The Rise and Fall of Assembler and the VGIC from Hell

Marc Zyngier, ARM
Christoffer Dall, Linaro

KVM/ARM:

The rise and fall of assembly code and the VGIC from hell

ARM

Christoffer Dall <christoffer.dall@linaro.org>
Marc Zyngier <marc.zyngier@arm.com>

LCU16

September 14, 2016

KVM/ARM: Absolute Beginners

KVM/ARM, merged in v3.9

- `38 files changed, 6546 insertions(+), 20 deletions(-)`
- Not bad, for a start. But wait…
- `13 files changed, 2060 insertions(+), 12 deletions(-)`
- That's the vgic…
- `12 files changed, 489 insertions(+), 1 deletion(-)`
- And here's the timer

Over 9k LoC, with about 10% assembly code implementing the world switch (mostly).

KVM/arm64: Always Crashing In The Same Car

KVM/arm64, merged in v3.11

- The arm64 port didn't change this fine tradition
  - World switch in `asm`, the rest in C.
- Not changing the structure made it easy to build on the initial work
  - When you have something that works, it is tempting not to reinvent the wheel...
- The result: about 3400 lines of new code
- About 1000 lines of assembly code

EL2: What In The World

What does this assembly code do?

- It swaps two execution environments (between host and guest)
  - GPRs
  - FPSIMD
  - All system registers (including virtual memory)
  - Interrupt context
- Handle all exceptions happening whilst a guest runs
  - Interrupts
  - Page faults
  - Paravirtualized services
- Offers an small set of services to the host too
  - TLB invalidation, please run this guest...

Generally known as “the World Switch”.

World switch: Under Pressure

- Initial code is fairly straightforward
- Things become quickly more complicated
  - GICv3 support
  - Lazy FPSIMD
  - Debug support
- Interaction between various code paths are not obvious
- Register allocation gets a bit hairy
  - Try throwing 31 balls in the air...
  - ... and keep track of their individual positions...
  - ... before catching them
- Maintainers are feeling the pressure...
  - Optimizing is hard makes the code more fragile
  - Bugs are hard to squash

EL2: Life on Mars

Let's take a step back: why using assembly code:

- HYP/EL2 is a separate exception level
  - Its own exceptions, its own page tables
  - Its own rules too...
  - Its VA space is at an offset from the kernel VA
- Not the usual warm, cosy kernel environment
  - No printk, no tracing, no debug facility (omg, no printk!!)
  - More akin being stranded on an iceberg. Naked.
- Easier to write a standalone piece of code
  - Exception boundaries are well understood
  - Creates clear delimitations between kernel and HYP spaces

VHE: Breaking Glass

And then comes the feature that breaks everything: VHE.

- VHE allows the kernel to run at EL2 on ARMv8.1 systems
  - Does so by aliasing `_EL2` registers to their `_EL1` counterpart
  - The kernel runs unmodified
  - The hypervisor needs to be heavily modified
- But making the world switch code VHE compliant is ... interesting.
- Entirely relies on code patching
  - Tons of system register renaming
  - Alternate sequences for some paths
  - See presentation at LCA15
- The result, although functionnal, is not easily maintainable
- Optimizing becomes extremely hard, wasting the VHE effort

Maybe it is time to reconsider how the world-switch code is architected.

WSinC: Changes

So what is actually required to use C at EL2? Surprizingly little:

- Have a valid stack
- Respect the AArch64 PCS (IHI 0055B)
- Map the read-only data into EL2

And a few more things that are Linux specific:

- Put the code ends up in a separate section
- Do NOT call any kernel function from the HYP code
  - Unless you can guarantee they are inlined
  - Remember the bit about having a different VA space?
- Make sure nothing gets traced or instrumented

WSinC: Shape of Things

There is still some bits of assembly code that are required:

- Calling into HYP
  - Marshalling the parameters across HVC
  - Just a function call for VHE
- Entering the guest
  - Seen as a normal function from C code (`__guest_enter`)
  - Performs GPR save/restore
- Taking exceptions (interrupt, hypercall, fault)
  - Save a very minimal context
  - Seen by the C code as `__guest_enter` returning

Everything else is written as C code.

WSinC: Get Real

So what?

- A couple of weeks spent hacking the kernel
  - That's what holidays are for!
- `28 files changed, 1532 insertions(+), 1612 deletions(-)`
  - Yes, we actually removed a bit of code
  - Very few bugs (the compiler catches the silly stuff early)
- The result is slightly faster, despite not being optimized
  - Turns out CPUs are optimized for compiled code...
- `32 files changed, 862 insertions(+), 377 deletions(-)`
  - And then comes VHE
- And then we can start optimizing, because it's easy!
  - Up to 40% reduction in interrupt latency
  - VHE-specific optimizations on the way
- We can now share some the HYP code with the 32bit port

Thank you!

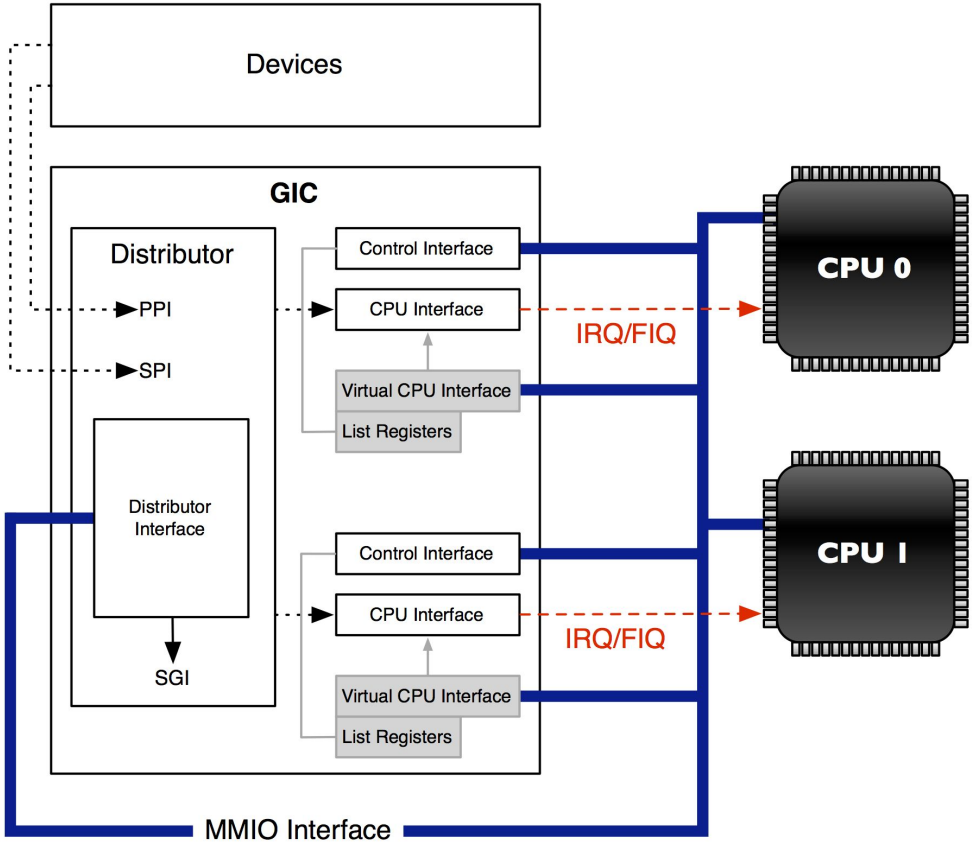
ARM

Linaro connect
Las Vegas 2016

ENGINEERS
AND DEVICES
WORKING
TOGETHER

The VGIC from Hell

Generic Interrupt Controller (GIC) - simplified view



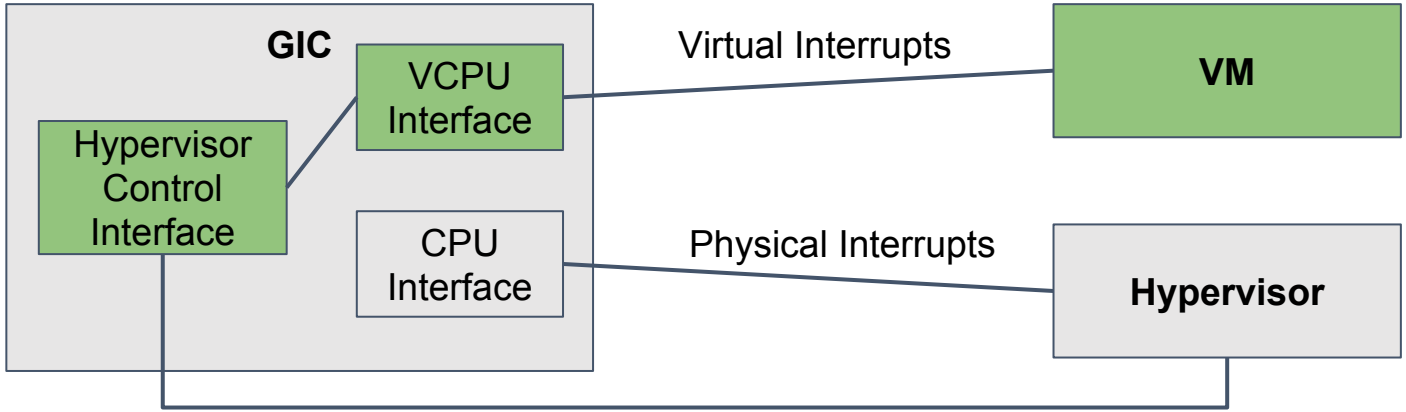
The Gist of the GIC

- Devices signal interrupts to the GIC
- CPUs can receive interrupts (ACK) and complete interrupts (EOI)
- CPUs can configure the GIC:
  - CPU affinity
  - IRQ priority
  - Level vs. Edge trigger
  - Enable/Disable IRQs
  - ...and more scary stuff
- CPUs can ask the GIC to interrupt other CPUs (IPIs)

The V in VGIC

- **V**irtualization Extensions (Hardware Virtualization Support)
- Provides a **virtual CPU interface** that the VM can interact with directly
- Provides a **hypervisor control interface** to deliver virtual interrupts
- Benefit: No traps on ACK/EOI

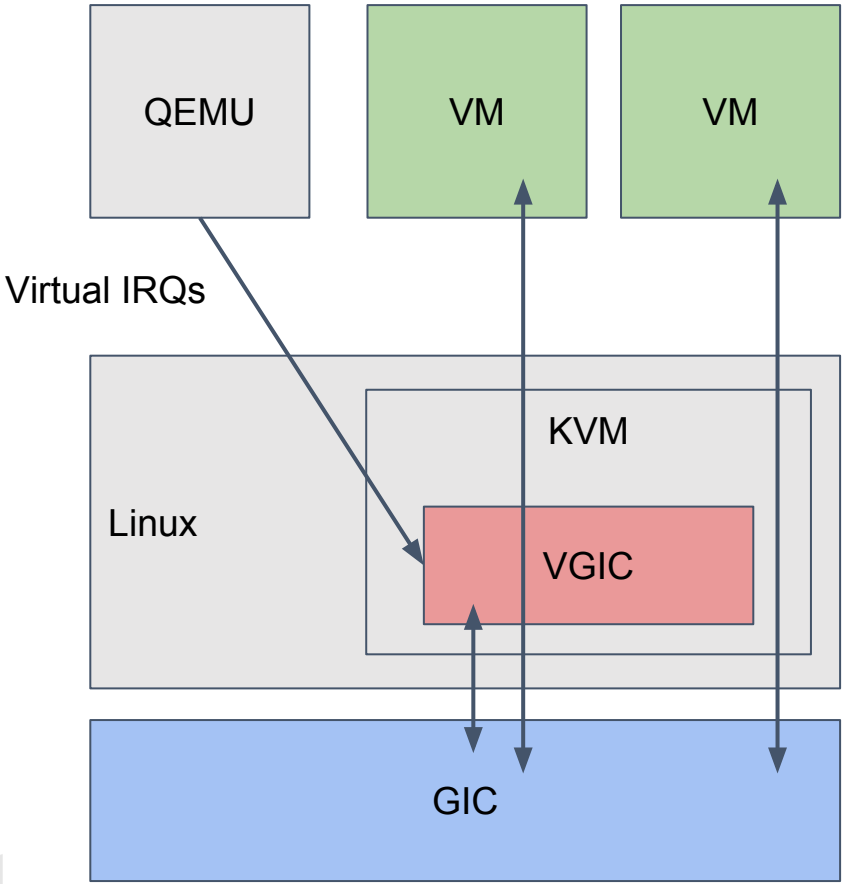
  Hardware takes care of priorities, masking, etc.



The Software Problem

- The GIC is split in two:
  - Distributor (configuration side)
  - CPU Interface (delivery side)
- No virtualization support for the distributor
- Must fully emulate distributor in software
- Emulated distributor drives delivery of virtual interrupts

Software Architecture



Software Challenges

- Lots of state
  - Each IRQ has: enabled/disabled, priority, active, pending, soft_pending, affinity, and more...
  - Global state: enable/disable
  - Per-vcpu state: List Registers (LRs) in Hypervisor Control Interface
  - ...all of this is per-VM.
- Lots of transitions:
  - Userspace and vhost can make virtual IRQ lines go up and down
  - Virtual CPUs can make interrupts pending (IPIs)
  - Virtual CPUs can modify other individual IRQ state (e.g. affinity)
  - Hardware can change state without notifying software (GIC Virtualization Extensions)
- Everything happens asynchronously

The old VGIC

- ...was a mess, because
- Maintained per-IRQ state as global state based on many large bitmaps
- Made it possible to compute global state quickly
- Duplicated pre-computed distributor and VCPU state
- Level-triggered interrupts were shoe-horned into design

Symptoms:

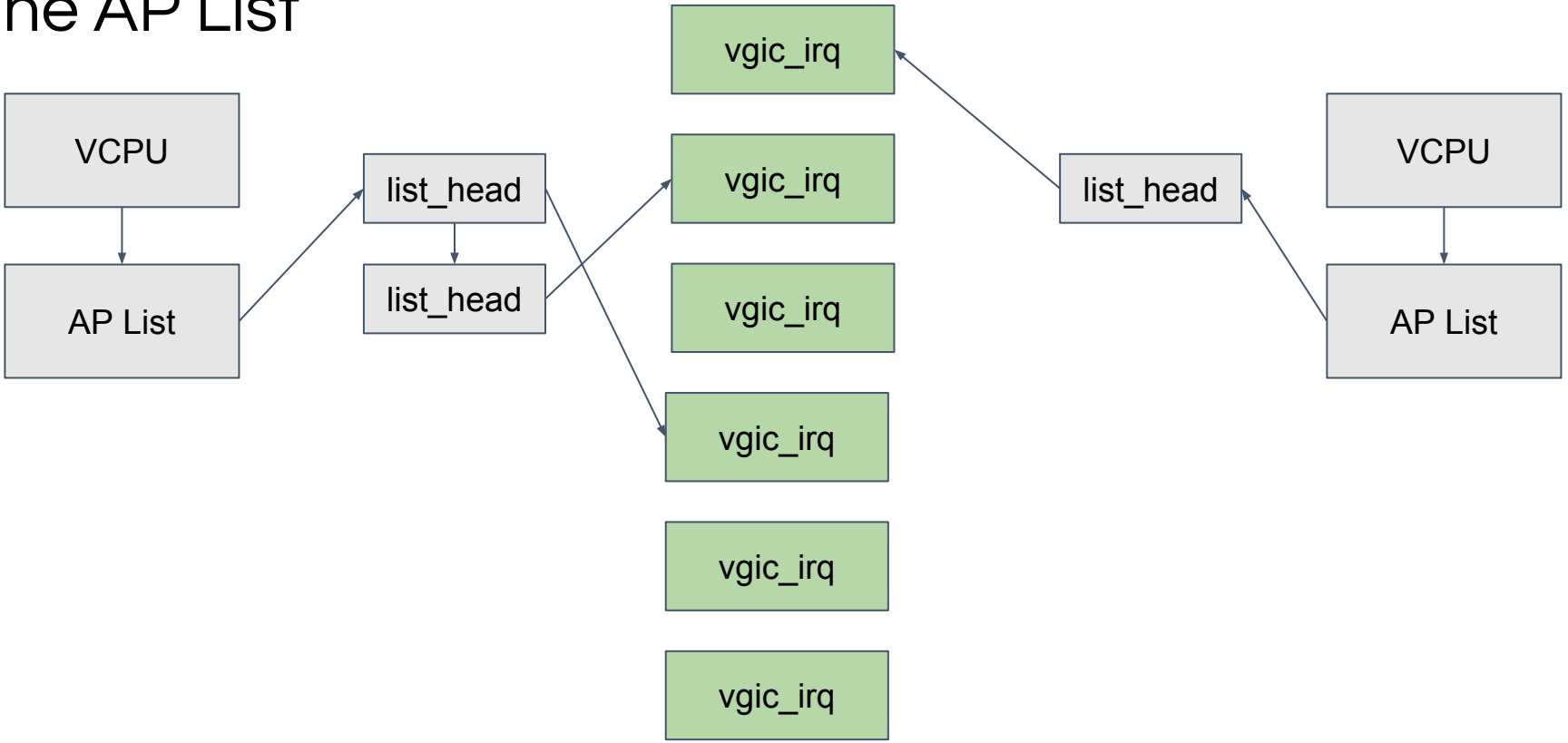
- Made it very hard to ensure consistent state
- Required global lock on almost every operation (measurable!)
- Unintuitive code; calculate bit-positions to modify a boolean state
- Drove maintainers to point of insanity

The New VGIC

- Was designed during a Linaro mini-sprint
- Covers GICv2, GICv3, and data structures for the ITS
- Key insight #1:
  - Most of the time, there are no IRQs in flight
- Key insight #2:
  - MMIO operations are rare, and not in the critical path
- The basic idea:

```
struct vgic_irq {
    int intid;
    struct list_head ap_list;
    bool pending;
    ...
};
```

The AP List


VCPU
AP List
list_head
list_head
vgic_irq
vgic_irq
vgic_irq
vgic_irq
vgic_irq
vgic_irq
list_head
VCPU
AP List
...

Locking in the new world

- Historical data has shown we need more fine-grained locking than a per-VM lock.
- Locking scheme becomes:
  - One lock per struct vgic_irq to ensure consistency
  - One lock per AP list
  - Only the VCPU thread itself may remove IRQs from its AP list
- Sometimes you need to grab more than one lock
  - Solution: Define strict locking order
- Locking order:
  - AP List lock
    - IRQ lock
  - Lowest-numbered VCPU's AP list lock first
- Documented in virt/kvm/arm/vgic/vgic.c

Worst Locking Example: Reassign pending IRQ

1. VCPU 3 takes its AP List lock
2. VCPU 3 takes the IRQ lock
3. VCPU 3 concludes that this interrupt is still pending and must now be handled by VCPU 1
4. VCPU 3 releases the IRQ lock
5. VCPU 3 releases the AP List lock
6. VCPU 3 takes the VCPU 1 AP List Lock
7. VCPU 3 takes its own AP List lock
8. VCPU 3 takes the IRQ lock
9. Re-check all conditions for the reassignment
10. Carry out reassignment if all conditions are still met
11. Release all locks in reverse order

Status of the new VGIC

- Merged in v4.7
- Roughly 600 fewer lines of code
- Pretty stable since the merge
- Improved world switch performance
- Happier maintainers
- Huge thanks to: Andre Przywara, Eric Auger, Peter Maydell, Alex Bennée

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Where this leaves us

- KVM/ARM is in really good shape!
- Highlighted new'ish features:
  - Virtual GICv3
  - Virtual ITS
  - VHOST with virtual MSIs and virtual ITS
  - VHE support on ARMv8.1
  - Reduced world-switch time
- In the pipeline:
  - GICv3 save/restore
  - ITS save/restore
  - PCIe with MSI passthrough
  - Cross CPU-Type Support (migration in heterogeneous datacenters)
  - Optimizations
  - GICv4 (direct virtual interrupt injection)

Thank You

#LAS16

For further information: www.linaro.org

LAS16 keynotes and videos on: connect.linaro.org