The PPC KVM paravirtual interface

The basic execution principle by which KVM on PowerPC works is to run all kernel space code in PR=1 which is user space. This way we trap all privileged instructions and can emulate them accordingly.

Unfortunately that is also the downfall. There are quite some privileged instructions that needlessly return us to the hypervisor even though they could be handled differently.

This is what the PPC PV interface helps with. It takes privileged instructions and transforms them into unprivileged ones with some help from the hypervisor. This cuts down virtualization costs by about 50% on some of my benchmarks.

The code for that interface can be found in arch/powerpc/kernel/kvm*

Querying for existence

To find out if we're running on KVM or not, we leverage the device tree. When Linux is running on KVM, a node /hypervisor exists. That node contains a compatible property with the value "linux,kvm".

Once you determined you're running under a PV capable KVM, you can now use hypercalls as described below.

KVM hypercalls

Inside the device tree's /hypervisor node there's a property called 'hypercall-instructions'. This property contains at most 4 opcodes that make up the hypercall. To call a hypercall, just call these instructions.

The parameters are as follows:

Register	IN	OUT
r0	•	volatile
r3	1st parameter	Return code
r4	2nd parameter	1st output value
r5	3rd parameter	2nd output value
r6	4th parameter	3rd output value
r7	5th parameter	4th output value
r8	6th parameter	5th output value
r9	7th parameter	6th output value
r10	8th parameter	7th output value
r11	hypercall number	8th output value
r12	•	volatile

Hypercall definitions are shared in generic code, so the same hypercall numbers apply for x86 and powerpc alike with the exception that each KVM hypercall also needs to be ORed with the KVM vendor code which is (42 << 16).

Return codes can be as follows:

Code	Meaning
0	Success
12	Hypercall not implemented
<0	Error

The magic page

To enable communication between the hypervisor and guest there is a new shared page that contains parts of supervisor visible register state. The guest can map this shared page using the KVM hypercall KVM_HC_PPC_MAP_MAGIC_PAGE.

With this hypercall issued the guest always gets the magic page mapped at the desired location. The first parameter indicates the effective address when the MMU is enabled. The second parameter indicates the address in real mode, if applicable to the target. For now, we always map the page to -4096. This way we can access it using absolute load and store functions. The following instruction reads the first field of the magic page:

```
ld      rX, -4096(0)
```

The interface is designed to be extensible should there be need later to add additional registers to the magic page. If you add fields to the magic page, also define a new hypercall feature to indicate that the host can give you more registers. Only if the host supports the additional features, make use of them.

The magic page layout is described by struct kvm_vcpu_arch_shared in arch/powerpc/include/asm/kvm_para.h.

Magic page features

When mapping the magic page using the KVM hypercall KVM_HC_PPC_MAP_MAGIC_PAGE, a second return value is passed to the guest. This second return value contains a bitmap of available features inside the magic page.

The following enhancements to the magic page are currently available:

KVM_MAGIC_FEAT_SR	Maps SR registers r/w in the magic page
KVM_MAGIC_FEAT_MAS0_TO_SPRG7	Maps MASn, ESR, PIR and high SPRGs

For enhanced features in the magic page, please check for the existence of the feature before using them!

Magic page flags

In addition to features that indicate whether a host is capable of a particular feature we also have a channel for a guest to tell the guest whether it's capable of something. This is what we call "flags".

Flags are passed to the host in the low 12 bits of the Effective Address.

The following flags are currently available for a guest to expose:

MAGIC_PAGE_FLAG_NOT_MAPPED_NX Guest handles NX bits correctly wrt magic page

MSR bits

The MSR contains bits that require hypervisor intervention and bits that do not require direct hypervisor intervention because they only get interpreted when entering the guest or don't have any impact on the hypervisor's behavior.

The following bits are safe to be set inside the guest:

- MSR_EE
- MSR_RI

If any other bit changes in the MSR, please still use mtmsr(d).

Patched instructions

The "ld" and "std" instructions are transformed to "lwz" and "stw" instructions respectively on 32 bit systems with an added offset of 4 to accommodate for big endianness.

The following is a list of mapping the Linux kernel performs when running as guest. Implementing any of those mappings is optional, as the instruction traps also act on the shared page. So calling privileged instructions still works as before.

From	To
mfmsr rX	ld rX, magic_page->msr
mfsprg rX, 0	ld rX, magic_page->sprg0
mfsprg rX, 1	ld rX, magic_page->sprg1
mfsprg rX, 2	ld rX, magic_page->sprg2
mfsprg rX, 3	ld rX, magic_page->sprg3
mfsrr0 rX	ld rX, magic_page->srr0
mfsrr1 rX	ld rX, magic_page->srr1
mfdar rX	ld rX, magic_page->dar
mfdsisr rX	lwz rX, magic_page->dsisr
mtmsr rX	std rX, magic_page->msr
mtsprg 0, rX	std rX, magic_page->sprg0
mtsprg 1, rX	std rX, magic_page->sprg1
mtsprg 2, rX	std rX, magic_page->sprg2
mtsprg 3, rX	std rX, magic_page->sprg3
mtsrr0 rX	std rX, magic_page->srr0
mtsrr1 rX	std rX, magic_page->srr1
mtdar rX	std rX, magic_page->dar
mtdsisr rX	stw rX, magic_page->dsisr

From	To
tlbsync	nop
mtmsrd rX, 0	b <special mtmsr section>
mtmsr rX	b <special mtmsr section>
mtmsrd rX, 1	b <special mtmsrd section>
[Book3S only]	
mtsrin rX, rY	b <special mtsrin section>
[BookE only]	
wrteei [0|1]	b <special wrteei section>

Some instructions require more logic to determine what's going on than a load or store instruction can deliver. To enable patching of those, we keep some RAM around where we can live translate instructions to. What happens is the following:

1. copy emulation code to memory
2. patch that code to fit the emulated instruction
3. patch that code to return to the original pc + 4
4. patch the original instruction to branch to the new code

That way we can inject an arbitrary amount of code as replacement for a single instruction. This allows us to check for pending interrupts when setting EE=1 for example.

Hypercall ABIs in KVM on PowerPC

1. KVM hypercalls (ePAPR)

These are ePAPR compliant hypercall implementation (mentioned above). Even generic hypercalls are implemented here, like the ePAPR idle hcall. These are available on all targets.

2. PAPR hypercalls

PAPR hypercalls are needed to run server PowerPC PAPR guests (-M pseries in QEMU). These are the same hypercalls that pHyp, the POWER hypervisor implements. Some of them are handled in the kernel, some are handled in user space. This is only available on book3s_64.

3. OSI hypercalls

Mac-on-Linux is another user of KVM on PowerPC, which has its own hypercall (long before KVM). This is supported to maintain compatibility. All these hypercalls get forwarded to user space. This is only useful on book3s_32, but can be used with book3s_64 as well.