kretprobe可扩展性性能提升
高性能无锁对象池objpool实现

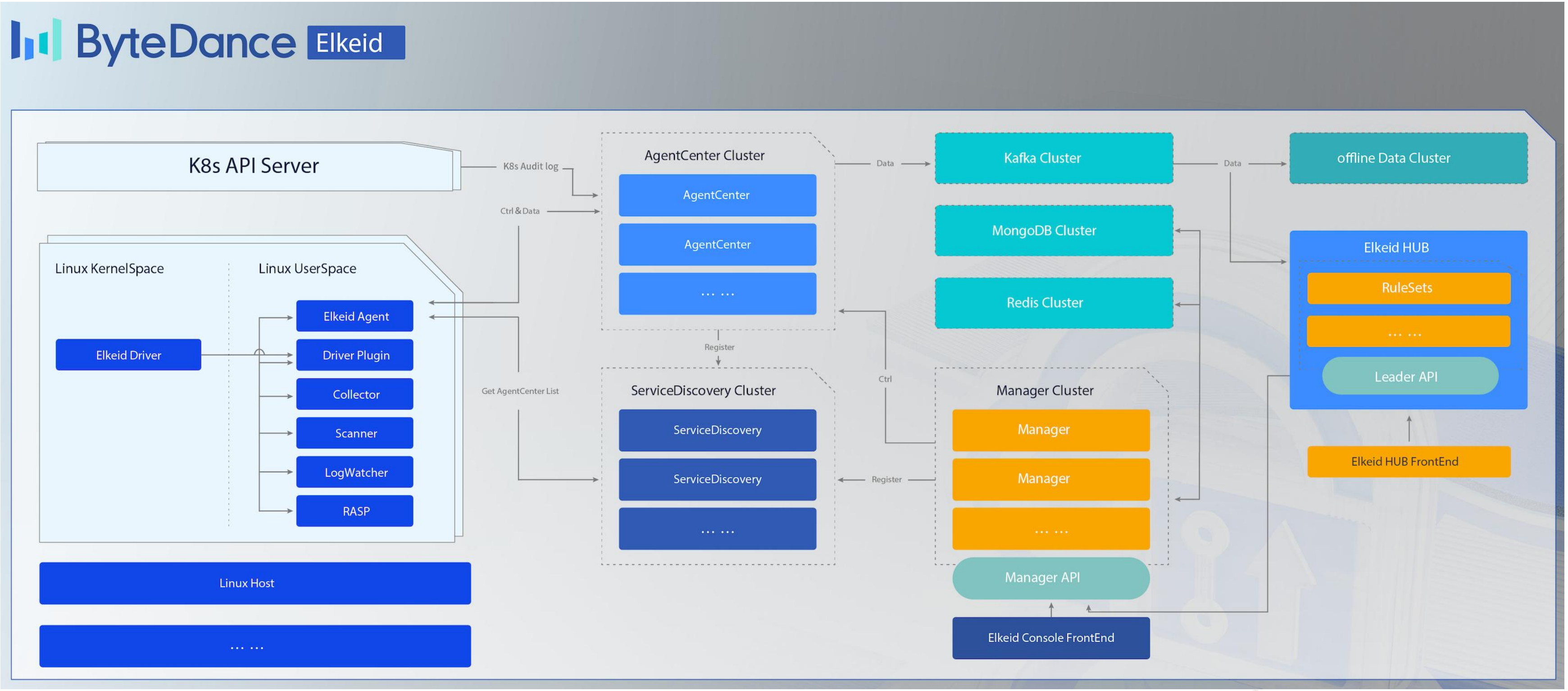
巫强 - 字节跳动终端安全

2024/10/26

CHINA LINUX KERNEL
中国内核开发者大会

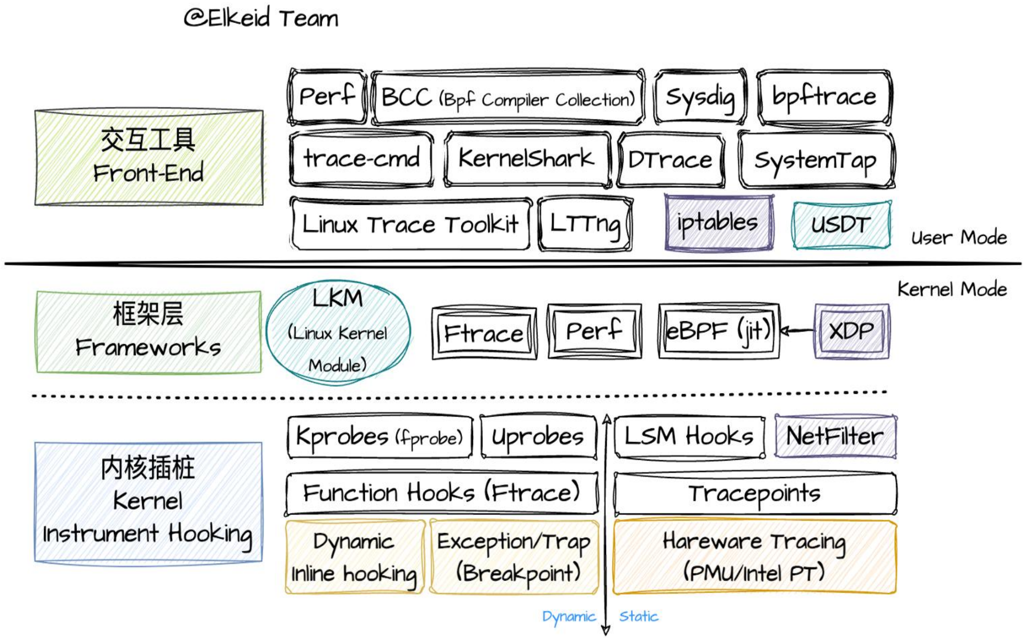
ByteDance字节跳动

安全业务- Linux内核试金石

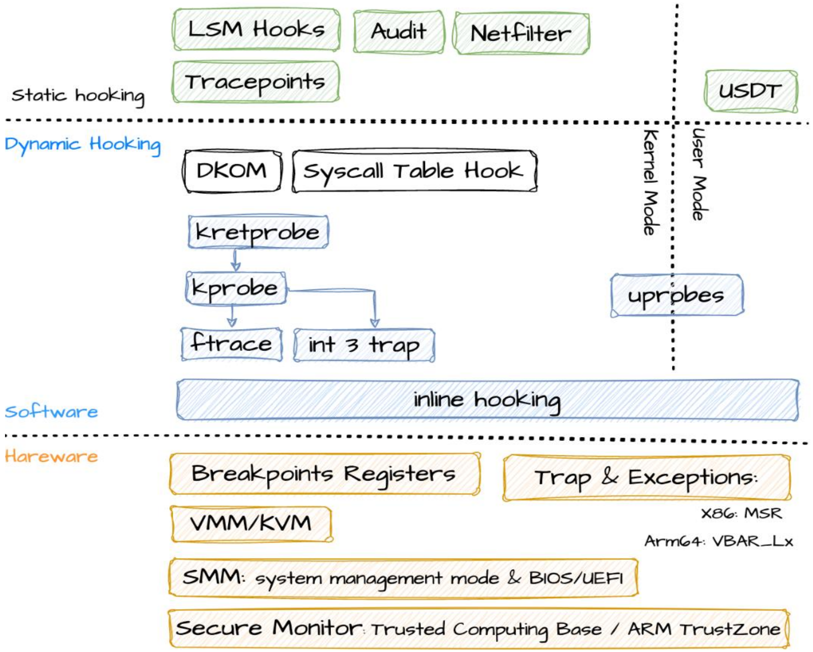


字节开源HIDS：https://github.com/bytedance/elkeid

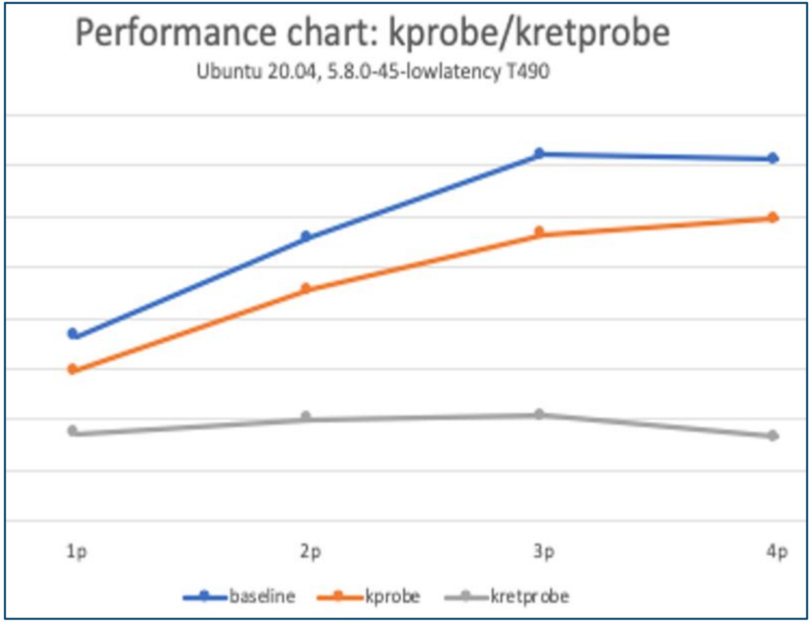
Linux动态跟踪技术

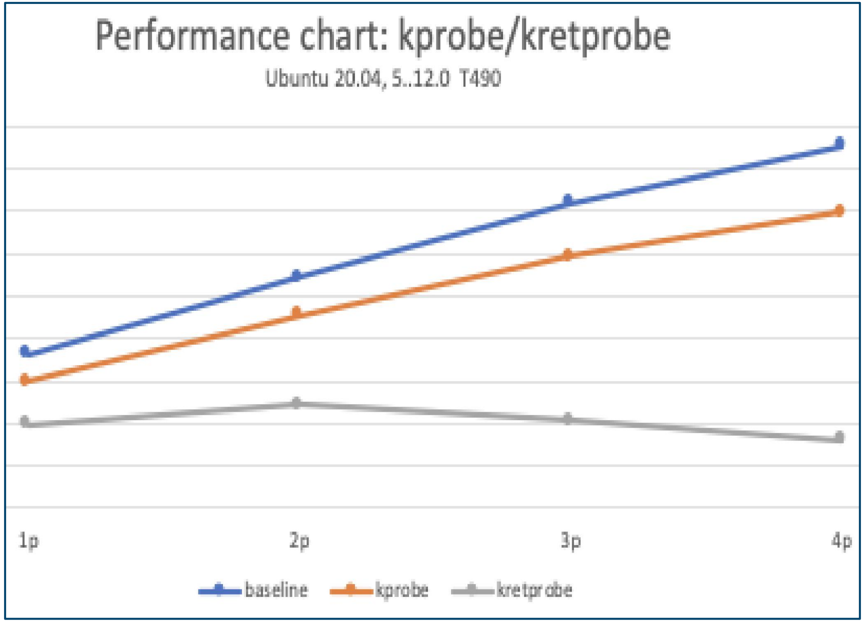


kretprobe = kprobe + stack hook



kretprobe的~~性能~~死锁问题



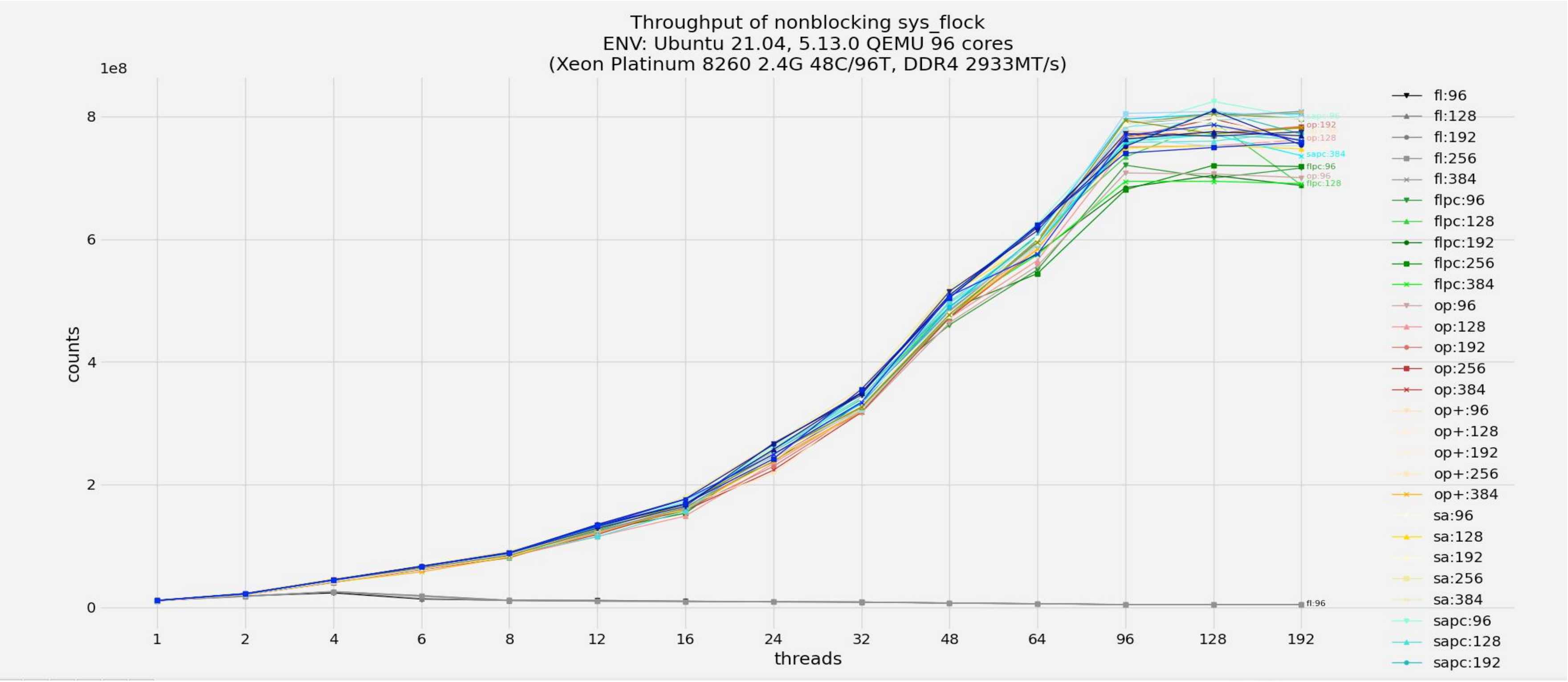


无锁不是银弹

- 竞争问题：无锁 (lockfree) 可能会加剧竞争
- 公平问题：NUMA、大小核、KVM
- 复杂度高：不同的CPU架构上的实现差异

percpu-freelsit vs objpool vs ...

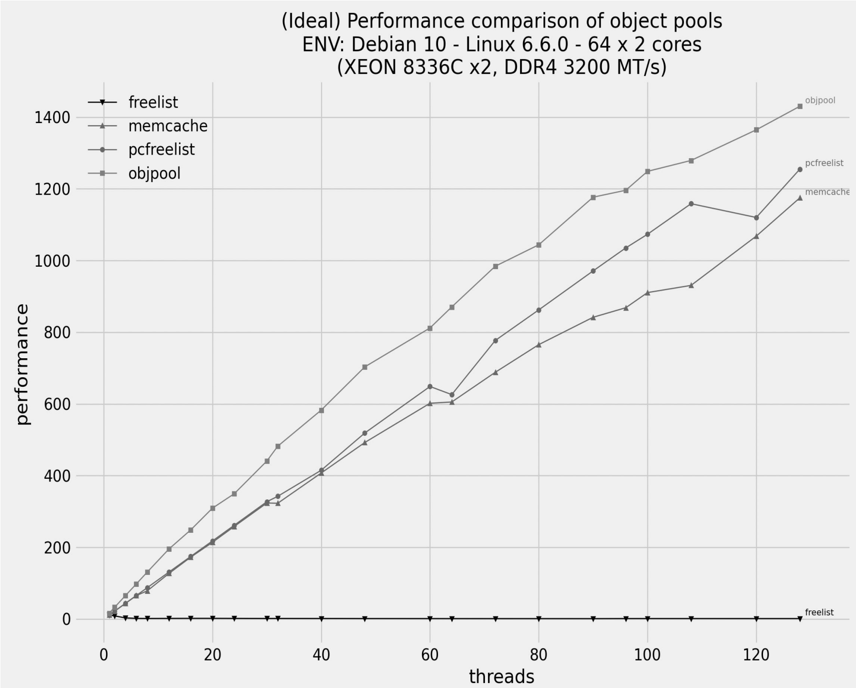
CHINA LINUX KERNEL
中国内核开发者大会

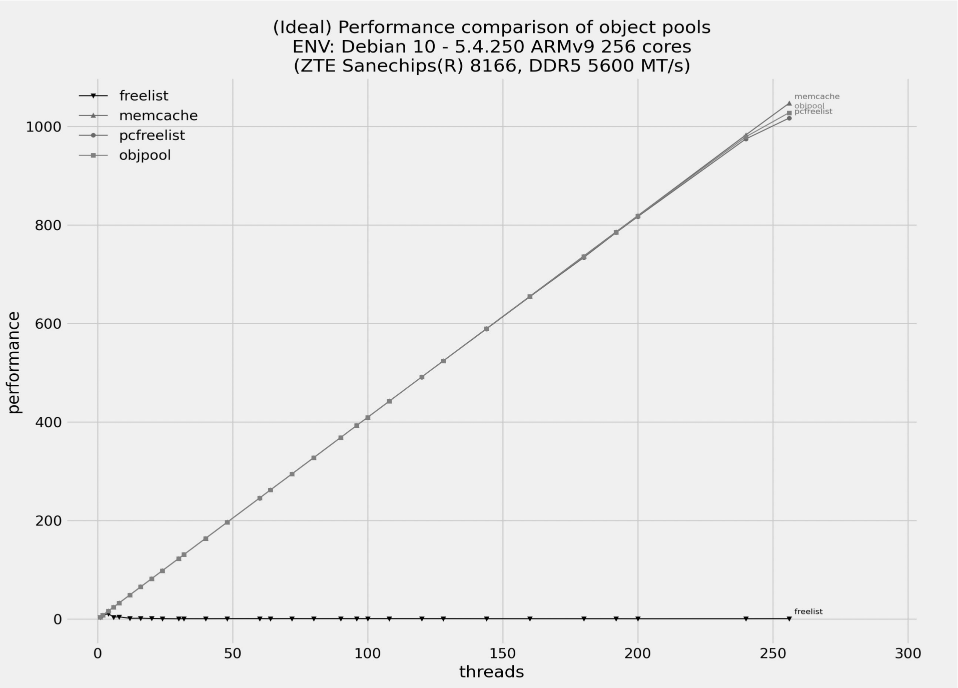


ByteDance字节跳动

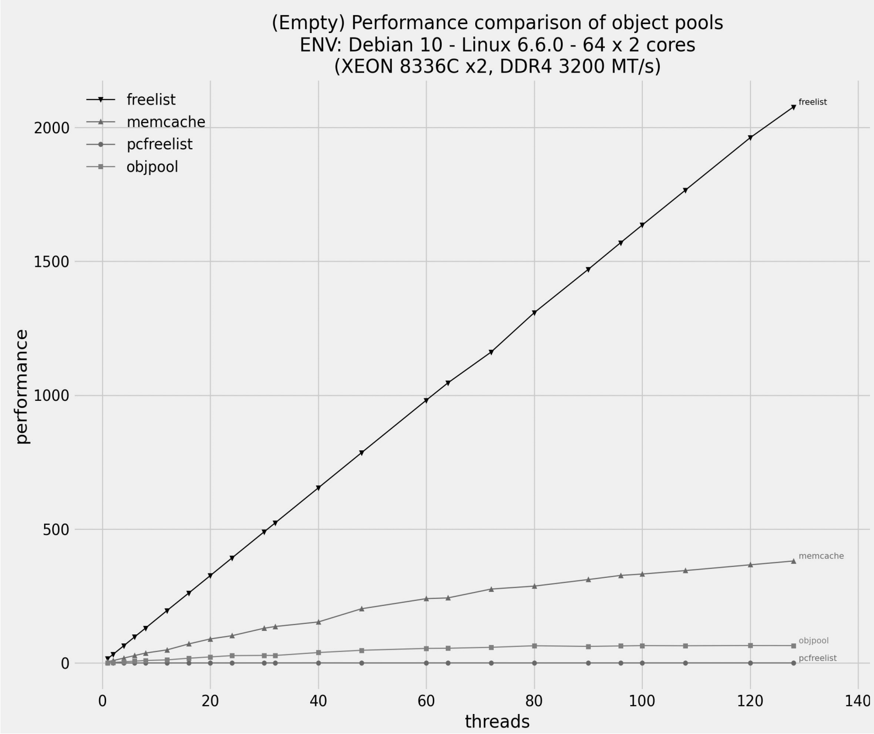
理想状态下的性能对比

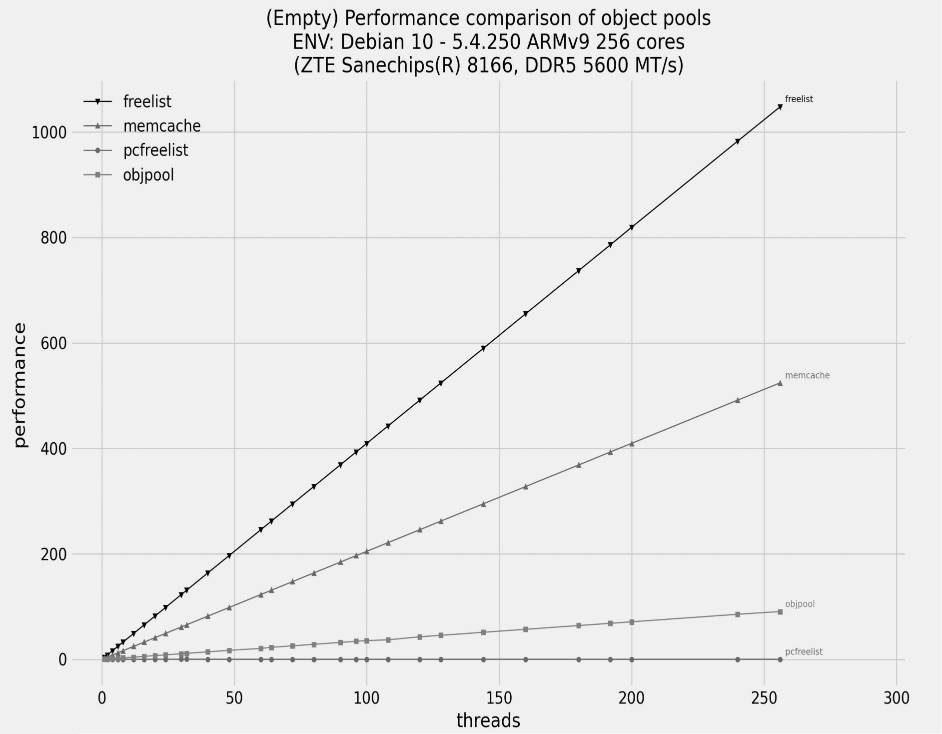
CHINA LINUX KERNEL 中国内核开发者大会





“最坏情况”才是决定性的？


(Empty) Performance comparison of object pools
ENV: Debian 10 - Linux 6.6.0 - 64 x 2 cores
(XEON 8336C x2, DDR4 3200 MT/s)
freelist
memcache
pcfreelist
objpool
performance
threads
0
500
1000
1500
2000
0
20
40
60
80
100
120
140


(Empty) Performance comparison of object pools
ENV: Debian 10 - 5.4.250 ARMv9 256 cores
(ZTE Sanechips(R) 8166, DDR5 5600 MT/s)
freelist
memcache
pcfreelist
objpool
performance
threads
0
200
400
600
800
1000
0
50
100
150
200
250
300

可扩展性的评估

Amdahl's Law:

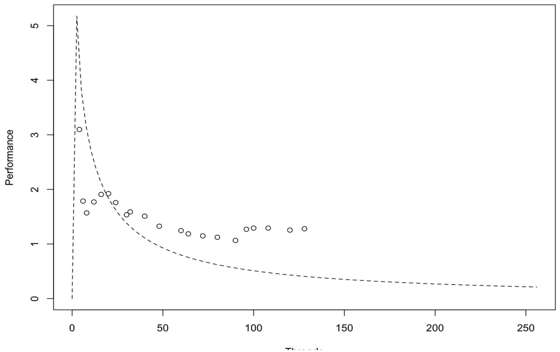
$$T_p = \sigma T_1 + \frac{(1-\sigma)T_1}{p}$$

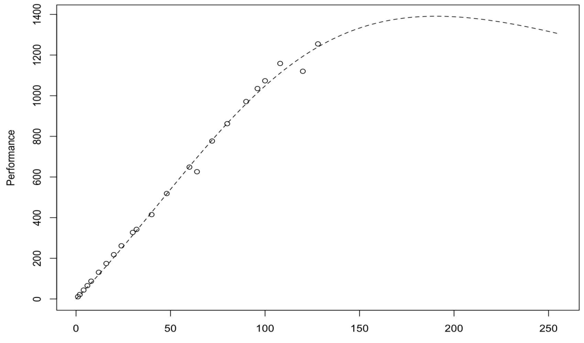
$$S(p) = \frac{p}{1+\sigma(p-1)}$$

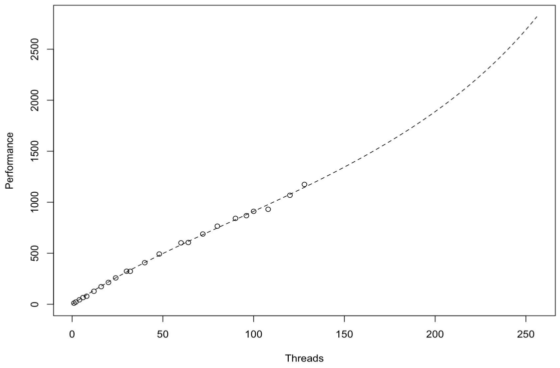
Universal Scalability Law (USL)：

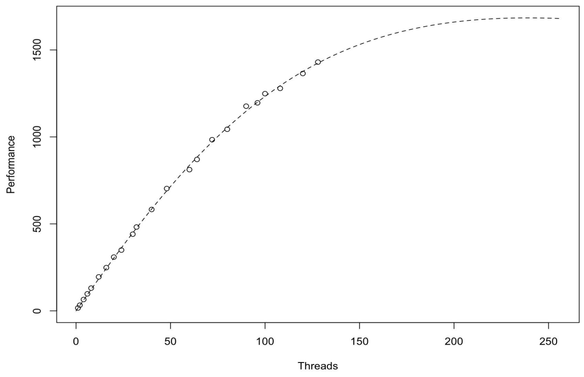
$$C(p) = \frac{p}{1+\sigma(p-1)+\kappa p(p-1)}$$

USL计算与预测


Performance
0
1
2
3
4
5
0
50
100
150
200
250


Performance
0
200
400
600
800
1000
1200
1400
0
50
100
150
200
250


Performance
0
500
1000
1500
2000
2500
0
50
100
150
200
250
Threads


Performance
0
500
1000
1500
0
50
100
150
200
250
Threads

percpu-freelist vs objpool

CHINA LINUX KERNEL
中国内核开发者大会

```c
static inline void __freelist_add(struct freelist_node *node, struct freelist_head *list)
{
	/*
	 * Since the refcount is zero, and nobody can increase it once it's
	 * zero (except us, and we run only one copy of this method per node at
	 * a time, i.e. the single thread case), then we know we can safely
	 * change the next pointer of the node; however, once the refcount is
	 * back above zero, then other threads could increase it (happens under
	 * heavy contention, when the refcount goes to zero in between a load
	 * and a refcount increment of a node in try_get, then back up to
	 * something non-zero, then the refcount increment is done by the other
	 * thread) -- so if the CAS to add the node to the actual list fails,
	 * decrese the refcount and leave the add operation to the next thread
	 * who puts the refcount back to zero (which could be us, hence the
	 * loop).
	 */
	struct freelist_node *head = READ_ONCE(list->head);

	for (;;) {
		WRITE_ONCE(node->next, head);
		atomic_set_release(&node->refs, 1);

		if (!try_cmpxchg_release(&list->head, &head, node)) {
			/*
			 * Hmm, the add failed, but we can only try again when
			 * the refcount goes back to zero.
			 */
			if (atomic_fetch_add_release(REFS_ON_FREELIST - 1, &node->refs) == 1)
				continue;
		}
		return;
	}
}
```

```c
static inline void __memcache_cas_add(struct memcache_node *node, struct memcache_slot *slot)
{
	/*
	 * Since the refcount is zero, and nobody can increase it until it's
	 * zero (except us, and we run only one copy of this method per node at
	 * a time, i.e. the single thread case), then we know we can safely
	 * change the next pointer of the node; however, once the refcount is
	 * back above zero, then other threads could increase it (happens under
	 * heavy contention, when the refcount goes to zero in between a load
	 * and a refcount increment of a node in try_get, then back up to
	 * something non-zero, then the refcount increment is done by the other
	 * thread) -- so if the CAS to add the node to the actual list fails,
	 * decrese the refcount and leave the add operation to the next thread
	 * who puts the refcount back to zero (which could be us, hence the
	 * loop).
	 */
	struct memcache_node *head;

	for (;;) {
		head = READ_ONCE(slot->fs_head);
		smp_rmb();
		WRITE_ONCE(node->next, head);
		atomic_set(&node->refs, 1);
		smp_wmb();

		if (head == memcache_cmpxchg(&slot->fs_head, head, node))
			break;

		/*
		 * Hmm, the add failed, but we can only try again when refcount
		 * goes back to zero (with REFS_IN_MEMCACHE set).
		 */
		if (atomic_add_return(REFS_IN_MEMCACHE - 1, &node->refs) != REFS_IN_MEMCACHE)
			break;
	}
}
```

```c
static inline void __memcache_cas_add(struct memcache_node *node, struct memcache_slot *slot)
{
	/*
	 * Since the refcount is zero, and nobody can increase it until it's
	 * zero (except us, and we run only one copy of this method per node at
	 * a time, i.e. the single thread case), then we know we can safely
	 * change the next pointer of the node; however, once the refcount is
	 * back above zero, then other threads could increase it (happens under
	 * heavy contention, when the refcount goes to zero in between a load
	 * and a refcount increment of a node in try_get, then back up to
	 * something non-zero, then the refcount increment is done by the other
	 * thread) -- so if the CAS to add the node to the actual list fails,
	 * decrese the refcount and leave the add operation to the next thread
	 * who puts the refcount back to zero (which could be us, hence the
	 * loop).
	 */
	struct memcache_node *head;

	for (;;) {
		head = READ_ONCE(slot->fs_head);
		WRITE_ONCE(node->next, head);
		smp_wmb();
		atomic_set(&node->refs, 1);

		if (head == memcache_cmpxchg(&slot->fs_head, head, node))
			break;

		/*
		 * Hmm, the add failed, but we can only try again when refcount
		 * goes back to zero (with REFS_IN_MEMCACHE set).
		 */
		if (atomic_add_return(REFS_IN_MEMCACHE - 1, &node->refs) != REFS_IN_MEMCACHE)
			break;
	}
}
```

kretprobe/rethook另一种优化思路

从内存占用角度：Per-task stack vs. Global pool (percpu or unqiue ...)

Per-task stack (function-graph)

- ❑ Allocate stack page(s) for each task (thread)
- ❑ Simple array of the saved entries

Pros

- ❑ Simple and fast
- ❑ Scalable (in performance)

Cons

- ❑ Consume memory even if the task is not involved

Global pool (rethook)

- ❑ Allocate fixed number of entries in
- ❑ system-wide pool.
- ❑ Make a linked list for each task

Pros

- ❑ Object size is controllable.
- ❑ Usually smaller memory consumption

Cons

- ❑ User needs to tune the number of objects to avoid **miss-hit**
- ❑ Consuming memory if many objects selected
- ❑ Not scalable (in performance), resolved by objpool (from v6.7)

?

THANKS

ByteDance 字节跳动

CHINA LINUX KERNEL
中国内核开发者大会