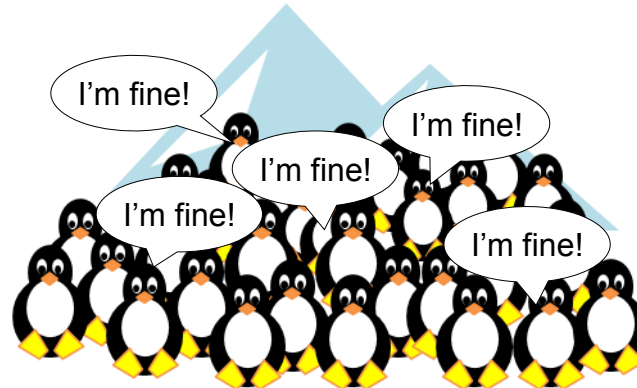
- Qualitative issue: “Does kprobes work fine?”



- Qualitative issue: “Does kprobes work fine?”



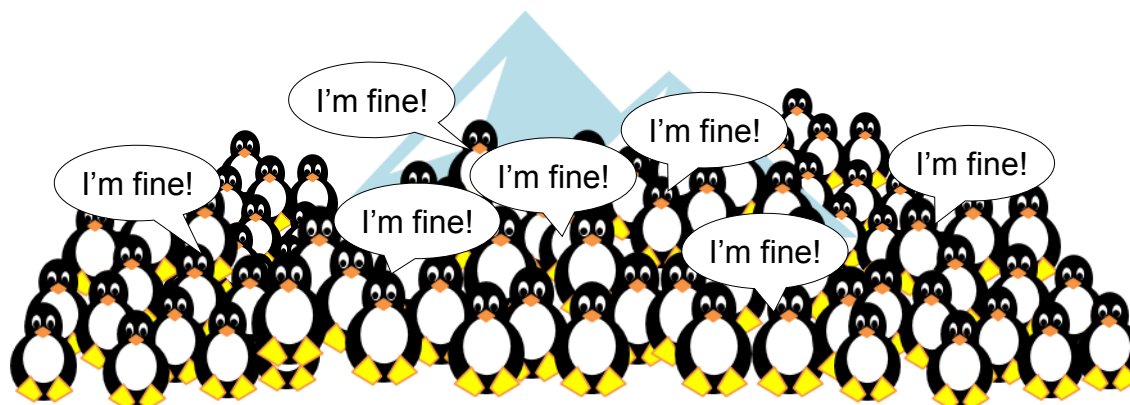
- Quantitative issue:



- Qualitative issue: “Does kprobes work fine?”



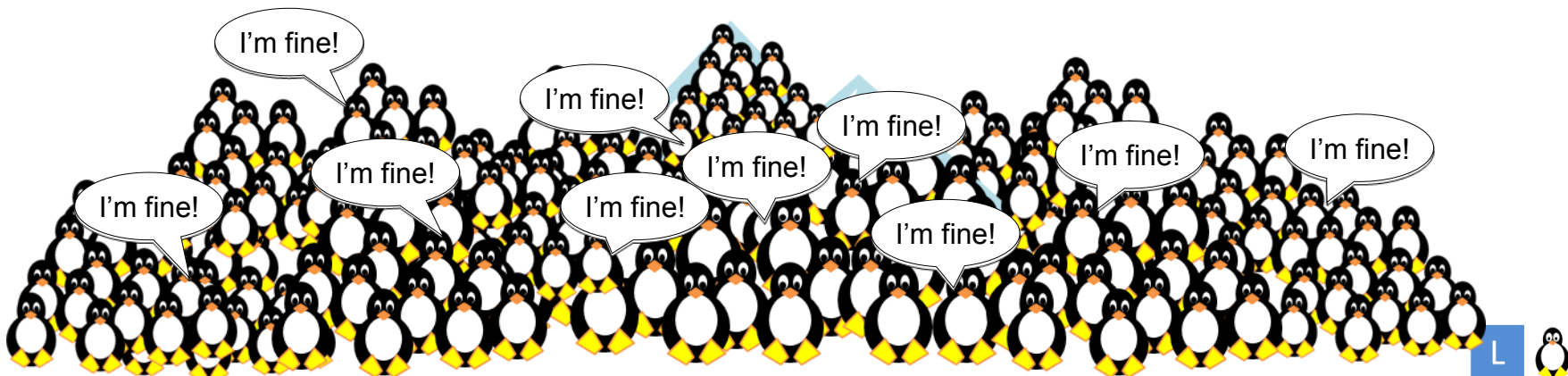
- Quantitative issue:



- Qualitative issue: “Does kprobes work fine?”



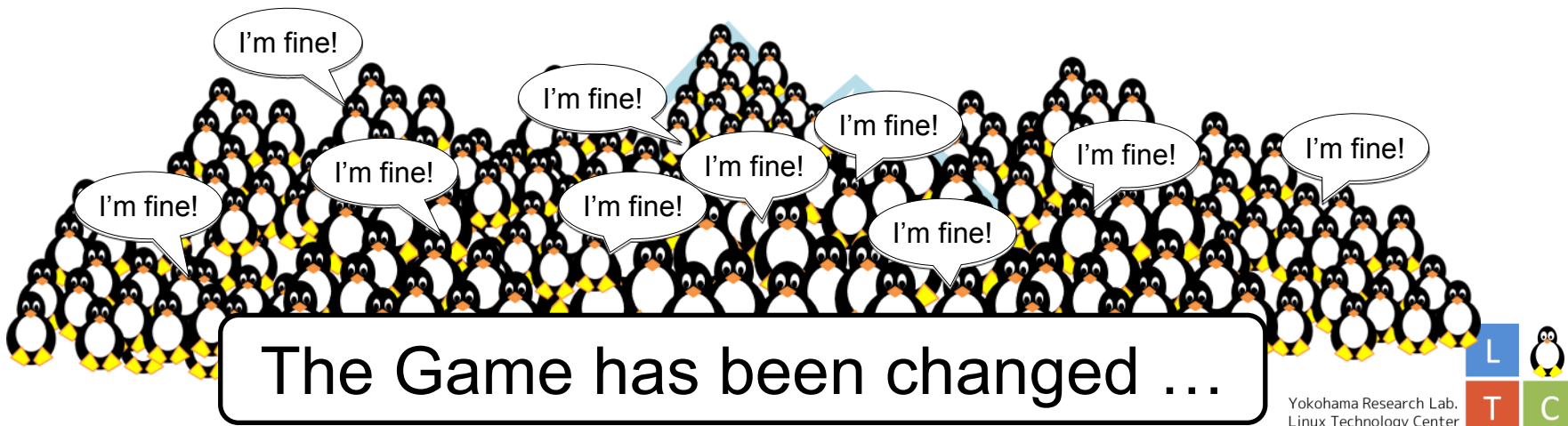
- Quantitative issue: “Does kprobes scale up?”



- Qualitative issue: “Does kprobes work fine?”



- Quantitative issue: “Does kprobes scale up?”



## Background Story

Kprobes Blacklist Improvement

Testing the Blacklist

Qualitative versus Quantitative

## Scalability Efforts

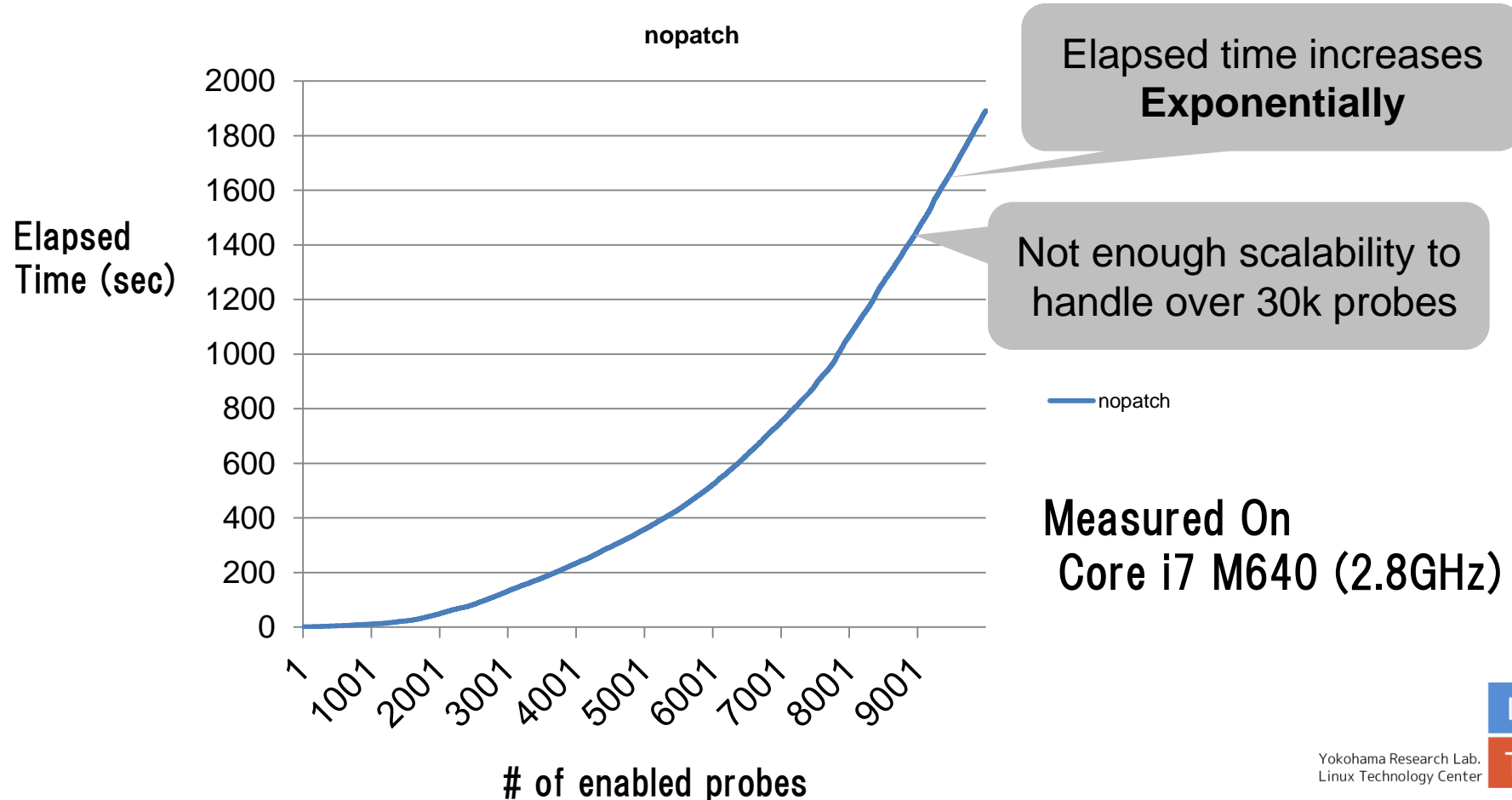
Enlarge the Hash Table

Cache the Hash List

Reduce Redundancy

## Conclusion and Discussion

- What happens if we put and enable kprobes on a massive number of functions?



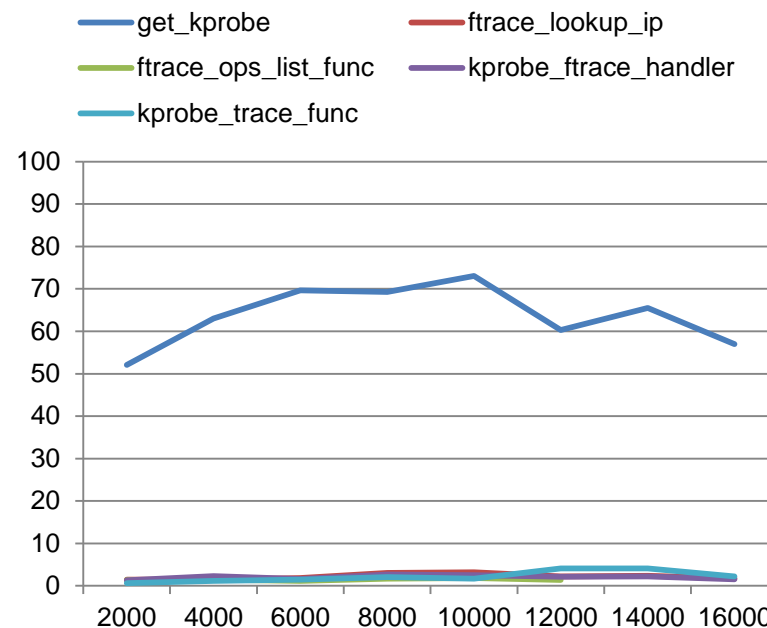
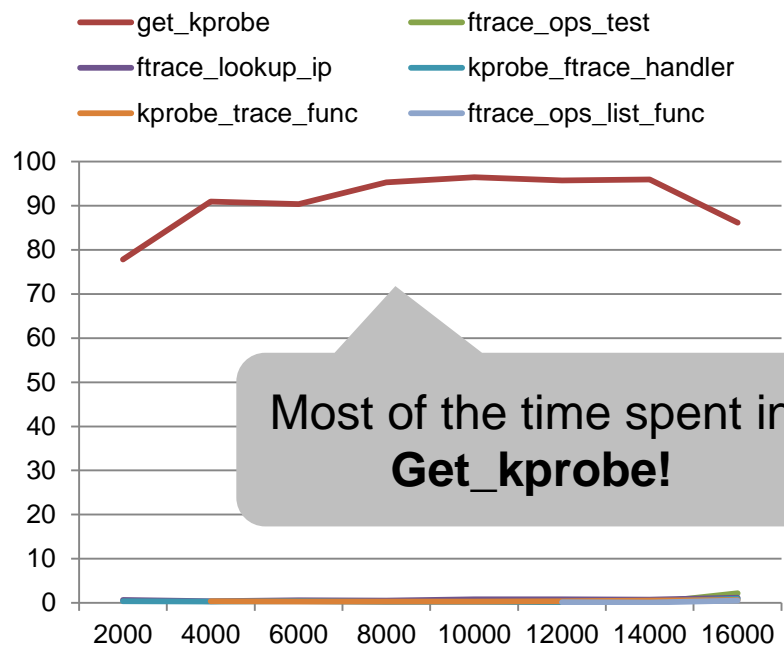


- Analysis target: kprobes scalability
  - A kernel feature, **not a user process**
  - Resource: **CPU and Memory**
  - Run everywhere in kernel (no specific workload)
- Perf record options

```
$ perf record -a -g --call-graph fp -e cycles -e cache-misses -e instructions sleep 60
```

- **-a**: all cpus, no specific process
- **-e cycles,cache-misses,instructions**
  - Cycles: Time consuming
  - Cache-miss: Memory consuming
  - Instructions: CPU consuming

- Using perf tools to clarify the bottleneck



- “get\_kprobe” is the bottleneck
  - And too many instructions are executed

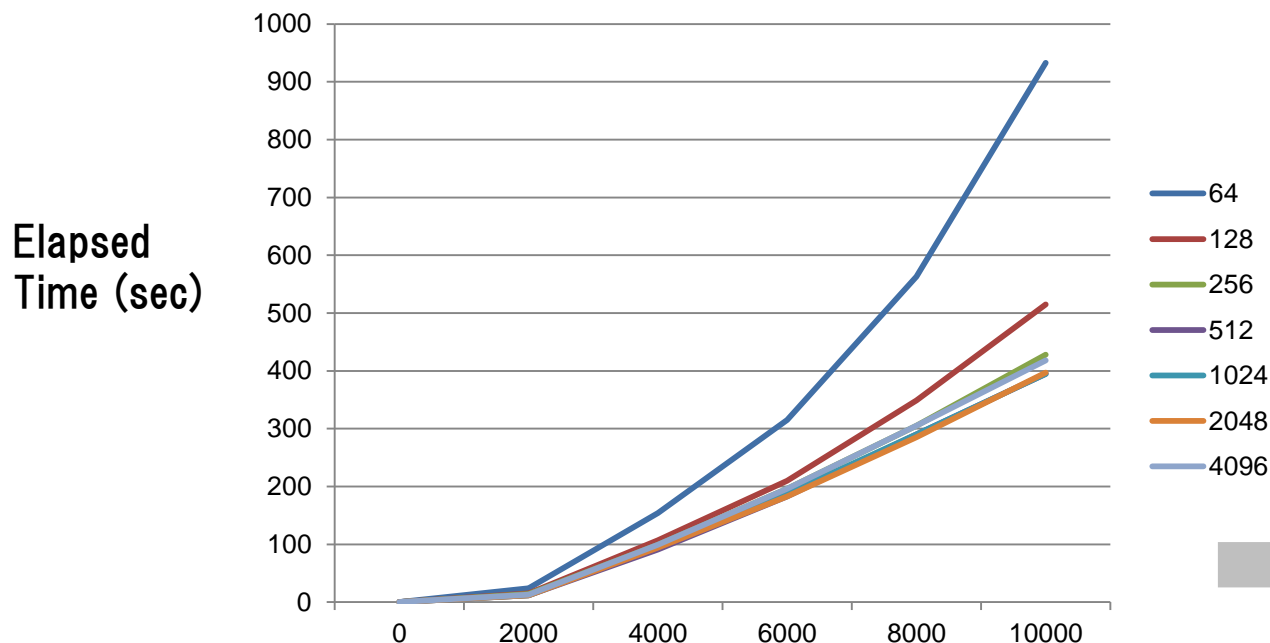
- Look up a kprobe from a hash-table by address

```
86.33 | head = &kprobe_table[hash_ptr(addr, KPROBE_HASH_BITS)];  
11.24 | hlist_for_each_entry_rcu(p, head, hlist) {  
      |     mov    (%rax), %rax  
      |     test   %rax, %rax  
      |     jne    60  
      |         if (p->addr == addr)  
      |             return p;  
      | }
```

- Table size is too small
  - It has just **64** entries for 30k.

➡ But, how large?

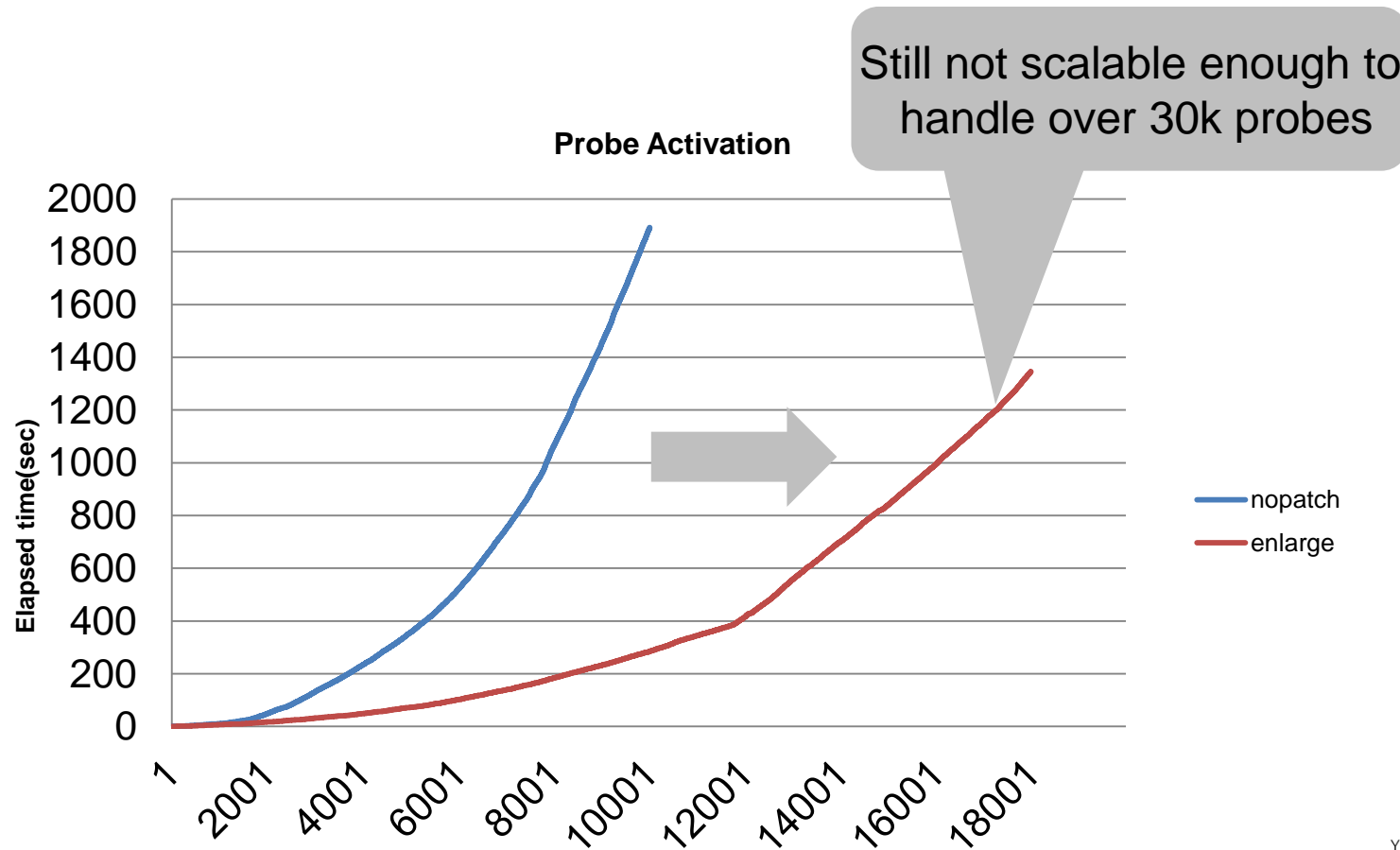
- Find the best size for hash-table with 10k probes



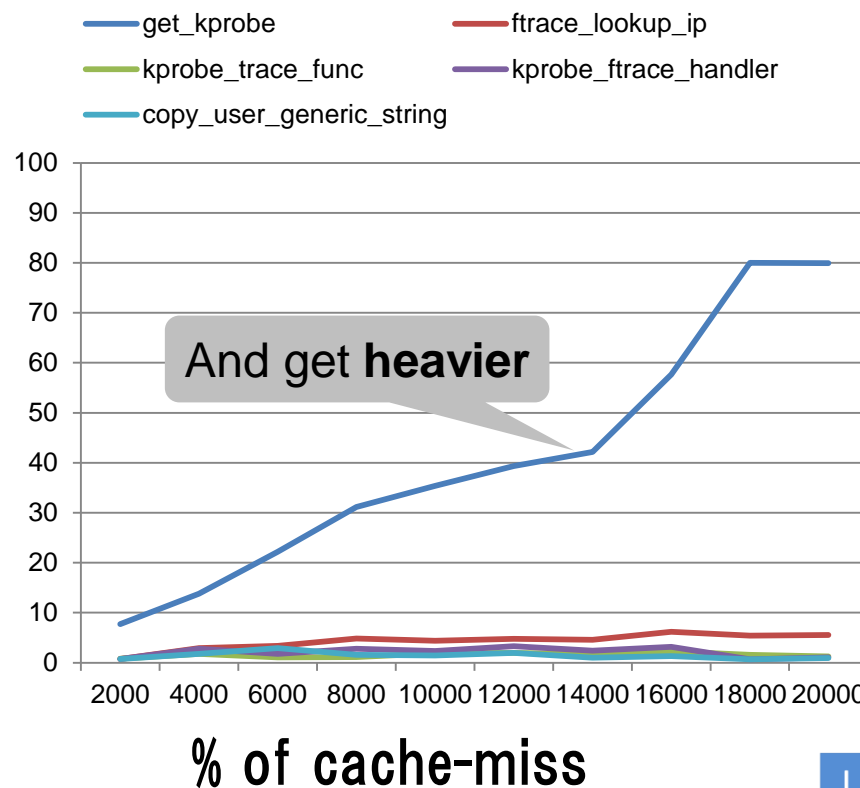
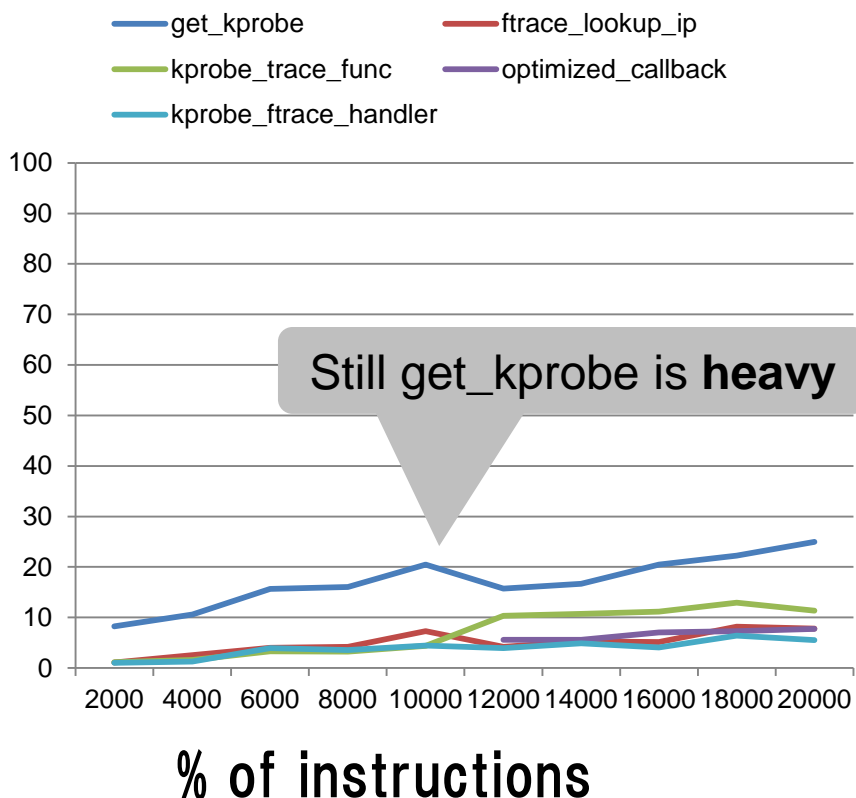
➔ **512 is the best**

# of probes	64	128	256	512	1024	2048	4096
0	0	0	0	0	0	0	0
2000	24	15	14	12	12	12	13
4000	154	107	99	91	97	94	99
6000	315	210	196	183	188	183	196
8000	563	349	305	290	290	285	305
10000	933	515	428	395	395	397	418

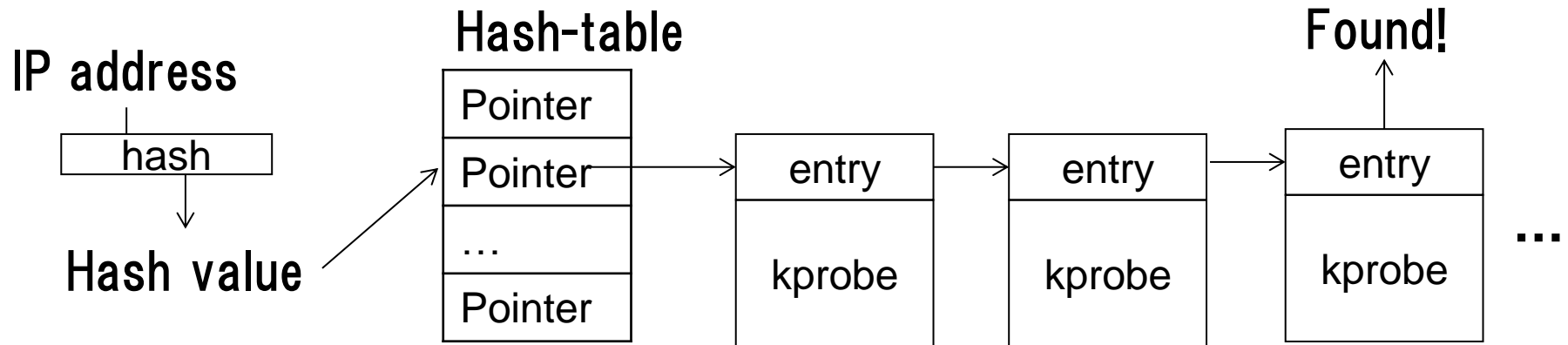
- What happens if we put kprobes on a massive number of functions? With 512 entries?



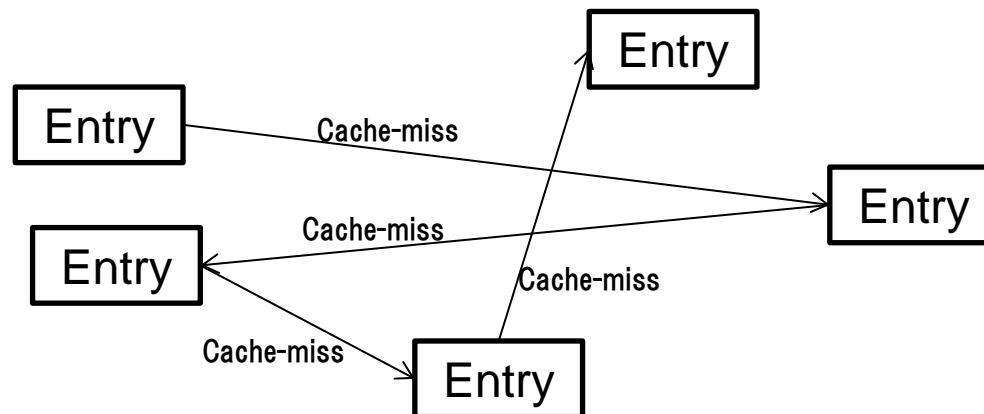
- Hash-list reduces the number of instructions, but increases cache-miss



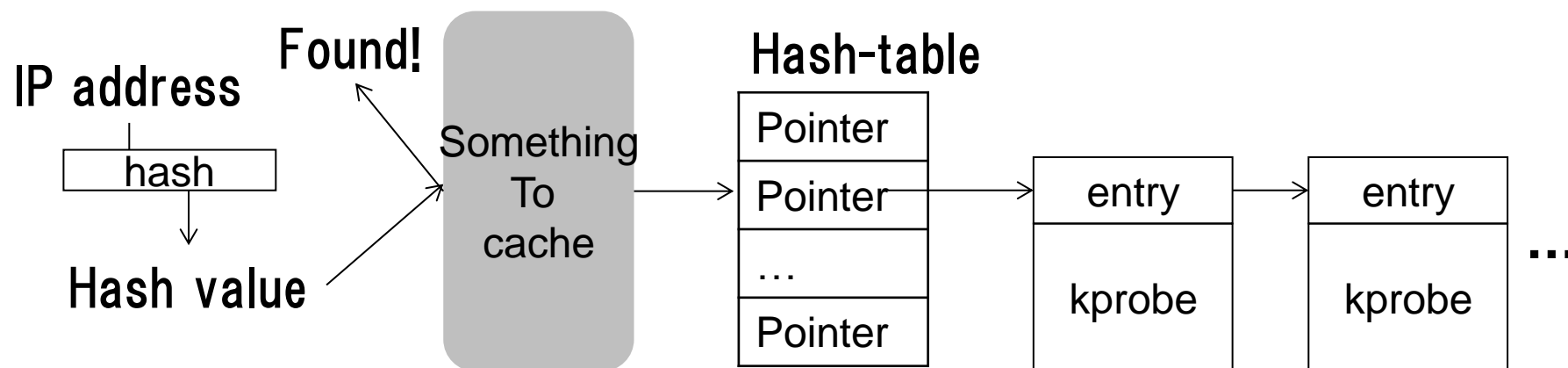
- Hash-list consists of a hash-table and lists



- Entries are scattered in kernel space

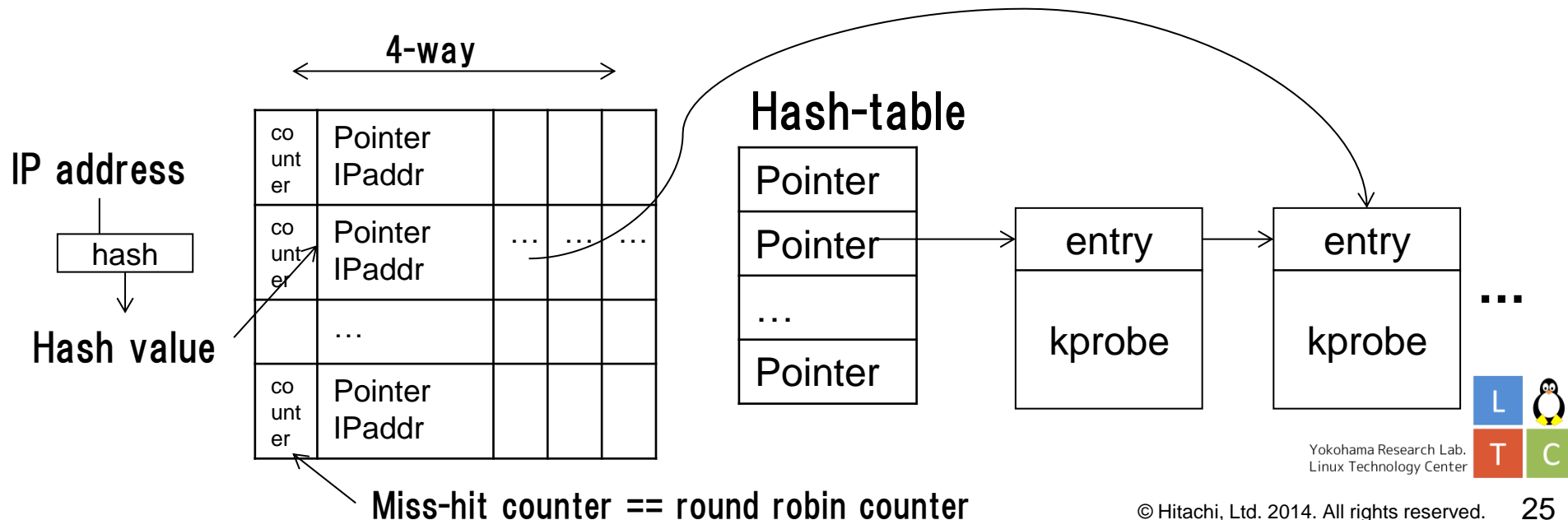


- To reduce “random” memory access
  - Hardware does that! Why can’t software do that?

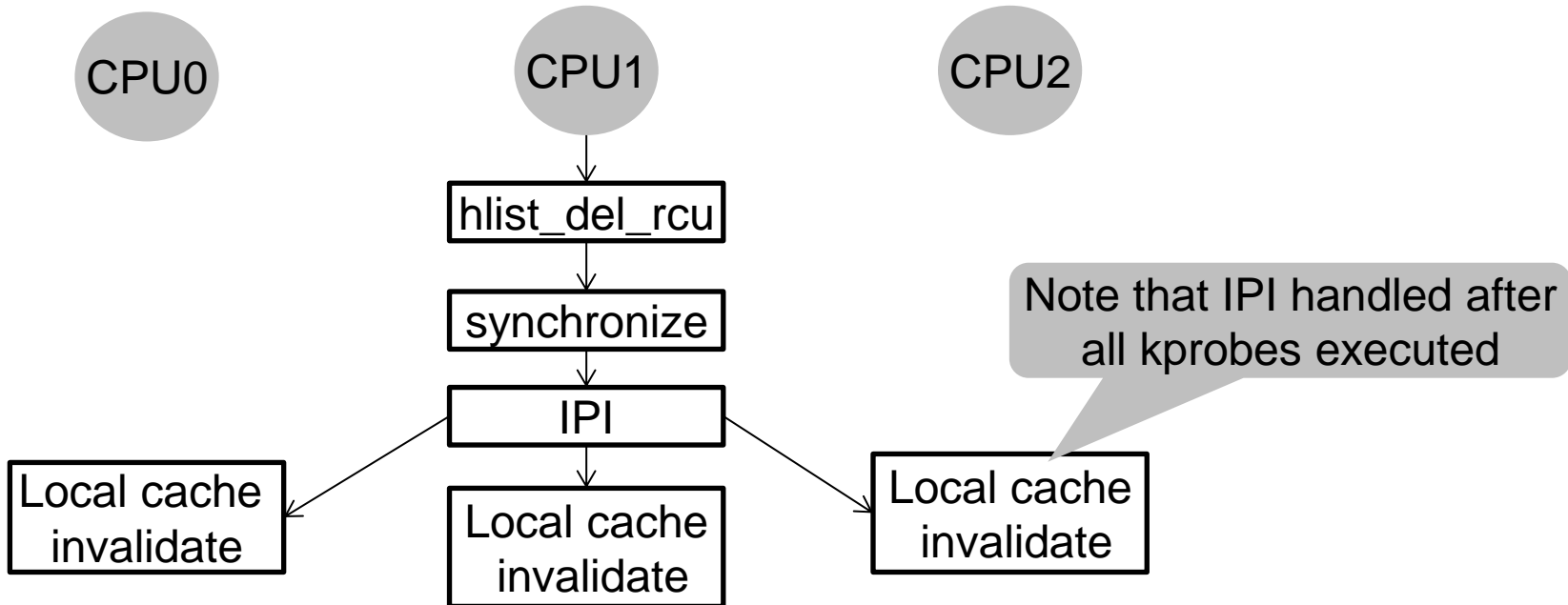




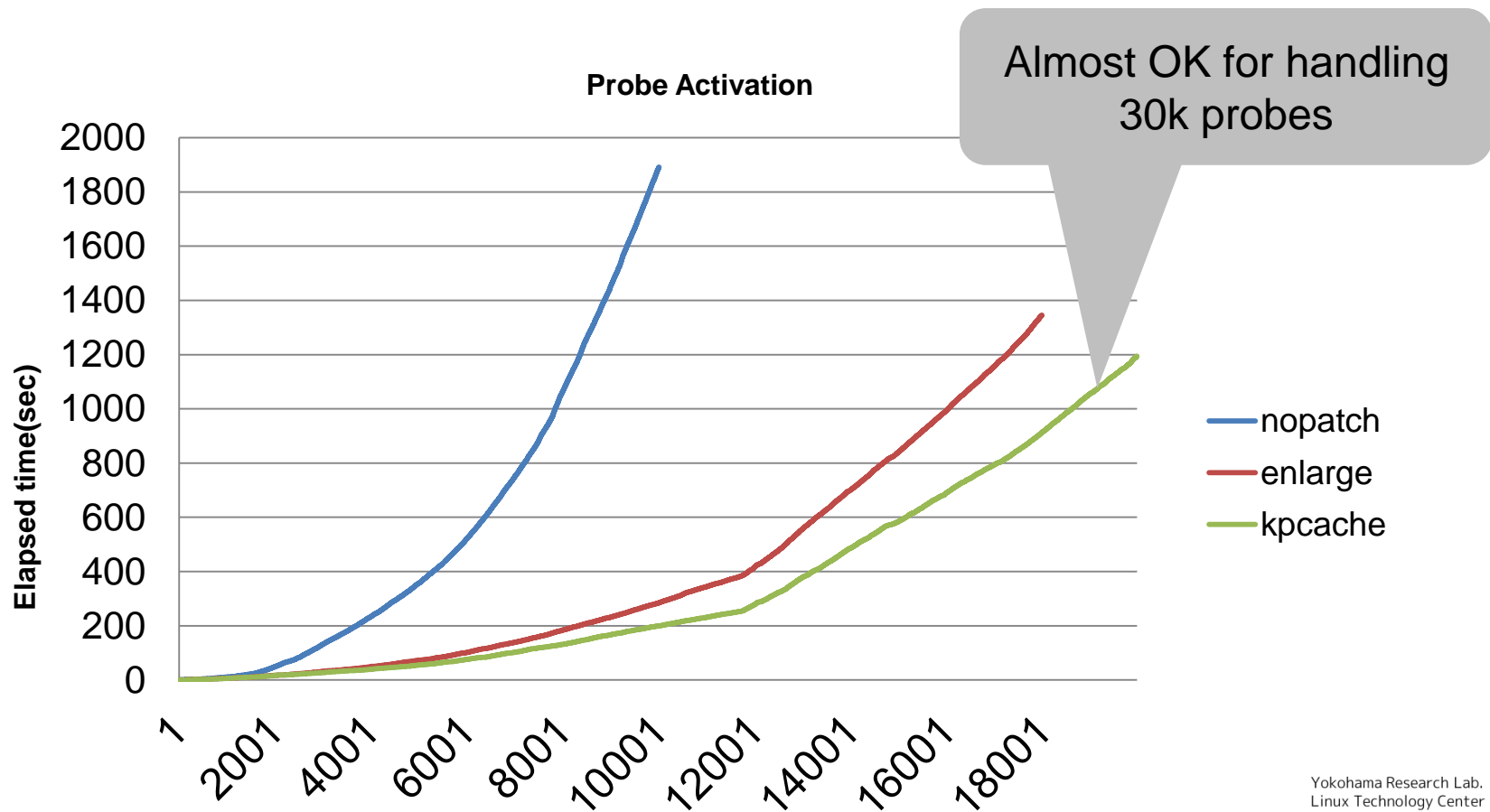
- Kpcache for caching hash-list
  - Per-cpu
    - Cache entries are replaced locally (no IPI needed)
  - 4-way set-associative
    - 4 entries for each cache entry
  - Hashlist cache
    - Hash value can be shared with hashlist and cache
  - Round-robin Refill, invalidate-protocol



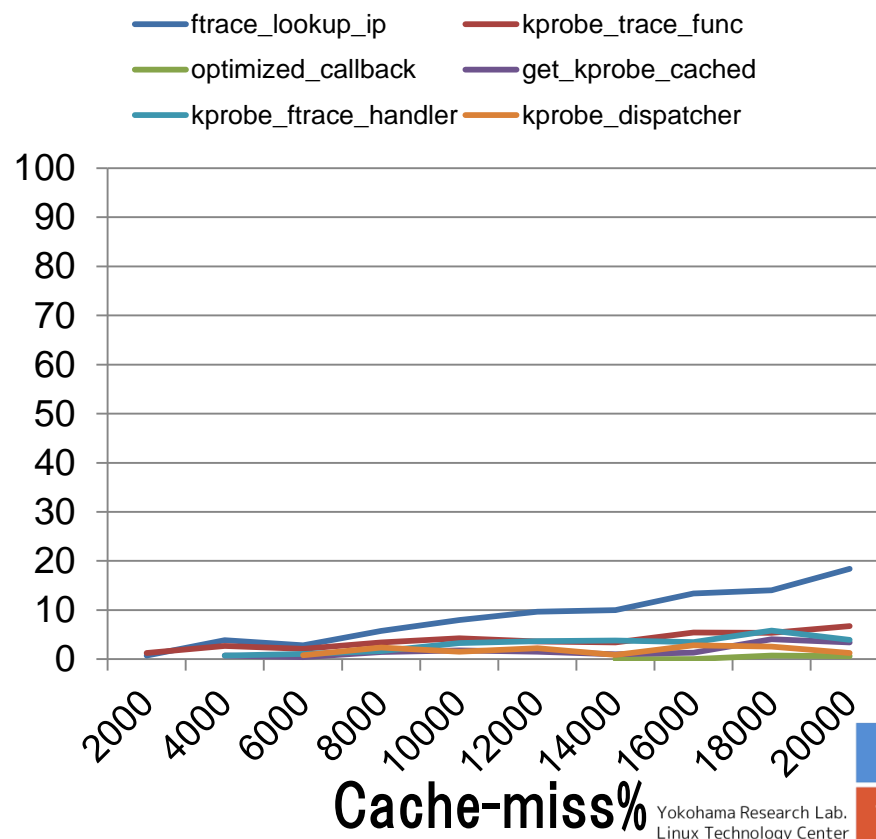
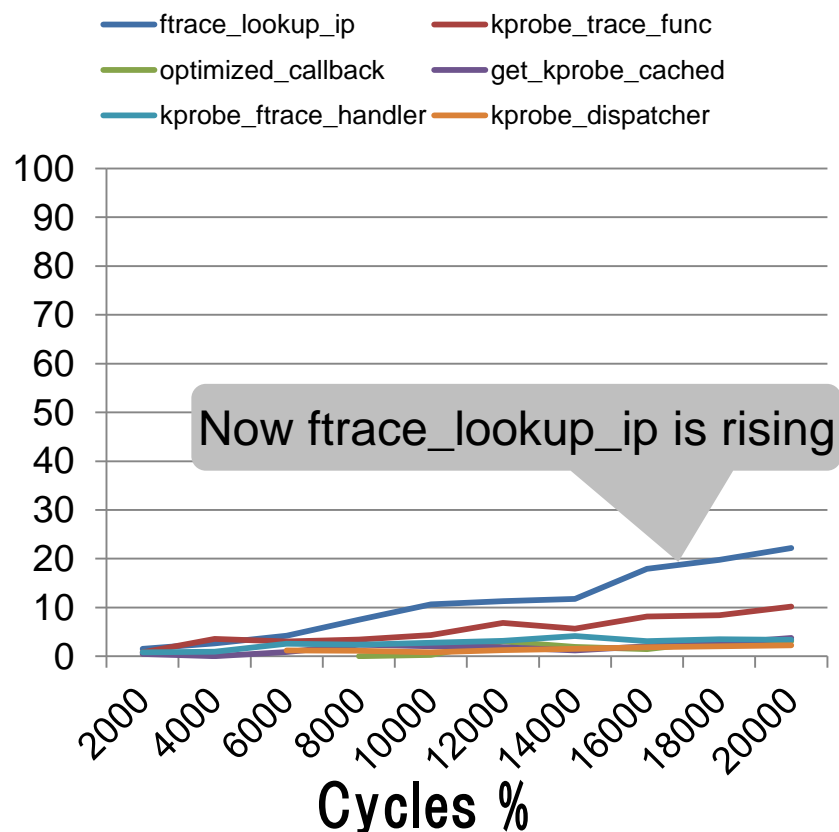
- Cache-miss always causes update (round robin)
- Kprobes-unregister causes invalidation



- What happens if we put kprobes on a massive number of functions? With 512 entries and kpcache?



- Ftrace\_lookup\_ip is a dominant bottleneck
  - Cycles and cache-miss show it.

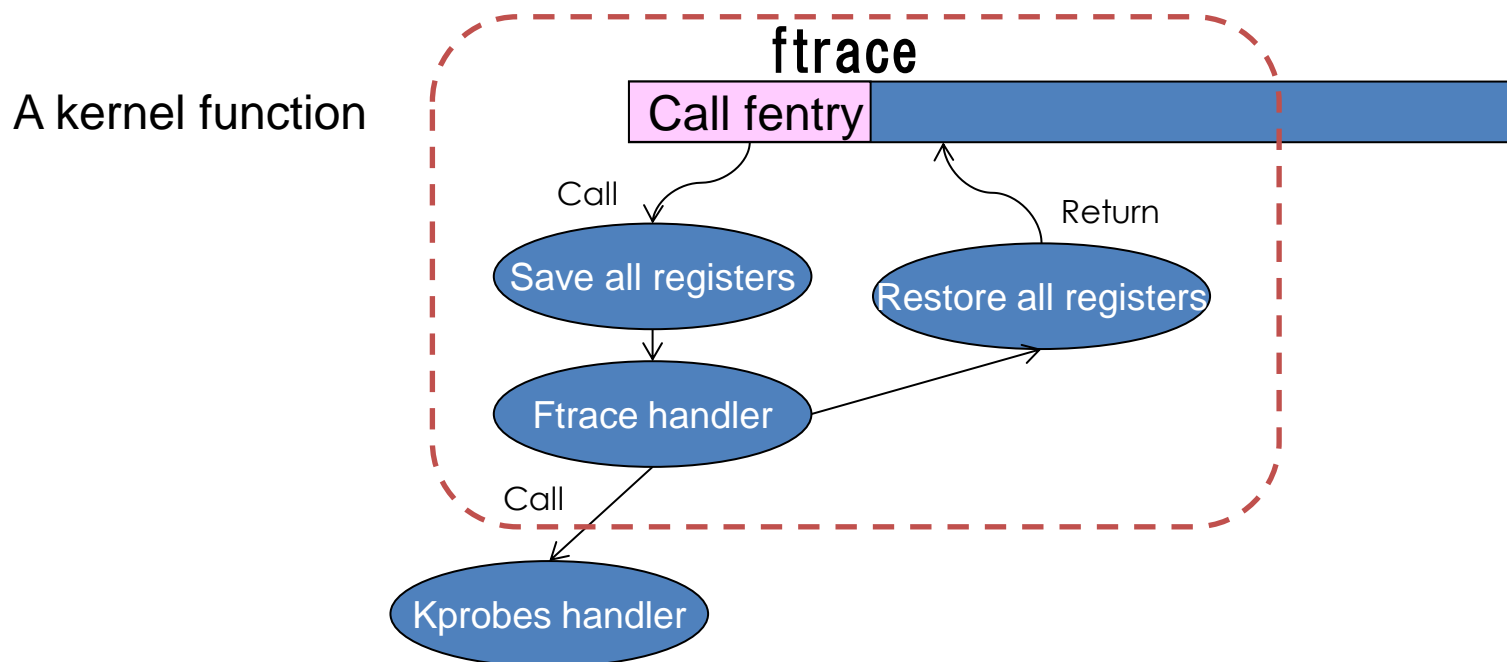


- This test uses kprobes. Why does ftrace matter?

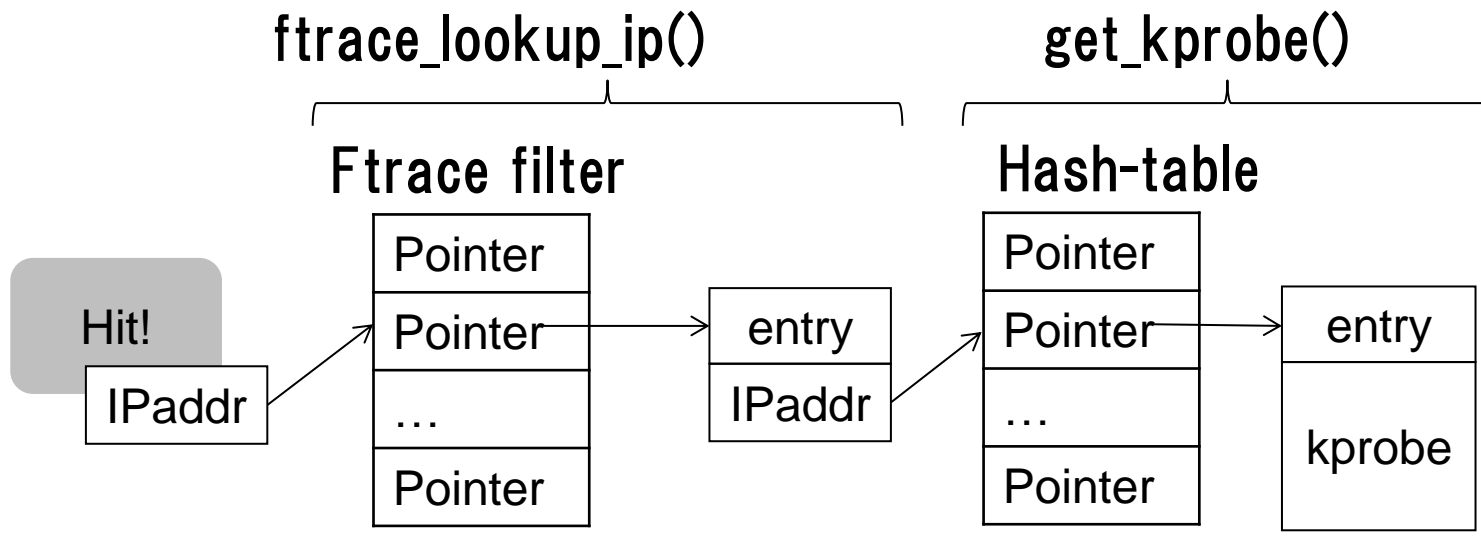


- Since kprobes uses ftrace if the probe-point is on the function entry
  - We call it “ftrace-based kprobes”

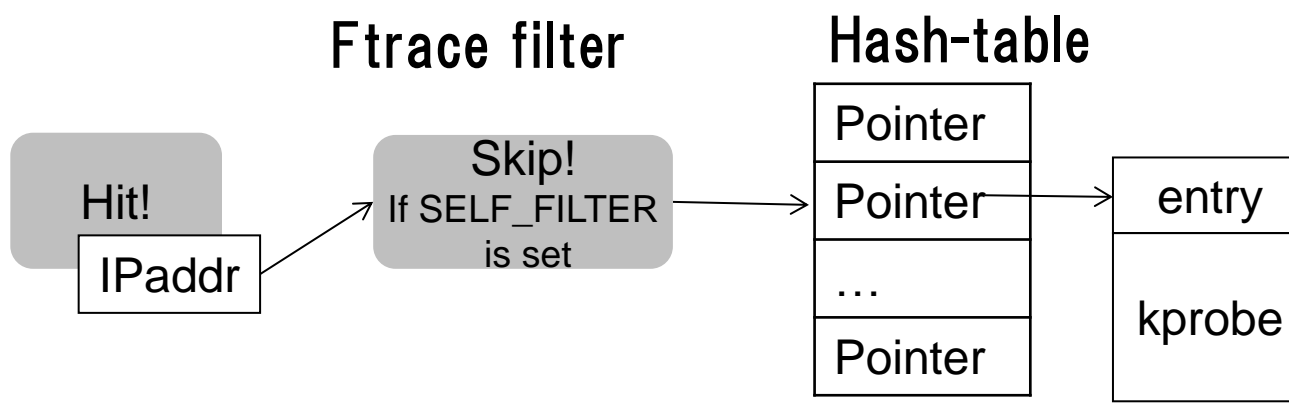
- The entries of functions are hooked with ftrace.
  - With -mfentry option(on x86), puts mcount call on the 1<sup>st</sup> byte of the functions. (conflict with kprobes)
  - With FTRACE\_OPS\_FL\_SAVE\_REGS, ftrace saves all registers, same as an interrupt handler



- ftrace-based kprobe adds new ftrace\_ops for handling ftrace mcount handler.
- In this case, ftrace starts checking which ftrace\_ops should be invoked from ip address
- Each ftrace hit causes 2 other hashtable checks!
  - Mcount->ftrace->hash check->kprobe->hash check

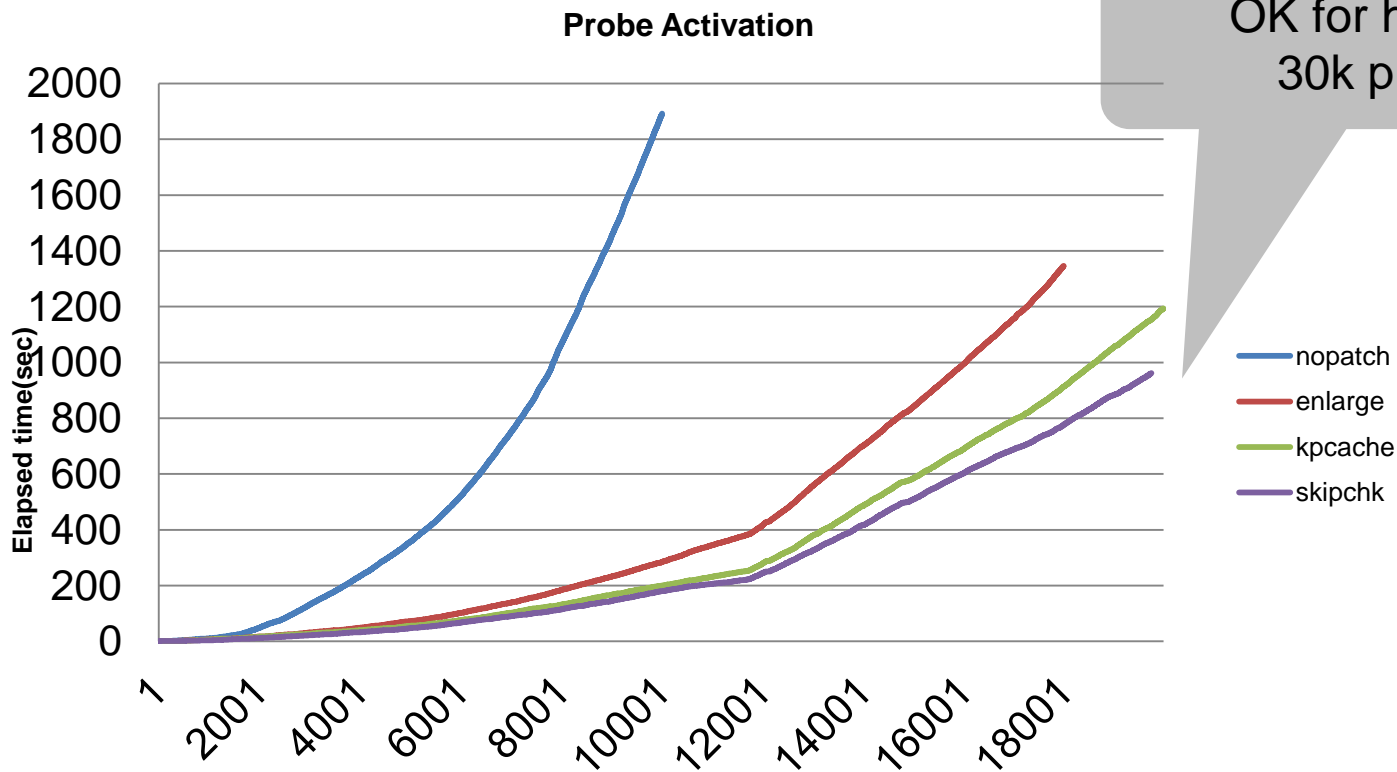


- kprobes itself has its own hashlist(filter)
  - Ftrace doesn't need to check its hashlist. Then, skip it!
- **FTRACE\_OPS\_FL\_SELF\_FILTER**
  - With this flag, ftrace skips checking filter(hashlist) and always calls the ops->func.
  - Kprobes always checks its own hashlist first, and if there is no hit, just returns.

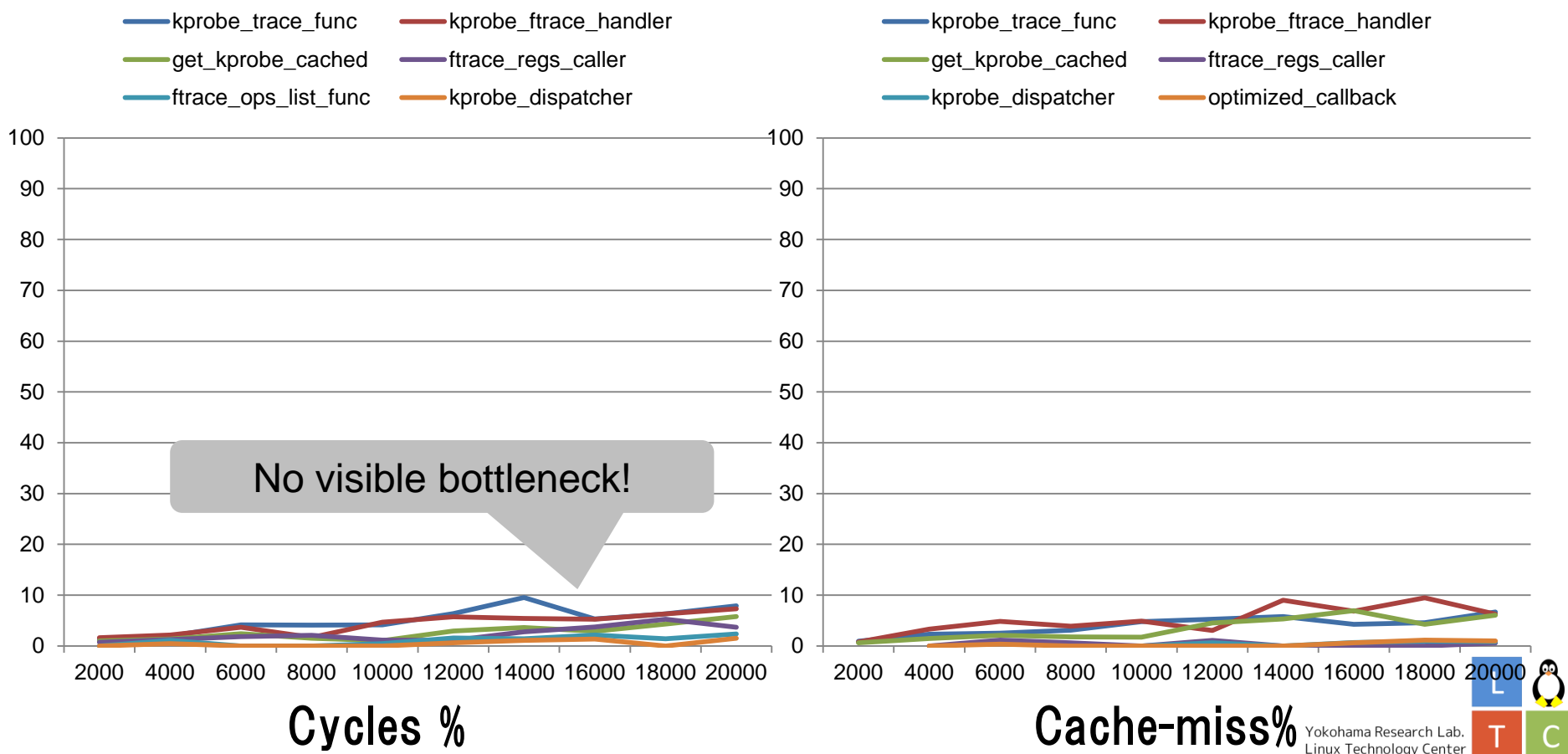




- What happens if we put kprobes on a massive number of functions? With 512 entries, kpcache, and self-filter?



- Win! There is no obvious bottleneck
  - All functions consume less than 10%



- Finally, it took just 2254 sec for enabling 37222 probes. 😊
  - This test still takes a long time, but is possible to finish!

## Background Story

Kprobes Blacklist Improvement

Testing the Blacklist

Qualitative versus Quantitative

## Scalability Efforts

Enlarge the Hash Table

Cache the Hash List

Reduce Redundancy

## Conclusion and Discussion

- kpcache consumes "some" memory
  - 32KB table, 4KB counter = 36KB/core
  - 8core -> 256KB table, 32KB index
  - 32core -> 1MB/128KB
  - 256core -> 8MB/1MB = 9MB for cache
- (outdated)Recommend not to enable by default
  - Anyway, this feature is only good for stress testing with massive multiple kprobes
  - CONFIG\_KPROBE\_CACHE=n by default

- **CONFIG\_KPROBE\_CACHE** is gone
  - Makes the code simple and does not confusing users.
  - Anyway, it is easy to remove kpcache if needed.
    - Requires just 6 lines of code.

```
--- a/kernel/kprobes.c
+++ b/kernel/kprobes.c
@@ -91,6 +91,7 @@ static raw_spinlock_t *kretprobe_table_lock_ptr(unsigned long
static LIST_HEAD(kprobe_blacklist);
static DEFINE_MUTEX(kprobe_blacklist_mutex);

+#ifdef CONFIG_KPROBE_CACHE
/* Kprobe cache */
#define KPCACHE_BITS 2
#define KPCACHE_SIZE (1 << KPCACHE_BITS)
@@ -181,6 +182,11 @@ static void kpcache_invalidate(unsigned long addr)
    * So it is already safe to release them beyond this point.
    */
}
+#else
+#define kpcache_get(hash, addr) (NULL)
+#define kpcache_set(hash, addr, kp) do {} while (0)
+#define kpcache_invalidate(addr) do {} while (0)
+#endif
```

- Can hash-cache technique be used in general?
  - Yes, if ...
    - the hash table is used so frequently
    - the hash table is sparse
    - the hash client can be pinned down to one cpu
    - the hash can be invalidated via IPI
  - > Perhaps, ftrace could be a good candidate
- Is it applicable for Userspace?
  - Per-cpu is hard to implement in Userspace
  - Per-thread/Shared cache may be possible

- Now kprobes is ready for a massive number of probes
- “Perf” is great for the bottleneck analysis
- There are many ways to solve performance bottlenecks
  - Optimize hash-size
    - If hlist-lookup requires many instructions
  - Cache hash-list
    - If hlist-lookup causes many cache-misses
  - Remove redundancy
    - If you find redundant code



- More testing
  - Kprobes-fuzzer: Probe random addresses
  - Test without ftrace-based kprobe (only with native kprobes)
  - Test kretprobe/jprobe too

Our Adventure has just started!  
(Please stay tuned for next series!)

**HITACHI**  
**Inspire the Next**

*Thank you!*

Yokohama Research Lab.  
Linux Technology Center



- Linux is a trademark of Linus Torvalds in the United States, other countries, or both.
- Other company, product, or service names may be trademarks or service marks of others.

- UnixBench Index (4 CPUs)

	No Probe	Probe All Funcs	Performance%
Dhrystone	5664.5	5440.0	96.0%
Whetstone	2314.9	2258.5	97.6%
Execl	2600.0	95.5	3.7%
FileCopy 1024	3128.2	58.0	1.9%
FileCopy 256	2098.8	34.9	1.7%
FileCopy 4096	6268.6	151.3	2.4%
Pipe	2209.4	39.0	1.8%
Context-switch	1475.7	26.9	1.8%
Process create	2100.8	72.3	3.4%
Shell (single)	2898.2	188.5	6.5%
Shell (8 process)	3479.8	177.1	5.1%
System Call	2576.2	51.1	2.0%
Total	2817.0	137.7	4.9%