Cleaning Up Linux's CPU Hotplug For Real Time and Energy Management

Thomas Gleixner
Linutronix

Paul E. McKenney
IBM LTC assigned to Linaro

Vincent Guittot
ST-Ericsson assigned to Linaro

Abstract

Linux's CPU-Hotplug facility was originally designed to allow failing hardware to be removed from a running system. Hardware fails quite infrequently, so CPU-hotplug performance (much less real-time response) was not a major consideration. However, CPU hotplug is now used for energy management and (believe it or not!) real-time response, both of which have unsurprisingly exposed some shortcomings in CPU hotplug. This document reviews a number of these shortcomings, and then proposes an alternative CPU-hotplug approach that we believe will address these shortcomings.

1 Introduction

The Linux kernel's CPU-hotplug facility allows CPUs to be added to or removed from a running kernel. CPU hotplug has historically been used to isolate failing CPUs or to simplify running scalability benchmarks [5]. The roots of the Linux kernel's CPU-hotplug facility go back almost ten years [4], but during that time, it has gained a number of additional uses, including adjusting the sizes of guest OSes in virtualized environments, clearing current and future work from a given CPU, and improving energy efficiency. These new uses place considerable stress on the Linux kernel's implementation, which was not designed with them in mind.

Linux's CPU-hotplug implementation is based on *notifiers*, which are callbacks into the subsystems that need to be aware of CPUs coming and going. These notifiers are invoked repeatedly in multiple phases, so that when a CPU is coming online, they are invoked with `CPU_UP_PREPARE` (which runs on some other CPU), then `CPU_STARTING` (which runs with interrupts disabled on the CPU coming online), and finally `CPU_ONLINE` (which might run on any CPU, but after the CPU has come online). CPUs going offline have four notification phases: `CPU_DOWN_PREPARE` (which might run on any CPU), `CPU_DYING` (which runs with interrupts disabled on the offlining CPU while all other CPUs are spinning waiting), `CPU_DEAD` (which runs on some other CPU after the CPU has gone offline), and `CPU_POST_DEAD` (which runs on some other CPU after some of the CPU-hotplug locks have been dropped). The `CPU_UP_PREPARE` and `CPU_DOWN_PREPARE` notifiers are permitted to "fail", in other words, to refuse to allow the hotplug operation to proceed.

Section 2 reviews shortcomings of the current implementation, Section 3 overviews our work in progress, Section 4 lists alternative proposals, and Section 5 lists potential issues with our approach.

2 CPU-Hotplug Shortcomings

The shortcomings of CPU hotplug are well known, but worth discussion. The most obvious from a real-time-computing perspective is OS jitter, discussed in Section 2.1. From a Linux-kernel implementation viewpoint, the lack of a well-defined CPU model during hotplug operations is most vexing, as described in Section 2.2. A few unlucky portions of the Linux kernel must correctly handle offline (or "zombie") CPUs, which is covered in Section 2.3. Finally, CPU hotplug notifies kernel subsystems of hotplug operations, but a number of these notifiers run in extremely constrained software contexts, as documented in Section 2.4.

2.1 Overhead and OS Jitter

In a perfect world, a given CPU could come online or go offline quickly and without disturbing the rest of the system. Unfortunately, in this world, handling the CPU's per-CPU kthreads takes a long time (hundreds of milliseconds or even seconds) [1]. This limits CPU hotplug's use as an energy efficiency measure because the CPU must stay powered off for quite some time to make

up for the CPU-hotplug overhead [3][1]. Furthermore, CPU hotplug uses __stop_machine(), which halts application execution on all online CPUs for an extended period of time. This rules out use of CPU hotplug for real-time workloads—and makes its use difficult on battery-powered systems because the CPU hotplug operation might consume more energy than is saved by powering off the CPU for a short time.

The traditional rule "don't use CPU hotplug on real-time systems" is now starting to fail due to real-time guests running on real-time hypervisors. In this case, the hypervisor needs to offline a failing CPU without causing all the real-time guest OSes to miss their deadlines. Furthermore, a guest might need to add or remove CPUs without disrupting its real-time application.

In addition, offlining and immediately onlining a CPU has the useful side-effect of forcing all current and future work off of that CPU, providing better response to its real-time application. Unfortunately, such offlining and onlining will cause any pre-existing real-time application running on that same system to miss its deadlines.

These use cases are specific examples of the trend towards increasing general-purpose functionality on real-time systems [2]. Now that CPU hotplug has moved away from its original intended use of removal of failing CPUs, excessive overhead and OS jitter from CPU-hotplug operations is no longer acceptable.

2.2 Ill-Defined Model of CPU

Suppose that a given CPU is going offline, and that half of its notifiers have completed. What state is the CPU in?

The answer to this question is unclear. The CPU is marked as online in the cpu_online_mask, but some of its functionality really has been disabled. Worse yet, the default is for the notifiers to execute in the same order as they were registered at boot time. This is a problem because the boot process adds capabilities to the CPU in a good order that respects dependencies among those capabilities, which means that removing them in the same order can be problematic. For example, the scheduler uses both RCU and IPIs, and during boot, RCUs and IPIs are initialized before the scheduler. So the boot process builds up the CPU's capabilities in a tree-like fashion, and then the CPU-hotplug system attempts to remove the tree starting at the trunk, in this case removing RCU and IPIs before removing the scheduler.

Although notifier priorities are used to handle this specific case, this requires painstaking manual intervention. The Linux kernel deserves better.

[1] Five milliseconds is a good upper bound on CPU-hotplug latency.

2.3 Zombie CPUs

The __stop_machine() primitive forces all CPUs on the system to switch to special kernel threads (kthreads). The outgoing CPU then executes CPU_DYING class of notifiers in the context of its kthread, while the other CPUs spin in the context of their kthreads. Therefore, a newly offlined CPU passes through the scheduler when switching from its __stop_machine() kthread to the idle loop, where it is powered off.

This in turn means that both the scheduler and RCU must handle zombie offline CPUs for a short period after they have marked themselves offline. RCU handles this by assuming that a given CPU will not remain a zombie for more than one jiffy, which does currently work, but will eventually lead to baffling failures. Again, the Linux kernel deserves better.

2.4 Inconvenient Software Contexts

The CPU_DYING and CPU_STARTING classes of notifiers execute with interrupts disabled, which prevents them from blocking, which in turn prevents them from starting or stopping kthreads, which in turn can be problematic.

For example, when preemptible RCU is configured with priority boosting, it uses a set of per-CPU kthreads to boost callback-execution priority. RCU must interact with these threads in the CPU_UP_PREPARE, CPU_ONLINE, CPU_DOWN_PREPARE, and CPU_DEAD notifiers, which means that RCU must deal with either a CPU that doesn't have an RCU kthread or an RCU kthread that doesn't have a CPU, both of which are fragile and bug-prone. Once again, the Linux kernel deserves better.

3 Approach

A successful approach to new-age CPU hotplug must provide the following:

1. Robust design for CPUs that are partially online.
2. Simple and fast handling of per-CPU kthreads.
3. Explicit specification of notifier dependencies.
4. Parallel CPU-hotplug notification.
5. Full software capabilities in all CPU-hotplug notifiers.
6. Elimination of OS jitter.

To provide all this, our approach provides a generic facility to create and park per-CPU hotplug kthreads (see Section 3.1), executes execute in per-CPU kthread context (see Section 3.2, runs notifiers in reverse order for offline (see Section 3.3), and restricts hotplug-time execution to per-CPU hotplug kthreads (see Section 3.4.

```
 1 static int my_cpu_hotplug_kthread(void *arg)
 2 {
 3   int cpu = (int)(long)arg;
 4
 5   /*
 6    * Code here from CPU_STARTING notifier.
 7    */
 8
 9   cpu_hotplug_kthread_started();
10
11   while (!kthread_should_park()) {
12
13     /* Do actual work here. */
14
15   }
16
17   /*
18    * Code here from CPU_DYING notifier.
19    */
20
21 }
```

Figure 1: CPU-Hotplug Per-CPU kthread Structure

3.1 Generic Per-CPU Hotplug kthreads

Vincent established that creation and deletion of kthreads can add multiple *seconds* to CPU-hotplug latencies [1]. One best to avoid this is simply to leave those kthreads in place while the corresponding CPU is offline. A key observation (due to Thomas) is that the Linux kernel already has some threads that remain runnable and bound to offline CPU, namely the idle threads. A generic facility will allow the per-CPU hotplug kthreads to have this same capability, so that they remain quiescent while the corresponding CPU is offline.

3.2 Notify From Per-CPU kthreads

Given special kthreads managed by the CPU-hotplug facility, it makes sense to run the code currently in CPU-hotplug notifiers from within these kthreads. This allows the `CPU_DYING` and `CPU_STARTING` notifiers to use the full capabilities of the scheduler, allowing more of the notifier code to execute at `CPU_DYING` and `CPU_STARTING` time. This in turn simplifies the CPU-hotplug per-CPU kthreads, as shown by the `my_cpu_hotplug_kthread()` function in Figure 1. When a CPU boots or comes online, this function is invoked. When it finishes initialization, it invokes `cpu_hotplug_kthread_started()` as shown on line 11, signalling that the next notifier or kthread may now be started.

The loop spanning lines 11-15 terminates when `kthread_should_park()` returns true, indicating that the corresponding CPU is going offline. The function then executes offline-time cleanups as indicated by the comment on lines 17-19.

As noted earlier, the `CPU_UP_PREPARE` and `CPU_DOWN_PREPARE` phases can block CPU hotplug. Those that actually do (for example, `smp_core99_cpu_notify()`) must remain notifiers. That said, most do not, and can therefore can run in kthread context.

However, a great many of the Linux kernel's notifiers do not involve a kthread. Creating an additional per-CPU kthread for each of these notifiers would be overkill, so these notifiers should remain notifiers. They nevertheless should run in kthread context, for example, in the context of the kthread coordinating CPU-hotplug operation.

3.3 Reverse Notifier Order For Offline

One of the reasons for notifier priorities and for the current multi-phase CPU-hotplug operation is dependencies among different subsystems. These dependencies must be handled manually in a distributed fashion, and is a major source of pain and of bugs.

A better approach is to note that CPU hotplug is not atomic, and that CPUs are booted up in an orderly manner, with later function depending on earlier function. For example, the scheduler uses IPIs and RCU, so a CPU initializes its IPI and RCU handling before it starts scheduling processes. Given that the scheduler relies on IPIs and RCU, it makes no sense whatsoever for the CPU-hotplug offlining path to shut down IPIs and RCU before the scheduler. However, that is exactly what the current CPU-hotplug notifier operation encourages by invoking offline-time notifiers in the same order that it invokes online-time notifiers.

The only reasonable approach is to run the offline-time notifiers in the opposite order from online-/boot-time notifiers. With this approach, IPIs and RCU are notified before the scheduler when a CPU comes online, and the scheduler is notified before IPIs and RCU when a CPU goes offline. This means that the scheduler can count on IPIs and RCU being operational at all times.

In addition, this approach permits a CPU to be partially torn down to a well-defined checkpoint, for example, a CPU might be torn down to the point that all it can do is run in the idle loop, possibly permitting more aggressive power-efficiency measures to be brought to bear while providing improved CPU-online latency. Another example is partially tearing the CPU down to the point that it still handles interrupts, but does not run normal tasks, providing a form of CPU isolation.

There will of course be the occasional necessary evil of layering violations, but with this scheme such violations should be the exception rather than the rule.

3.4 Only Per-CPU Hotplug kthreads During CPU Hotplug

One feature of the current CPU-hotplug approach that has caused RCU much grief is the fact that interrupts are

disabled during the CPU_STARTING and CPU_DYING notifiers. This means that RCU cannot create or destroy kthreads at the logical time to do so, but must instead do so in one of the other notifiers, and handle either semi-crippled or semi-safe operations betweentimes.

On the other hand, it would be far worse to continue running arbitrary tasks during this time because those tasks would use facilities that had already been torn down. This is bad for the kernel's actuarial statistics.[2]

One solution to this problem is for the scheduler to run only special CPU-hotplug per-CPU tasks during CPU-hotplug processing. This allows the notifiers to make full use of the scheduler facilities when handling their kthreads, while preventing unwary normal tasks from straying onto a CPU that is only half working.

4 Alternative Approaches Considered

We considered several alternatives. The first alternative, continuing with existing CPU hotplug, was dispensed with in Section 2.

The second alternative was continuing with existing CPU-hotplug, but running the offline-time notifiers in reverse order. While this would be an improvement, it would do nothing to solve the kthread-parking problem.

The final alternative was dispensing with CPU hotplug completely in favor of things like cpusets and interrupt affinity. However, the most troublesome aspects of CPU hotplug are inherent in clearing all current and future work from a given CPU [3], so little is gained. In addition, CPU hotplug is still required for failing hardware.

5 Issues

Old-Style Interrupt Controllers Some interrupt controllers cannot be directed away from a given not-yet-offline CPU. If such an interrupt controller absolutely must be used, we propose interrupt trampolining as a workaround.

Scheduler Once RCU has marked a CPU as offline, it cannot safely do a context switch because the scheduler relies on RCU. This can be handled by splitting the RCU notifiers into an earlier-in-boot notifier that handles marking the CPU online or offline, and a later-in-boot notifier that manages RCU's kthreads.

Early-Boot kthreads RCU initializes itself and registers its notifiers during early boot, long before kthreads may be created. The implementations of RCU that require kthreads manually defer kthread creation until after the scheduler is running.

x86 MTRRs Updates to x86 memory type range registers (MTRRs) require that all hardware threads in a given core be quiesced. Although this might slow down hotplug for hyperthreaded x86 kernels it should be quite a bit faster than a full stop_machine().

Scanning Online CPUs One of the advantages of stop_machine() is that the final CPU-offline process appears atomic to any in-kernel code that is not hotplug-aware. Removing this atomicity means that each of the several hundred occurrances of for_each_online_cpu() (which iterates over all online CPUs) will need to be inspected and perhaps modified.

6 Summary and Conclusions

We expect that the new approach to CPU hotplug will reduce hotplug latencies to 5ms, making Linux more useful in both the real-time and battery-powered-embedded arenas.

Acknowledgements and Legal Statement

We thank Amit Kucheria, Peter Zijlstra, Srivatsa Bhat, and Steven Rostedt for many valuable and illuminating discussions. We are indebted to Dave Rusling and Jim Wasko for their support of this effort.

References

[1] GUITTOT, V. Cpu hotplug. Available: https://wiki.linaro.org/WorkingGroups/PowerManagement/Doc/Hotplug [Viewed April 20, 2012], February 2012.

[2] MCKENNEY, P. E. SMP and embedded real time. *Linux Journal*, 153 (January 2007), 52–57. Available: http://www.linuxjournal.com/article/9361 [Viewed May 31, 2007].

[3] MCKENNEY, P. E. The linaro connect scheduler minisummit. Report from the Q2 2012 Linaro Connect scheduler mini-summit on ARM's big.LITTLE architecture., February 2012.

[4] RUSSELL, R. Hotplug cpu toy for i386. Available: http://lwn.net/Articles/76667/ [Viewed April 25, 2012], March 2004.

[5] SEQUENT COMPUTER SYSTEMS, INC. tmp_ctl - multi-processor-control. http://oss.sgi.com/projects/numa/download/dynix, March 2001.

[2] That said, RCU currently must handle offline CPUs running through the scheduler on their way from their CPU_DYING notifiers to the idle loop. RCU's method of handling this issue is at best inelegant.