Xen on ARM vITS Handling

Ian Campbell ian.campbell@citrix.com

Draft A

Contents

1 Introduction

ARM systems containing a GIC version 3 or later may contain one or more ITS logical blocks. An ITS is used to route Message Signalled interrupts from devices into an LPI injection on the processor.

The following summarises the ITS hardware design and serves as a set of assumptions for the vITS software design. (XXX it is entirely possible I've

horribly misunderstood how this stuff fits together). For full details of the ITS see the "GIC Architecture Specification".

Message signalled interrupts are translated into an LPI via a translation table which must be configured for each device which can generate an MSI. The ITS uses the device id of the originating device to lookup the corresponding translation table. Devices IDs are typically described via system firmware, e.g. the ACPI IORT table or via device tree.

The ITS is configured and managed, including establishing a Translation Table for each device, via an in memory ring shared between the CPU and the ITS controller. The ring is managed via the `GITS_CBASER` register and indexed by `GITS_CWRITER` and `GITS_CREADR` registers.

A processor adds commands to the shared ring and then updates `GITS_CWRITER` to make them visible to the ITS controller.

The ITS controller processes commands from the ring and then updates `GITS_CREADR` to indicate the the processor that the command has been processed.

Commands are processed sequentially.

Commands sent on the ring include operational commands:

- Routing interrupts to processors;
- Generating interrupts;
- Clearing the pending state of interrupts;
- Synchronising the command queue

and maintenance commands:

- Map device/collection/processor;
- Map virtual interrupt;
- Clean interrupts;
- Discard interrupts;

The ITS provides no specific completion notification mechanism. Completion is monitored by a combination of a `SYNC` command and either polling `GITS_CREADR` or notification via an interrupt generated via the `INT` command.

Note that the interrupt generation via `INT` requires an originating device ID to be supplied (which is then translated via the ITS into an LPI). No specific device ID is defined for this purpose and so the OS software is expected to fabricate one.

Possible ways of inventing such a device ID are:

- Enumerate all device ids in the system and pick another one;
- Use a PCI BDF associated with a non-existent device function (such as an unused one relating to the PCI root-bridge) and translate that (via firmware tables) into a suitable device id;
- ???

2 vITS

A guest domain which is allowed to use ITS functionality (i.e. has been assigned pass-through devices which can generate MSIs) will be presented with a virtualised ITS.

Accesses to the vITS registers will trap to Xen and be emulated and a virtualised Command Queue will be provided.

Commands entered onto the virtual Command Queue will be translated into physical commands (this translation is described in the GIC specification).

XXX there are other aspects to virtualising the ITS (LPI collection management, assignment of LPI ranges to guests). However these are not currently considered here. XXX Should they be/do they need to be?

2.1 Requirements

Emulation should not block in the hypervisor for extended periods. In particular Xen should not busy wait on the physical ITS. Doing so blocks the physical CPU from doing anything else (such as scheduling other VCPUS)

There may be multiple guests which have a vITS, all targeting the same underlying pITS. A single guest VCPU should not be able to monopolise the pITS via its vITS and all guests should be able to make forward progress.

2.2 Command Queue Virtualisation

The command queue of each vITS is represented by a data structure:

```
struct vits_cq {
    list_head schedule_list; /* Queued onto pits.schedule_list */
    uint32_t creadr;         /* Virtual creadr */
    uint32_t cwriter;        /* Virtual cwriter */
    uint32_t progress;       /* Index of last command queued to pits */
    [ Reference to command queue memory ]
};
```

Each pITS has an associated data structure:

```
struct pits {
    list_head schedule_list; /* Contains list of vitq_cq.schedule_lists */
uint32_t last_creadr;
};
```

On write to the virtual `CWRITER` the cwriter field is updated and if that results in there being new outstanding requests then the vits_cq is enqueued onto pITS' schedule_list (unless it is already there).

On read from the virtual `CREADR` iff the vits_cq is such that commands are outstanding then a scheduling pass is attempted (in order to update `vits_cq.creadr`). The current value of `vitq_cq.creadr` is then returned.

2.2.1 pITS Scheduling

A pITS scheduling pass is attempted:

- On write to any virtual `CWRITER` iff that write results in there being new outstanding requests for that vits;
- On read from a virtual `CREADR` iff there are commands outstanding on that vits;
- On receipt of an interrupt notification arising from Xen's own use of `INT`; (see discussion under Completion)
- On any interrupt injection arising from a guests use of the `INT` command; (XXX perhaps, see discussion under Completion)

Each scheduling pass will:

- Read the physical `CREADR`;
- For each command between `pits.last_creadr` and the new `CREADR` value process completion of that command and update the corresponding `vits_cq.creadr`.
- Attempt to refill the pITS Command Queue (see below).

2.2.2 Filling the pITS Command Queue.

Various algorithms could be used here. For now a simple proposal is to traverse the `pits.schedule_list` starting from where the last refill finished (i.e not from the top of the list each time).

If a `vits_cq` has no pending commands then it is removed from the list.

If a `vits_cq` has some pending commands then `min(pits-free-slots, vits-outstanding, VITS_BATCH_SIZE)` will be taken from the vITS command queue, translated and placed onto the pITS queue. `vits_cq.progress` will be updated to reflect this.

Each `vits_cq` is handled in turn in this way until the pITS Command Queue is full or there are no more outstanding commands.

There will likely need to be a data structure which shadows the pITS Command Queue slots with references to the `vits_cq` which has a command currently occupying that slot and corresponding the index into the virtual command queue, for use when completing a command.

`VITS_BATCH_SIZE` should be small, TBD say 4 or 8.

Possible simplification: If we arrange that no guest ever has multiple batches in flight (which can occur if we wrap around the list several times) then we may be able to simplify the book keeping required. However this may need some careful thought wrt fairness for guests submitting frequent small batches of commands vs those sending large batches.

2.2.3 Completion

It is expected that commands will normally be completed (resulting in an update of the corresponding `vits_cq.creadr`) via guest read from `CREADR`. This will trigger a scheduling pass which will ensure the `vits_cq.creadr` value is up to date before it is returned.

A guest which does completion via the use of `INT` cannot observe `CREADR` without reading it, so updating on read from `CREADR` suffices from the point of view of the guests observation of the state. (Of course we will inject the interrupt at the designated point and the guest may well then read `CREADR`)

However in order to keep the pITS Command Queue moving along we need to consider what happens if there are no `INT` based events nor reads from `CREADR` to drive completion and therefore refilling of the Queue with other outstanding commands.

A guest which enqueues some commands and then never checks for completion cannot itself block things because any other guest which reads `CREADR` will drive completion. However if *no* guest reads from `CREADR` then completion will not occur and this must be dealt with.

Even if we include completion on `INT`-base interrupt injection then it is possible that the pITS queue may not contain any such interrupts, either because no guest is using them or because the batching means that none of them are enqueued on the active ring at the moment.

So we need a fallback to ensure that queue keeps moving. There are several options:

- A periodic timer in Xen which runs whenever there are outstanding commands in the pITS. This is simple but pretty sucky.
- Xen injects its own `INT` commands into the pITS ring. This requires figuring out a device ID to use.

The second option is likely to be preferable if the issue of selecting a device ID can be addressed.

A secondary question is when these `INT` commands should be inserted into the command stream:

- After each batch taken from a single `vits_cq`;
- After each scheduling pass;
- One active in the command stream at any given time;

The latter should be sufficient, by arranging to insert a `INT` into the stream at the end of any scheduling pass which occurs while there is not a currently outstanding `INT` we have sufficient backstop to allow us to refill the ring.

This assumes that there is no particular benefit to keeping the `CWRITER` rolling ahead of the pITS's actual processing. This is true because the IRS operates on commands in the order they appear in the queue, so there is no need to maintain a runway ahead of the ITS processing. (XXX If this is a concern perhaps the INT could be inserted at the head of the final batch of commands in a scheduling pass instead of the tail).

Xen itself should never need to issue an associated `SYNC` command, since the individual guests would need to issue those themselves when they care. The `INT` only serves to allow Xen to enqueue new commands when there is space on the ring, it has no interest itself on the actual completion.

2.2.4 Locking

It may be preferable to use `atomic_t` types for various fields (e.g. `vits_cq.creadr`) in order to reduce the amount and scope of locking required.

2.2.5 Multiple vITS instances in a single guest

As described above each vITS maps to exactly one pITS (while each pITS servers multiple vITSs).

In principal it might be possible to arrange that a vITS can enqueue commands to different pITSs depending on e.g. the device id. However this brings significant additional complexity (what to do with SYNC commands, how order completion such that one pITS does not block another, book keeping etc).

In addition the introduction of direct interrupt injection in version 4 GICs may imply a vITS per pITS. (XXX???)

Therefore it is proposed that the restriction that a single vITS maps to one pITS be retained. If a guest requires access to devices associated with multiple pITSs then multiple vITS should be configured.

2.2.6 vITS for purely software interrupts (e.g. event channels)

It has been proposed that it might be nice to inject event channels as LPIs in the future. Whether or not that would involve any sort of vITS is unclear, but if it did then it would likely be a separate emulation to the vITS emulation used with a pITS and as such is not considered further here.

3 Glossary

- *MSI*: Message Signalled Interrupt
- *ITS*: Interrupt Translation Service
- *GIC*: Generic Interrupt Controller
- *LPI*: Locality-specific Peripheral Interrupt

4 References

"GIC Architecture Specification" PRD03-GENC-010745 24.0