ACPI fast handover for kexec live-update

Fam Zheng <fam.zheng@bytedance.com>

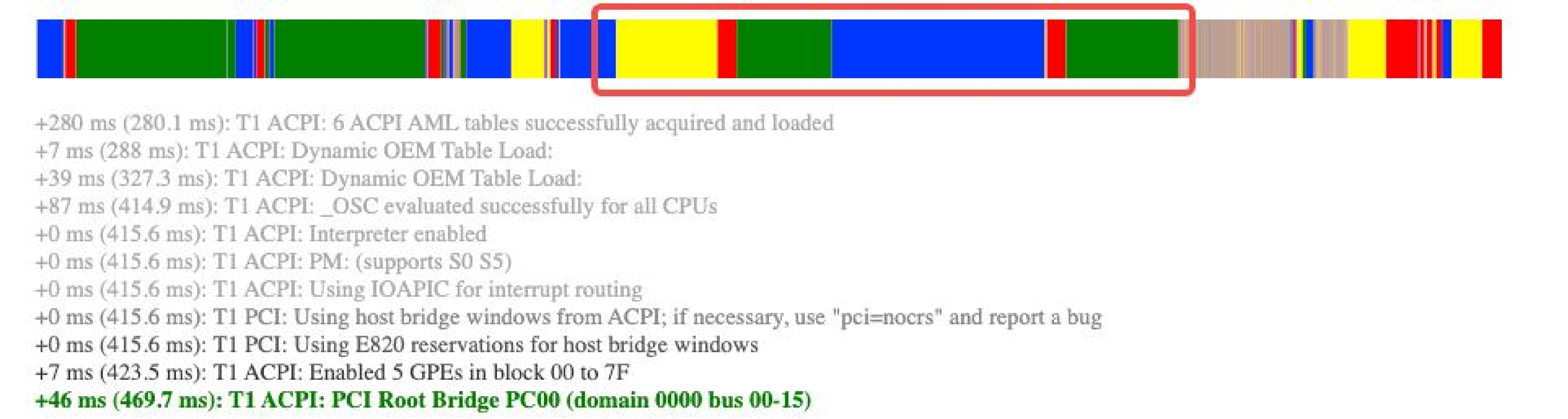
System Technologies and Engineering (STE), ByteDance

Agenda

- Introduction
- ACPI during boot
- Preserving ACPI state across kexec
- Conclusion and future plan

Introduction

- Why care about ACPI in kexec?
  - kexec is very useful to quickly deploy new kernel in production
  - On top of previous optimizations[1], acpi_init() is the next target



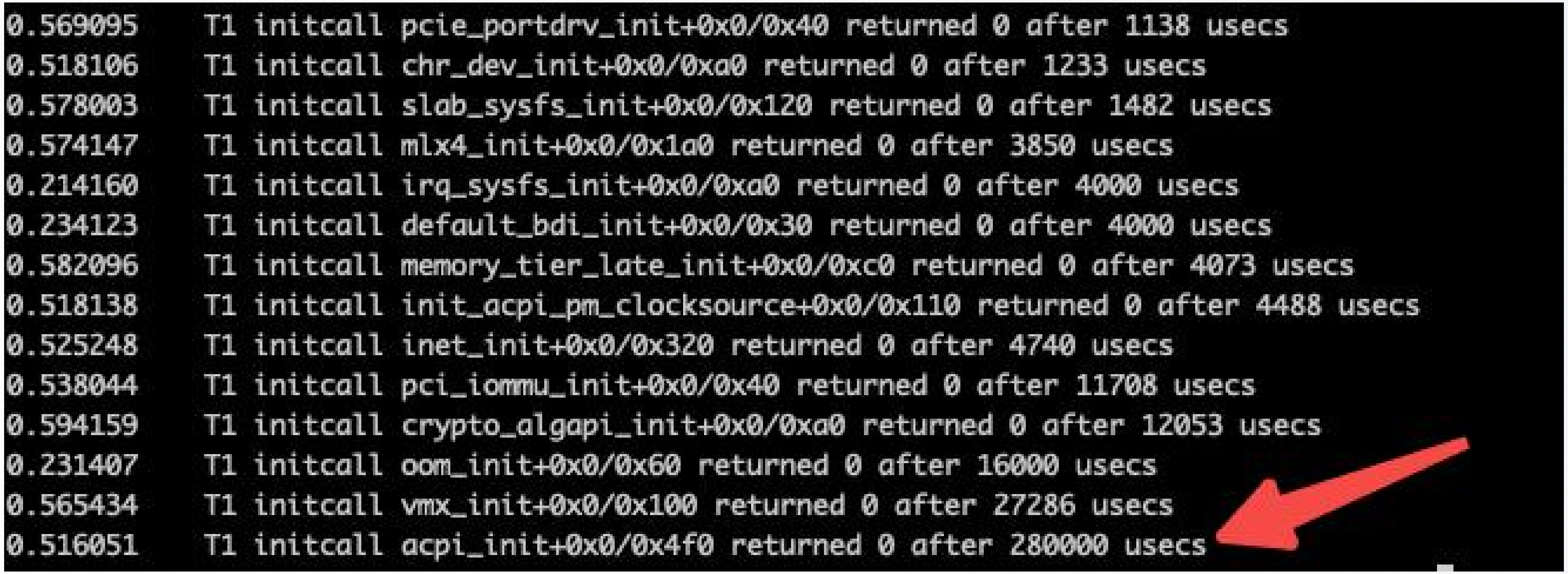
```
+280 ms (280.1 ms): T1 ACPI: 6 ACPI AML tables successfully acquired and loaded
+7 ms (288 ms): T1 ACPI: Dynamic OEM Table Load:
+39 ms (327.3 ms): T1 ACPI: Dynamic OEM Table Load:
+87 ms (414.9 ms): T1 ACPI: _OSC evaluated successfully for all CPUs
+0 ms (415.6 ms): T1 ACPI: Interpreter enabled
+0 ms (415.6 ms): T1 ACPI: PM: (supports S0 S5)
+0 ms (415.6 ms): T1 ACPI: Using IOAPIC for interrupt routing
+0 ms (415.6 ms): T1 PCI: Using host bridge windows from ACPI; if necessary, use "pci=nocrs" and report a bug
+0 ms (415.6 ms): T1 PCI: Using E820 reservations for host bridge windows
+7 ms (423.5 ms): T1 ACPI: Enabled 5 GPEs in block 00 to 7F
+46 ms (469.7 ms): T1 ACPI: PCI Root Bridge PC00 (domain 0000 bus 00-15)
```

The call chain

```
kernel_init
  do_one_initcall
    acpi_init
      acpi_load_tables
      acpi_initialize_objects
      acpi_early_processor_control_setup
      acpi_scan_init
      ...
```

Boot time analysis with initcall_debug=1

- dmesg | grep initcall | sort -n -k 8



```
0.569095    T1 initcall pcie_portdrv_init+0x0/0x40 returned 0 after 1138 usecs
0.518106    T1 initcall chr_dev_init+0x0/0xa0 returned 0 after 1233 usecs
0.578003    T1 initcall slab_sysfs_init+0x0/0x120 returned 0 after 1482 usecs
0.574147    T1 initcall mlx4_init+0x0/0x1a0 returned 0 after 3850 usecs
0.214160    T1 initcall irq_sysfs_init+0x0/0xa0 returned 0 after 4000 usecs
0.234123    T1 initcall default_bdi_init+0x0/0x30 returned 0 after 4000 usecs
0.582096    T1 initcall memory_tier_late_init+0x0/0xc0 returned 0 after 4073 usecs
0.518138    T1 initcall init_acpi_pm_clocksource+0x0/0x110 returned 0 after 4488 usecs
0.525248    T1 initcall inet_init+0x0/0x320 returned 0 after 4740 usecs
0.538044    T1 initcall pci_iommu_init+0x0/0x40 returned 0 after 11708 usecs
0.594159    T1 initcall crypto_algapi_init+0x0/0xa0 returned 0 after 12053 usecs
0.231407    T1 initcall oom_init+0x0/0x60 returned 0 after 16000 usecs
0.565434    T1 initcall vmx_init+0x0/0x100 returned 0 after 27286 usecs
0.516051    T1 initcall acpi_init+0x0/0x4f0 returned 0 after 280000 usecs
```

Boot time analysis with function_graph

- ftrace=function_graph ftrace_graph_max_depth=2 ftrace_graph_filter=acpi_init
    - 0) @ 223895.8 us | acpi_load_tables();
    - 0) @ 895929.9 us | acpi_early_processor_control_setup();
    - 0) @ 426643.9 us | acpi_scan_init();
- *Note these numbers are with significant tracing overhead*

Breaking down [1/3]

- acpi_load_tables()
  - 0) + 84.666 us | acpi_ev_install_region_handlers();
  - **0) @ 198017.2 us | acpi_tb_load_namespace();**
  - 0) * 14992.95 us | acpi_ns_initialize_objects();
- acpi_ev_install_region_handlers
  - Keep as-is
- **acpi_tb_load_namespace()**
  - **Mostly parsing tables**
  - **Focus of optimization**
- acpi_ns_initialize_objects()
  - Relatively cheap, skip for now

Breaking down [2/3]

- acpi_early_processor_control_setup
  - Calls _OSC or _PDC
  - Used to report to platform processor capability bits

- TBD: Is this safe/possible to bypass during kexec?

Breaking down [3/3]

- acpi_scan_init

- This does most of the device driver init

- TBD: can we safely use ACPI_NO_DEVICE_INIT if kexec?

Adding save/restore to acpi_load_tables

- To add acpi=restore mode
  - bypass acpi_tb_load_namespace()
  - “restore” from a preserved memory location

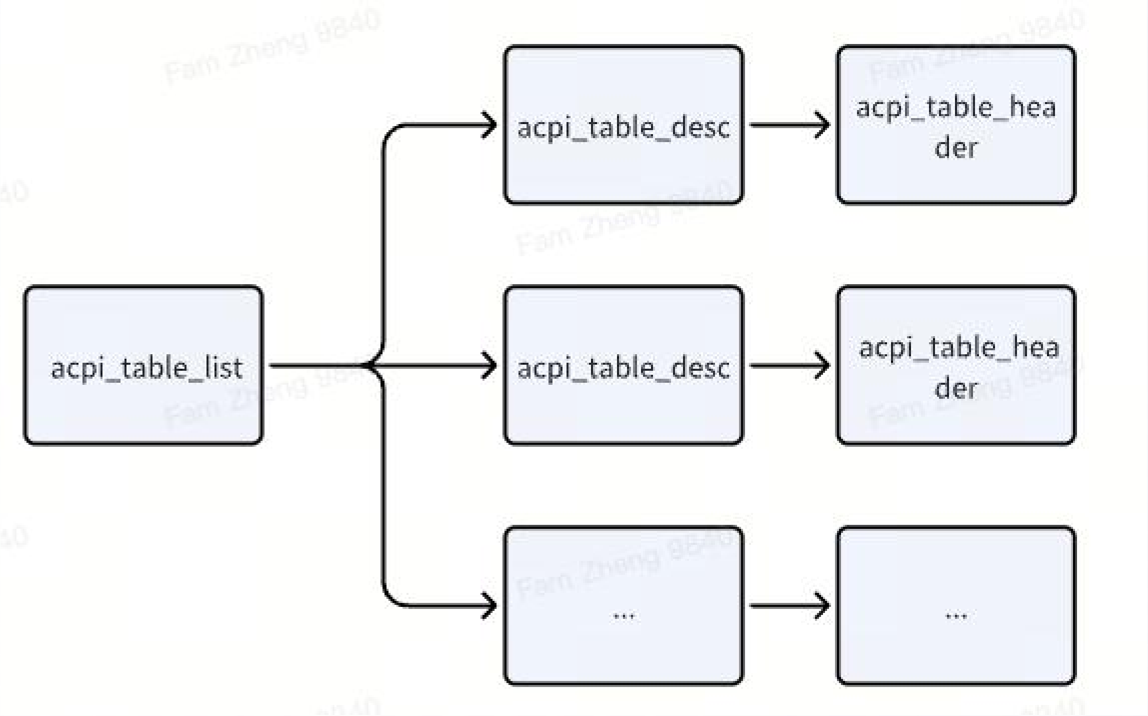
```
if (acpi==restore)
   acpi_restore_namespace(); /* fast path */
else
   acpi_tb_load_namespace(); /* slow path */
```

- ... which depends on:
  - A way to predictably reserve memory for save/restore
  - Simple and fast, to avoid expensive / complex setup at boot time

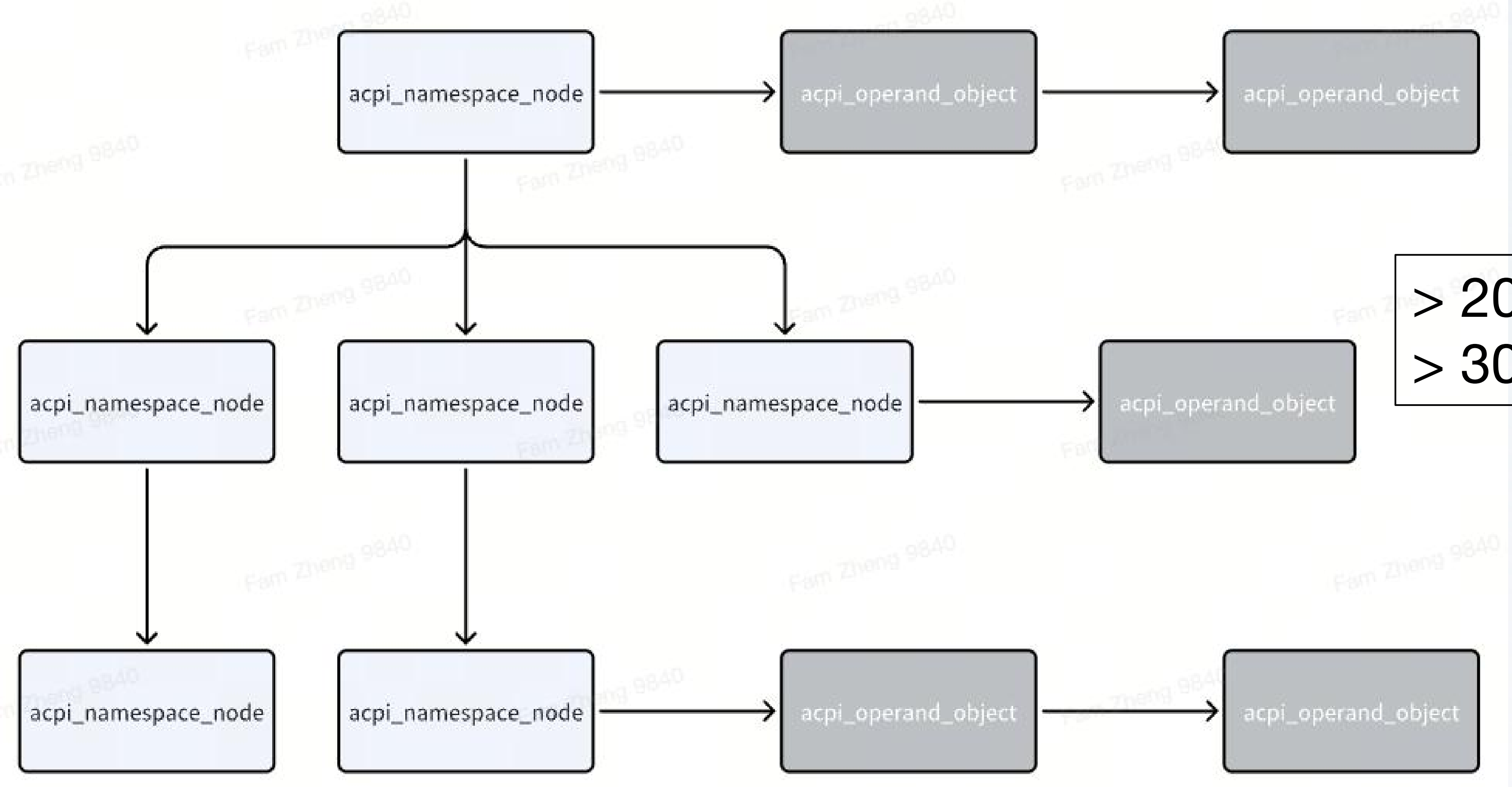
What to save/restore

- Conveniently collected in drivers/acpi/acpica/acglobal.h
  - Derived from https://github.com/acpica/acpica/blob/master/source/include/acglobal.h
- Tables
  - acpi_gbl_DSDT, acpi_gbl_original_dsdt_header, acpi_gbl_dsdt_index, ...
- Namespace
  - struct acpi_namespace_node acpi_gbl_root_node_struct
  - struct acpi_namespace_node acpi_gbl_root_node

ACPI runtime state



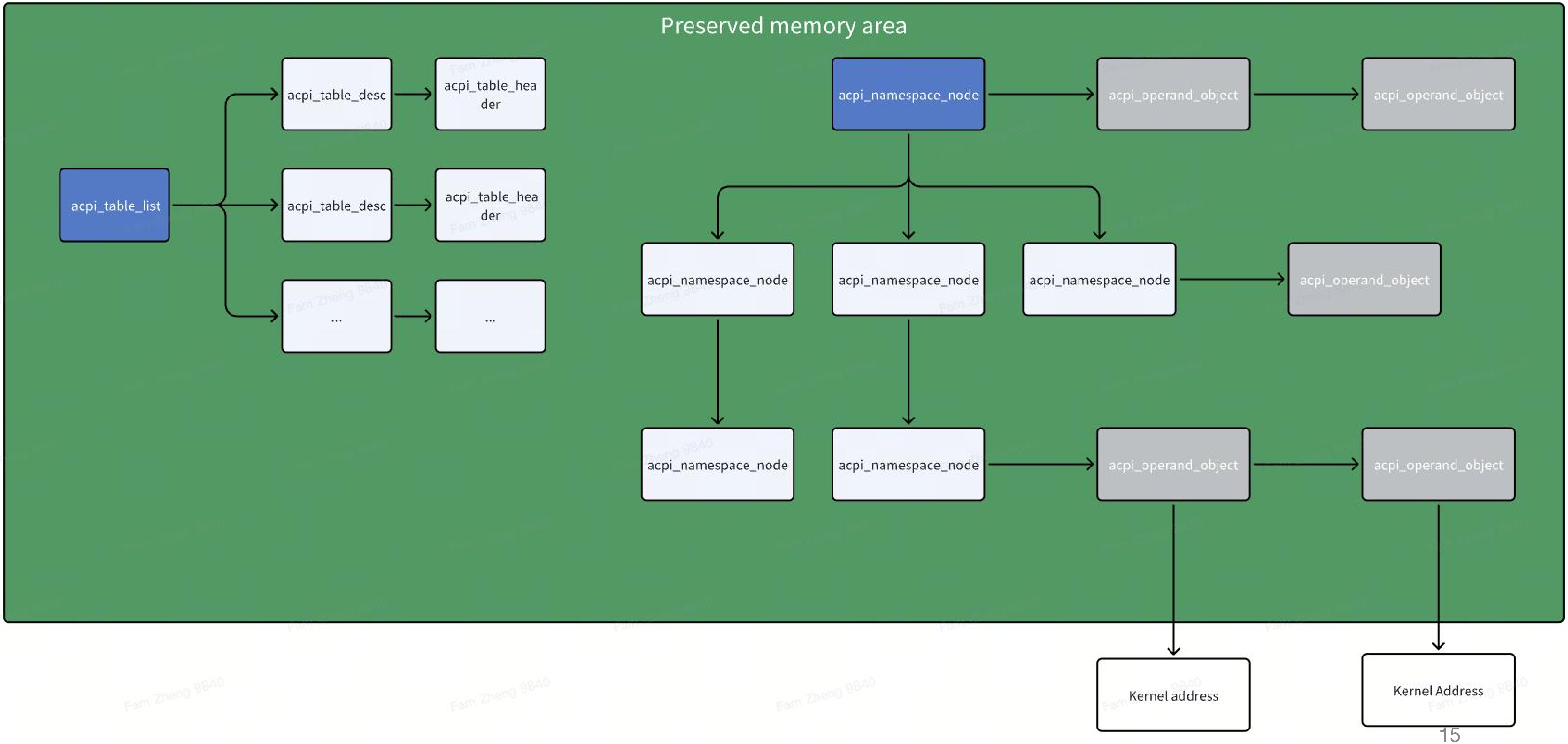
ACPI runtime state



> 20k nodes
> 300k objects

Preserving state across kexec

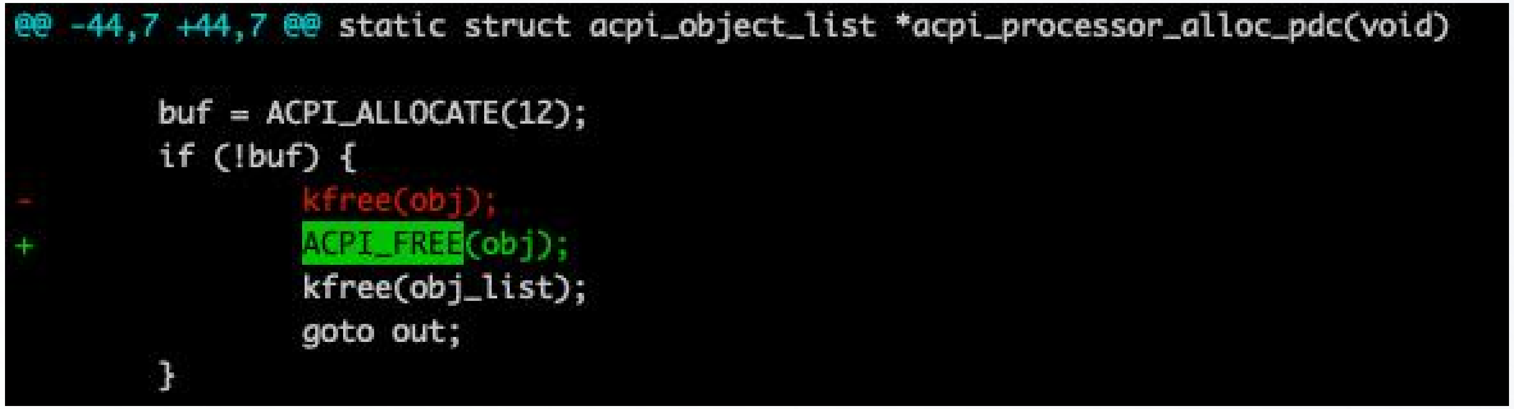
- Allocate ACPI objects in preserved memory areas
- Save/load root objects during reboot


Preserved memory area
acpi_table_list
acpi_table_desc
acpi_table_header
acpi_table_desc
acpi_table_header
...
...
acpi_namespace_node
acpi_operand_object
acpi_operand_object
acpi_namespace_node
acpi_namespace_node
acpi_namespace_node
acpi_operand_object
acpi_namespace_node
acpi_namespace_node
acpi_operand_object
acpi_operand_object
Kernel address
Kernel Address

Patching ACPI_ALLOCATE() & ACPI_FREE()

- Replace acpi_os_allocate / acpi_os_free to use a “preserved” allocator
  - All pointers returned remain “valid” after kexec

- Also, fix 160+ mismatched kfree() pairs, e.g.:

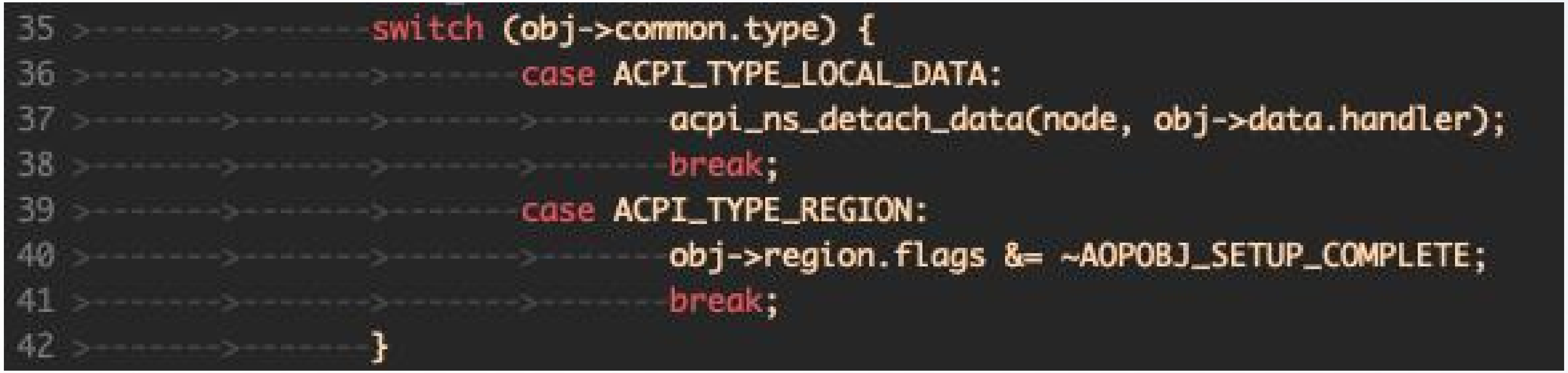


```diff
@@ -44,7 +44,7 @@ static struct acpi_object_list *acpi_processor_alloc_pdc(void)

 	buf = ACPI_ALLOCATE(12);
 	if (!buf) {
-		kfree(obj);
+		ACPI_FREE(obj);
 		kfree(obj_list);
 		goto out;
 	}
```

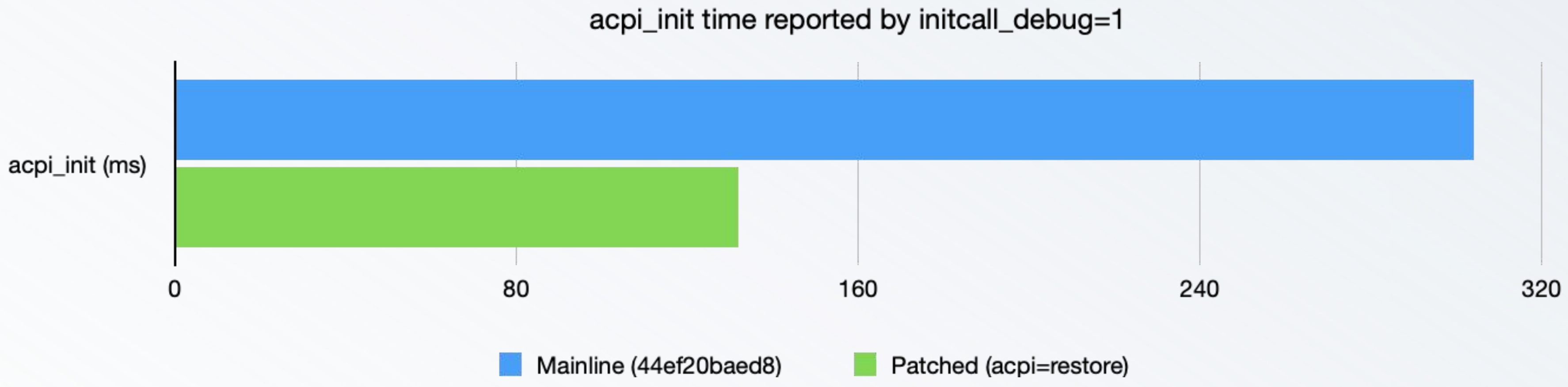
Restoring

- Apart from picking up the root node from last kernel, we need to setup some runtime state
- Mostly reuse acpi_bus_scan() etc.
- But must do some “fixup” first to “stale” nodes, e.g.:



```
		switch (obj->common.type) {
			case ACPI_TYPE_LOCAL_DATA:
				acpi_ns_detach_data(node, obj->data.handler);
				break;
			case ACPI_TYPE_REGION:
				obj->region.flags &= ~AOPOBJ_SETUP_COMPLETE;
				break;
		}
```

Conclusion



- The hacked acpi_init is significantly faster
- Changeset is reasonably small and non-intrusive
- More work needed to cover different cases/platforms
- Debugging is a bit tricky on baremetal

Future plans

- A more dynamic KRAM design, or a different approach
  - Or at least a more dynamic obj cache

- ACPI correctness
  - Ideas are welcome on verifier / sanitizer
  - Can KASAN help?

- Look into integrating Agraf’s Kexec HandOver (KHO)
  - https://lore.kernel.org/lkml/20240117144704.602-18-graf@amazon.com/T/

Related work

- [1] Improving kexec boot time, Usama Arif https://lpc.events/event/17/contributions/1512/
- [2] Parallel SMP boot, David Woodhouse & Usama Arif https://lwn.net/Articles/924933/
- [3] Kexec Handover, Alex Graf https://lwn.net/Articles/924933/
- [4] Preserving IOMMU States During Kexec https://kvm-forum.qemu.org/2022/
- [5] QEMU Live Update, Steve Sistare https://blogs.oracle.com/linux/post/qemu-live-update

Thank you!

- Questions?

- Contact: Fam Zheng <fam.zheng@bytedance.com>