Linux Kernel Hotplug CPU Support

Zwane Mwaikambo
FSMLabs
zwane@fsmlabs.com

Ashok Raj
Intel
ashok.raj@intel.com

Rusty Russell
IBM
rusty@rustcorp.com.au

Joel Schopp
IBM
jschopp@austin.ibm.com

Srivatsa Vaddagiri
IBM
vatsa@in.ibm.com

Abstract

During the 2.5 development series, many people collaborated on the infrastructure to add (easy) and remove (hard) CPUs under Linux. This paper will cover the approaches we used, tracing back to the initial PowerPC hack with Anton Blanchard in February 2001, through the multiple rewrites to inclusion in 2.6.5.

After the brief history lesson, we will describe the approach we now use, and then the authors of the various platform-specific code will describe their implementations in detail: Zwane Mwaikambo (i386) Srivatsa Vaddagiri (i386, ppc64), Joel Schopp (ppc64), Ashok Raj (ia64). We expect an audience of kernel programmers and people interested in dynamic cpu configuration in other architectures.

1 The Need for CPU Hotplug

Linux is growing steadily in the mission critical data-center type installations. Such installations requires Reliability, Availability and Serviceability (RAS) features. Modern processor architectures are providing advanced error correction and detection techiniques. CPU hotplug provides a way to realize these features in mission critical applications. CPU hotplug feature adds the following ability to Linux to compete in the high end applications.

- **Dynamic Partitioning**

  Within a single system multiple Linux partitions can be running. As workloads change CPUs can be moved between partitions without rebooting and without interrupting the workloads.

- **Capacity Upgrade on Demand**

  Machines can be purchaced with extra CPUs, without paying for those CPUs until they are needed. Customers can at a later date purchase activiation codes that enable these extra CPUs to match increases in demand, without interrupting service. These activiation codes can either be for temporary activation or permanant activation depending on customer needs.

- **Preventive CPU Isolation**

  Advanced features such as CPU Guard in PPC64 architectures, and Machine Check Abort (MCA) features in Itanium® Product Family (IPF) permit the hardware to catch recoverable failures that are symptomatic of a failing CPU and remove that CPU before an unrecoverable failure occurs. An unused CPU can later be brought online to replace the failed CPU.

2 The Initial Implementation

In February 2001, Anton Blanchard and Rusty Russell spent a weekend modifying the ppc32 kernel to switch CPUs on and off. Stress tests on a 4-way PPC crash box showed it to be reasonably stable. The resulting 60k patch to 2.4.1 was posted to the linux-kernel on February the 4th: `http://www.uwsg.iu.edu/hypermail/linux/kernel/0102.0/0751.html`.

Now we know that the problem could be solved, we got distracted by other things. Upon joining IBM, Rusty had an employer who actually had a use for hotplugging CPUs, and in 2002 the development started up again.

The 2.4 kernels used `cpu_number_map()` to map from the CPU number given by `smp_processor_id()` (between 0 and `NUM_CPUS`) to a unique number between 0 and `smp_num_cpus`. This allows simple iteration between 0 and `smp_num_cpus` to cover all the CPUs, but this cannot be maintained easily in the case where CPU are coming and going. Given my experience that `cpu_number_map()` and `cpu_logical_map()` (which are noops on x86) are a frequent source of errors, Rusty chose to eliminate them, and introduce a `cpu_online()` function which would indicate if the CPU was actually online. Much of the original patch consisted of removing the number remapping, and rewriting loops appropriately.

This change went into Linus' tree in 2.5.24, June 2002, which made the rest of the work much less intrusive.

In the next month, as we were trying to get the `cpu_up()` function used for booting, Linus insisted that we also change the boot order so that we boot as if we were uni-processor, and then bring the CPUs up. Unfortunately, this patch broke Linus' machine, and he partially reverted it, leaving us with the current situation where a little initialization is done before secondary CPUs come online, and normal `__initcall` functions are done with all CPUs enabled. This change also introduced the `cpu_possible()` macro, which can be used to detect whether a CPU could ever be online in the future.

The old boot sequence for architectures was:

1. `smp_boot_cpus()` was called to initialize the CPUs, then
2. `smp_commence()` was called to bring them online.

In addition, each arch optionally implemented a "maxcpus" boot argument. This was made into an arch-independent boot argument, and the boot sequence became:

1. `smp_prepare_cpus(maxcpus)` was called to probe for cpus and set up `cpu_present(cpu)`[1], then

[1]On arch's that dont fill in `cpu_present(cpu)` the function `fixup_cpu_present_map` just uses what `cpu_possible_map` was set during probe. See the section in IA64 for more details.

2. `__cpu_up(cpu)` was called for each CPU where `cpu_present(cpu)` was true, then

3. `smp_cpus_done(maxcpus)` was called after every CPU has been brought up.

At this stage, the CPU notifier chain and the `cpu_up()` function existed, but CPU removal was not in the mainstream kernel. Indeed, significant scheduler changes occurred, preemption went into the kernel, and Rusty was distracted by the module reworking. The result: hotplug CPU development floundered outside the main tree for over a year.

3 The Problem of CPU Removal

The initial CPU removal patch was very simple: the process scheduled on the dying CPU, moved interrupts away, set `cpu_online()` to false, and then scheduled on every other CPU to ensure that noone was looking at the old CPU values. The scheduler's `can_schedule()` macro was changed to return false if the CPU was offline, so the CPU would always run the idle task during this time. Finally, the arch-specific `cpu_die()` function actually killed the CPU.

Three things made this approach harder as the 2.5 kernel developed:

1. Ingo Molnar's O(1) scheduler was included. Rather than checking if the CPU was offline every time we ran `schedule()`, we wanted to avoid touching the highly-optimized code paths.

2. The kernel became preemptible. This means that scheduling on every CPU is not sufficient to ensure that noone is using the old online CPU information.

3. Workqueue and other infrastructure was introduced which used per-cpu threads, which had to be cleanly added and removed.

4. More per-CPU statistics were used in the kernel, which sometimes need to be merged when a CPU went offline (or each sum must be for every possible CPU, not just currently online ones)

5. Sysfs was included, meaning that the interface should be there, instead of in proc, along with structure for other CPU features

Various approaches were discussed and tried: some architectures (like i386) merely simulate CPUs going away, by looping in the idle thread. This is useful for testing. Others (like PPC64 and IA64) actually need to re-start CPUs.

The following were the major design points which were tested and debated, and the resolution of each:

- How should we handle userspace tasks bound to a single CPU?

  Our original code sent a SIGPWR to tasks which were bound such that we couldn't move them to another CPU. This has the default behaviour of killing the task, which is unfortunate if the task merely inherited the binding from its parent. The ideal would be a new signal which would also be delivered on other reconfiguration events (like addition of CPUs, memory), but the Linux ABI does not allow the addition of new signals.

  The final result was to rely on the hotplug scripts to handle this information, and rely on userspace to ensure that removing a CPU was OK before telling the kernel to switch it off.

- How should we handle kernel threads bound to a single CPU?

  Unlike userspace, kernel threads often have a correctness requirement that they run on a particular CPU. Our original approach used a notifier between marking the CPU offline, and actually taking it down; these threads would then shut themselves down. This two-stage approach caused other complications, and the legendary Ingo Molnar recommended a single-stage takedown, and that the kernel threads could be cleaned up later. While that simplified things in general, it involved some new considerations for such kernel threads.

- Issues Creating And Shutting Down Kernel Threads

  In general, the amount of code required to stop kernel threads proved to be significant: barriers and completions at the very least. The other issue is that most kernel threads assume they are started at boot: they don't expect to be started from whatever random process which brought up the CPU.

  This lead Rusty to develop the "kthread" infrastructure, which encapsulated the logic of starting and stopping threads in one place. In particular, it uses keventd (which is always started at boot) to create the new thread, ensuring that there is no contamination by forking the userspace process. The `daemonize()` function attempts to do this, but it's more certain to start from a clean slate than to try to fix a existing one.

- Issues Using keventd for CPU Hotplug

  keventd is used as a general purpose kernel thread for performing some deferred work in a thread context. The "kthread" infrastructure uses this framework to start and stop threads. In addition when various kernel code attempts to call user-space scripts and agents use `call_usermode_helper()`. This function used the keventd thread to spawn the user space program. This approach caused a dead lock situation when the `call_usermode_helper()` is called as part of the `_cpu_disable()`, since keventd threads are per-CPU threads. This results in queueing work to keventd thread via `schedule_work()`, then waiting for completion. This results in blocking the keventd thread. Unless the work queued gets to run, this keventd thread would never be woken again. To avoid this scenario, Rusty introduced the `create_singlethread_workqueue` which now provides a separate thread that is not bound to any particular CPU.

- How to Avoid Having To Lock Around Every Access to Online Map

  Naturally, we wanted to avoid locking around every access to `cpu_online_map` (via `cpu_online()` for example). The method was one Rusty invented for the module code: the so-called "bogolock". To make a change, we schedule a thread on every CPU and have them all simultaneously disabled interrupts, then make the change. This code was generalized from the module code, and called `stop_machine_run()`. This means that we only need to disable preemption to access `cpu_online_map` reliably. If you need to sleep, the `cpu_control` semaphore also protects the CPU hotplug code, so there is a slow-path alternative.

- How to Avoid Doing Too Much Work With the Machine Stopped

  While all CPUs are not taking interrupts, we don't want to take too long. The initial code walked the task list while the machine was frozen, moving any tasks away from the dying CPU. Nick Piggin came up with an improvement which only migrated the tasks on the CPU's runqueue, and then ensured no other tasks were migrated to the CPU, which reduced the hold time by an order of magnitude. Finally Srivatsa Vaddagiri went one better: by simply raising the priority of the idle task with a special `sched_idle_next()` function, we ensure that nothing else runs on the dying CPU.

The process by which the CPU actually goes offline is as follows:

1. Take `cpu_control` semaphore,
2. Check more than one CPU is online (a bug Anton discovered in the first implementation!),
3. Check that the CPU which they are taking down is actually online,
4. Take the target CPU out of the CPU mask of this process. When the other steps are finished, they will wake us up, and we must not migrate back onto the dead CPU!
5. Use `stop_machine_run()` to freeze the machine and run the following steps on the target CPU
6. Take the CPU out of `cpu_online_map` (easier for arch code to do this first).
7. Call the arch-specific `__cpu_disable()` which must ensure that no more hardware interrupts are received by this CPU (by reprogramming interrupt controllers, or whatever),
8. If that call fails, we restore the `cpu_online_map`. Otherwise we call `sched_idle_next()` to ensure that when we exit the CPU will be idle.
9. At this point, back in the caller, we wait for the CPU to become idle, then call the arch-specific `__cpu_die()` which actually kills the offline CPU, by setting a flag which the idle task polls for, or using an IPI, or some other method.
10. Finally, the `CPU_DEAD` notifier is called, which the scheduler uses to migrate tasks off the dead CPU, the workqueues use to remove the unneeded thread, etc.

The implementation specifics of each architecture can be found in the following sections.

4 Remaining Issues

The main remaining issue is the interaction of the NUMA topology and addition of new CPUs. An architecture can choose a static NUMA topology which covers all the possible CPUs, but for logical partitioning this might not be possible (we might not know in advance).

- Per-CPU variables are allocated using `__alloc_bootmem_node()` at boot, for performance reasons. Unknown CPUs are usually assumed to be in the boot node, which will impact performance.
- sysfs node topology entries need to be updated when a CPU comes online, if the node association is not known at boot.
- The NUMA topology itself should be updated if it is only known when a CPU comes online. This is now possible, using the `stop_machine_run()` function,

but no architectures, other than PPC64, currently do this.

- There are likely some tools in use today that would require minor changes as well. One such tool identified is the top(1) utility, which has trouble dealing with the fact that CPU's available in the system are not logically contiguous. For e.g in a 4-way system, if logical cpu2 was offlined, when cpu0, cpu1, cpu3 were still functional, top would display some error information. Also the tool does not update the CPU information and not able to dynamically update them when new CPU's are added, or removed from the system.

5 i386 Implementation

Commercial i386 hardware available today offer very limited support for CPU Hotplug. Hence the i386 implementation, as it exists, is more of a toy for fun and experimentation. Nevertheless, it was used intensively during development for exercising various code paths and, needless to say, it exposed numerous bugs. Most of these bugs were in arch-independent code.

Since the hardware does not support physical hotplugging of CPUs, only logical removal of a CPU is possible. Once removed from the system, a dead CPU does not participate in any OS activity. Instead, it keeps spinning, waiting for a online command, in the context of its idle thread. Once it gets the online command, it breaks out of the spin loop, puts itself in `cpu_online_map`, flushes TLB and comes alive!

Some important i386 specific issues faced during development are described below:

- **Boot processor**
  There are a few interrupt controller configurations, which necessitate that we not offline the boot processor. Systems may be running with the I/O APIC disabled in which case all interrupts are being serviced by the boot processor via the i8259A, which cannot be programmed to direct interrupts to other processors. Another being interrupts which may be configured to go via the boot processor's LVT (Local Vector Table) such as various timer interrupt setups.

- **smp_call_function**
  smp_call_function is one tricky function which haunted us a long time. Since it deals with sending IPIs to online CPUs and waiting for acknowledgement, number of races was found in this function wrt CPUs coming and going while this function runs on some CPU. Fortunately, when CPU offline was made atomic, most of these race conditions went away. CPU online operation, being still non-atomic, exposes a race wherein an IPI can be sent to a CPU coming online and the sender will not wait for it to acknowledge the IPI. The race was fixed by taking a spinlock (`call_lock`) before putting CPU in the online_map.

- **Interrupt redirection**
  If I/O APIC is enabled, then its redirection table entries (RTEs) need to be reprogrammed every time a CPU comes and goes. This is so that interrupts are delivered to only online CPUs.

  According to Ashok Raj, a safe time to reprogram I/O APIC RTE for any interrupt is when that interrupt is pending, or when the interrupt is masked in RTE.

  Going by the first option, we would have to wait for each interrupt to become pending before reprogramming its RTE. Waiting like this for all interrupts to become pending may not be a viable solution during CPU Hotplug. Hence the method followed currently is to reprogram RTEs from the dying CPU and wait for a small period (20 microseconds) with interrupts enabled to flush out any pending interrupts. This, in practice, has been enough to avoid lost interrupts.

The right alternative however would be to mask the interrupt in RTE before reprogramming it, but also accounting for the case where the interrupt might have been lost during the interval the entry was left masked. A detailed description of this method is provided in IA64 implementation section.

- **Disabling Local Timer Ticks**
Local timer ticks are local to each CPU and are not affected by I/O APIC reprogramming. Hence when a CPU is brought down, we have to stop local timer ticks from hitting the dying CPU. This feature is not implemented in the current code. As a consequence, local timer ticks keep hitting and are discarded in software by a `cpu_is_offline` check in its interrupt handler. There are a few solutions under consideration in order to avoid adding a conditional in the timer interrupt path. One method was setting up an offline processor IDT (Interrupt Descriptor Table) which would be loaded when the processor was in the final offline state. The offline IDT would be populated with an entry stub which simply returns from the interrupt. This method would mean that any interrupts hitting the offline processor would be blindly discarded, something which may cause problems if an ACK was required. So what may be safer and sufficient is simply masking the timer LVT for that specific cpu and unmasking it again on the way out of the offline loop.

6 IA64 Implementation

6.1 What is Required to Support CPU Hotplug in IA64?

IA64 CPU hotplug code was developed once Rusty had the base infrastructure support ready. Some of the work that was done to bring the code to stable state include:

- Remove section identifiers marked with `__init` that are required after completing SMP boot. for e.g `cpu_init()`, `do_boot_cpu()` used to wakeup a CPU from SAL_BOOT_RENDEZ mode, `fork_by_hand()` used to fork idle threads for newly added CPUs on the fly.
- Perform a safe interrupt migration from the CPU being removed to another CPU without loss of interrupts.
- Handing off the CPU being removed to SAL_BOOT_RENDEZ mode back to SAL.
- Handling platform level dependencies that trigger physical CPU hotplug in a platform capable of performing it.

6.2 Handling IA64 CPU removal

The arch-specific call `_cpu_disable()` implements the necessary functionality to offline a CPU. The different steps taken are:

1. Check if the platform has any restrictions on this CPU being removed. Returning an error from `_cpu_disable()` ensures that this CPU is still part of the `cpu_online_map`.
2. Turn of local timer interrupt. In IA64 there is a timer interrupt per CPU and not

an external interrupt as in i386 case. It is required that the `timer_interrupt` does not happen any further. It is possible there is one pending, hence check if this interrupt is from an this is an offline CPU, and ignore the interrupt, but just return IRQ_HANDLED, so that the local SAPIC can honour other interrupt vectors now.

3. Ensure that all IRQs bound to this CPU are now targeted to a different CPU by programming the RTEs for a new CPU destination. On return from this step, there must be no more interrupts sent to this CPU being removed from any IOSAPIC.

4. Now the idle thread gets scheduled last, and waits until the CPU state indicates that this CPU must be taken down. Then it hands the CPU to SAL.

6.3 Managing IA64 Interrupts

6.3.1 When Is It Safe to Reprogram an IOSAPIC?

IOSAPIC RTE entries should not be programmed when its actively receiving interrupt signals. The recommended method is to mask the RTE, reprogram for new destination, and then re-enable the RTE. The `/proc/irq` write handlers were calling the set affinity handlers immediately which can cause loss of interrupts, including IOAPIC lockups. In `i386` the introduction of IRQ_BALANCE did this the right way, which is to perform the reprograming operation when an interrupt is pending by storing the intend to change interrupt destinations in a deferred array `pending_irq_balance`.

The same concept was extended to `ia64` as well for the proc write handlers. With the CPU hotplug patches, the write to `/proc/irq` entries are stored in an array and performed when the interrupt is serviced, rather than calling it potentially when an interrupt can also be fired. Due to the delayed nature of these updates, with CPU hotplug, the new destination CPU may be offlined before an interrupt fired and the RTE can be re-programmed. Hence before setting IRQ destination CPU for an RTE, the code should check if the new destination processor is in the `cpu_online_map`.

6.3.2 Why Turn Off Interrupt Redirection Hint With CPU Hotplug?

Interrupt destination in any IOSAPIC RTE must be re-programmed to a different CPU if the CPU being removed is a possible interrupt destination. Since we cannot wait for the interrupt to fire to do the reprogramming, we must force the interrupt destination in safe way. IA64 interrupt architecture permits a platform chipset to perform redirection based on lowest priority based on a hint in the interrupt vector (bit 31) provided by the operating system. If platform interrupt redirection is enabled, it would imply that we need to reprogram all the interrupt destinations, because hotplug code in OS cannot be sure which CPU the chipset is going to direct this interrupt to. Hence if CONFIG_HOTPLUG_CPU is enabled, then we disable platform redirection hint at boot time.

6.3.3 Safely Migrating Interrupt Destinations

The function `fixup_irqs()` performs all the necessary tasks for safely migrating interrupts, and reprogramming interrupt destinations for which this CPU being removed was a destination. The handling of IRQ is managed in 3 distinct phases.

- `migrate_irqs()` performs the job of identifying all IRQs with this CPU as the interrupt destination. This iteration also keeps track of IRQs identified in `vectors_in_migration[]` for later processing to cover cases of missed interrupts, since we mask RTEs during reprogramming, if the device asserted an interrupt during that time, they get lost.

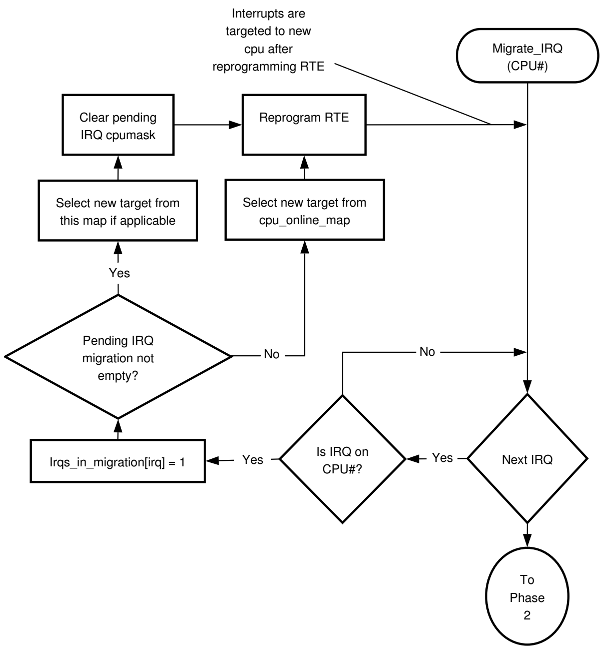
Interrupts are targeted to new cpu after reprogramming RTE
Migrate_IRQ (CPU#)
Clear pending IRQ cpumask
Reprogram RTE
Select new target from this map if applicable
Select new target from cpu_online_map
Yes
Pending IRQ migration not empty?
No
No
Irqs_in_migration[irq] = 1
Yes
Is IRQ on CPU#?
Yes
Next IRQ
To Phase 2

Figure 1: Phase1: Migrate IRQ

- `ia64_process_pending_intr()` Does normal interrupt style processing. During this phase, we look at the local APIC interrupt vector register `ivr` and process all pending interrupts on this CPU. For each processed interrupt, we also clear the bits set in `vectors_in_migration[]`.

- Phase 3 accounts for cases where a device possibly attempted to assert an interrupt, but got lost during the window the RTE was also being re-programmed. This phase looks at entries not accounted for in phase 2, and issues interrupt handler callbacks as if an interrupt happened. It is likely there were no interrupts asserted. We rely on the fact that most device drivers can tolerate calls even if there was no work to perform due to the fact that IRQs may be shared.

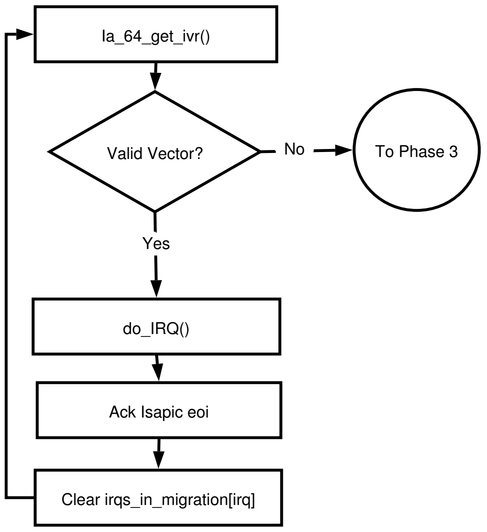

Figure 2: Phase2: Processing Pending intr

6.3.4 Managing Platform Interrupt Sources

IA64 architecture specifies platform interrupt sources to report corrected platform errors to the OS. ACPI specifies these sources via the Platform Interrupt Source Structures. These are communicated to the OS with data such as the following.

- Interrupt Type, indicating if the interrupt is Platform Management Interrupt (PMI), INIT, or CPEI.

- IOSAPIC vector the OS should program.

- The processor that should receive this interrupt, by specifying the APIC id.

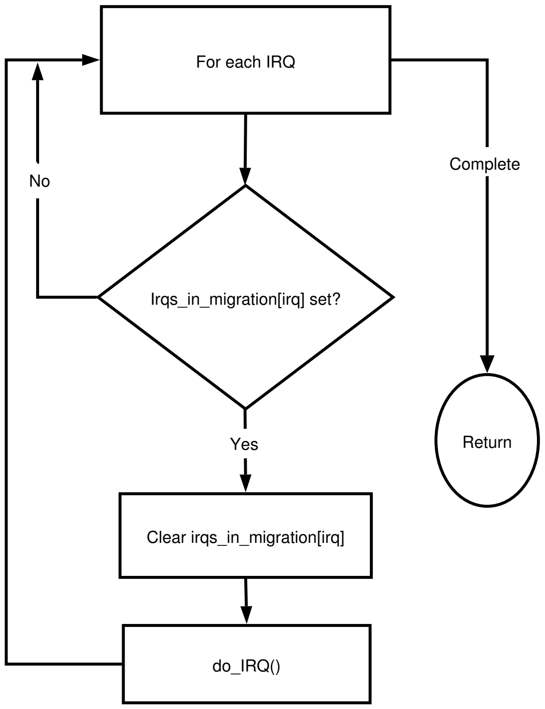

Figure 3: Phase3: Account for Lost Interrupts

- The interrupt line used to signal the interrupts by specifying the global system interrupt.

Some platforms do not support an interrupt model for retrieving platform errors via CPEI. Such platforms provide support via specifying polling tables that list all processors that can poll for Correctable Platform Errors by using the Correctable Platform Error Polling (CPEP) tables.

The issue with both above schemes is that CPEI specifies just one entry for a destination processor. This automatically restricts the target CPU that handles CPEI not removable. On the other hand with CPEP polling tables, although the scheme permits specifying more than one processor, the tables are static and cannot be expanded dynamically as new processors capable of handling polling to be updated.

The motivation for restricting certain processors was that for some platforms that are asymmetric, not all CPUs can retrieve the platform error registers. Hence it is required that only certain processors are permitted. Most platforms that support interruptible model are symmetric in nature. Hence any CPU is capable of accepting the interrupt for CPEI.

We are working with the ACPI specification team to try and address this capability to support platforms supporting CPU hotplug. In the interim before a specification change permits either specifying any CPU as a target, or a method to dynamically update the processors before a CPU gets removed, the code would fail removal of a CPU that is a target of CPEI. In the case of polling, the last processor in the list would be made non-removable.

6.4 Why Should the CPU be handed off to SAL?

The Itanium® processor architecture provides a machine check abort mechanism for reporting and recovering from a variety of errors that can be detected by the processor or chipset. In the event of global MCA, it is required that the slave processors perform checkin with the monarch processor, before which the master could call the recovery to resume execution. SAL would exclude processors in SAL_BOOT_RENDEZ mode. Hence it is important that we return the offlined processors to SAL to avoid processing MCA events on the offlined processor, as the OS would not have it in the active map of CPUs.

6.5 Handling Boot CPU Removal

IA64 architecture does not have any direct dependency that would preclude the boot CPU being removable. There may be some platform level issues such as the boot CPU is usually the target of CPEI or some such dependency that would make the boot CPU from being removable. In the existing IA64 code base, there is one dependency, that the boot CPU (CPU0) is the master time keeper. This dependency can be easily removed by electing a new CPU as the master timekeeper.

6.6 Recovering the Idle Thread After CPU Removal

Idle threads are created on demand when a new CPU is added to the OS image. These threads are special, since when we return the processor back to SAL, this is done from the context of the idle thread. These calls don't return, and don't have a natural exit path as other threads. The simplest thing to do would be to keep these free idle threads, and just reuse them the next time we need to create a new idle thread for a new CPU.

6.7 Why Was `cpu_present_map` Introduced?

There are several pieces of kernel code that size resources upfront. Before the advent of CPU hotplug, the variable `cpu_possible_map` also indicated the CPUs physically available in the system and would eventually be booted via `smp_init()`. It is very intrusive to make all these callers behave dynamically to CPU hotplug code. There are some issues around this is use of `boot_mem_allocator`. In order to simplify these issues the map `cpu_possible_map` was set to all bits indicating `NR_CPUS`. In order to start only CPUs that are physically present in the system, the new map `cpu_present_map` was added. On platforms capable of supporting CPU hotplug, this map would dynamically change depending on a new CPU being added or removed from the system. In order to accommodate systems that don't directly populate `cpu_present_map` the function `fixup_cpu_present_map` was introduced to just copy the bits from `cpu_possible_map` to `cpu_present_map`.

6.8 ACPI and Platform Issues With CPU hotplug

Any platform capable of supporting hotpluggable CPUs must provide a mechanism to initiate hotplug. Platforms supporting ACPI aware OSs could use ACPI mechanisms to initiate hotplug activity which I would call Physical CPU hotplug. The CONFIG_HOTPLUG_CPU provides the kernel capability and could still be useful if a CPU can be taken offline based on say, the number of correctable error rate.

A typical sequence of operations on a platform supporting a physical CPU is described below. Each specific platform may have additional steps, the following is only a possible sequence and applies to the ACPI based implementations as well.

1. Insert the CPU or the module that contains the CPU into the platform.

2. Platform BIOS does some preparation, and notifies the OS. The kernel platform component such as ACPI that registered to receive the notification, processes this event.

3. Platform dependent OS component prepares necessary information required to bring this CPU to the OS image. For example, in IA64, the code would initialise the following data structures before calling the `cpu_up()`:

   - `ia64_cpu_to_sapicid[]`, in the case of NUMA also populate `node_to_cpu_mask` and `cpu_to_node_map` necessary for NUMA kernels.

- Populate `cpu_present_map` so that kernel now knows about this new CPU is present in the system.

4. Create the necessary entries such as `/sys/devices/system/cpu/cpu#`.

5. Launch the `/sbin/hotplug` script that will now invoke the CPU hotplug agent, which in turn would use the sysfs entry just created to bring up the new CPU.

7 PPC64 Implementation

7.1 What PPC64 Specific Tasks Occur During a CPU Removal?

The architecure specific kernel pieces of a CPU removal focus on three functions mentioned previously: `__cpu_disable()`, `cpu_down()`, and `__cpu_die()`.

In `__cpu_disable()` all interrupts are disabled and migrated, with the exception of interprocessor interrupts (IPIs).

1. The process of disabling interrupts starts off by writing 0 into the processor's current processor priority register (CPPR) to reject any possible queued interrupts.

2. With the CPPR set to 0 it is safe to remove ourselves from the global interrupt queue server, which is done via a Run-Time Abstraction Service (RTAS) set-indicator call that is provided by the firmware. This has the effect of refusing new interrupts from being added to the processor.

3. After new interrupts are refused the next step is to set the CPPR back to default priority, which allows us to recieve IPIs again.

4. All interrupts are iterated through, checking via an RTAS "get-xive" call if any of the interrupts are specific to the target processor.

5. If an interrupt is specific to the target processor it is migrated via an RTAS "set-xive" call.

6. With the processor removed from the global interrupt queue server and all interrupts migrated it would be safe to remove the target processor without affecting the delivery of interrupts. Success is returned.

During `__stopmachine_run()` the online attribute of a CPU is set to to 0. On PPC64 we stop the CPU at this point by calling `cpu_die()` (not to be confused with `__cpu_die()`)

1. Depending on the machine model and kernel configuration, the idle function will be `default_idle()`, `dedicated_idle()`, or `shared_idle()`. All three idle functions check `cpu_is_offline()` and if it is true call `cpu_die()`.

2. `cpu_die` first disables IRQs.

3. After disabling IRQs it clears the CPPR.

4. Finally `rtas_stop_self()` is called, stopping the processor.

Most architectures use `__cpu_die()` to stop the processor. Because on PPC64 we poll for offline CPUs we only need to wait and confirm the CPU has been stopped while in this function.

1. We confirm the CPU has been stopped by using the RTAS `query-cpu-stopped-state` call.

2. Because this call can return busy, and because the CPU may not yet be stopped we loop and schedule timeouts.

3. After confirming the CPU is stopped we do a little extra cleanup by clearing the corresponding entry in the `cpu_callin_map` and `xProcStart` in the PACA.

7.2 What About Adding CPUs?

The initial structure of PPC64 CPU bringup required a lot of modification to be able to add CPUs after the system was already running. Most of the changes are trivial and straightforward, but one bears mentioning.

PPC64 used to number CPUs based on their physical id. With CPU hotplug it would have been necessary to reserve a CPU entry and corresponding structures for each possible physical CPU. It was quite possible that the machine could have more CPUs than the kernel was compiled to work with, as many CPUs would be assigned to other partitions. Furthermore, the number of CPUs in the machine was not necessarily a static number. Also, from a usability point of view there were going to be far too many entries in `/sys/devices/system/cpu/` compared to how many CPUs were actually online.

The CPU numbering was logically abstracted so that for kernel use there was a logical number, and when interfacing to the hardware there was a corresponding physical number. The kernel is able to read at boot time the maximum number of CPUs the partition is configured to be able to grow to. Thus it reserves less space in structures that must be allocated at boot time, allows reuse of logical CPUs for different physical CPUs, and presents a cleaner directory structure.

7.3 Other Software

While outside the scope of this paper it is worth mentioning that there is other software running on PPC64 platforms to enable customers halfway around the world from the machines they administer to use their mouse and move CPUs. This software is downloadable from IBM, and should be available on the bonus CD shipped with new machines.

Proceedings of the Linux Symposium

Volume Two

July 21st–24th, 2004
Ottawa, Ontario
Canada