Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

Getting maximum mileage out of tickless

Suresh Siddha Venkatesh Pallipadi Arjan Van De Ven
Intel Open Source Technology Center
{suresh.b.siddha|venkatesh.pallipadi|arjan.van.de.ven}@intel.com

Abstract

Now that the tickless(/dynticks) infrastructure is integrated into the base kernel, this paper talks about various add on changes that makes tickless kernels more effective.

Tickless kernel pose some hardware challenges that were primarily exposed by the requirement of continuously running per-CPU timer. We will discuss how this issue was resolved by using HPET in a new mode. Eliminating idle periodic ticks causes kernel process scheduler not do idle balance as frequently as it would do otherwise. We provide insight into how this tricky issue of saving power with minimal impact on performance, is resolved in tickless kernel.

We will also look at the kernel and user level daemons and drivers, polling for things with their own timers and its side effect on overall system idle time, with suggestions on how to make these daemons and drivers tickless-friendly.

1 Introduction

Traditionally, SMP Linux (just as most other operating systems) kernel uses a global timer (present in the chipset of the platform) to generate a periodic event for time keeping (for managing time of the day) and periodic local CPU timer event for managing per-CPU *timers*. *Timers* are basically events that the kernel wants to happen at a specified time in the future (either for its own use or on behalf of an application). An example of such a timer is the blinking cursor; the GUI wants the cursor to blink every 1.2 seconds so it wants to schedule an operation every 600 milliseconds into the future. In the pre tickless kernel, per-CPU "timers" were implemented using the periodic timer interrupt on each CPU. When the timer interrupt happens, the kernel checks if any of the scheduled events on that CPU are due and if so, it performs the operation associated with that event.

Old Linux kernels used a 100 Hz frequency (every 10 milliseconds) timer interrupt, newer Linux uses a 1000 Hz (1ms) or 250 Hz (4ms) timer interrupt. The frequency of the timer interrupt determines how fine grained you can schedule future events. Since the kernel knows this frequency, it also knows how to keep time: every 100 / 250 / 1000 timer interrupts another second has passed.

This periodic timer event is often called "the timer tick." This timer tick is nice and simple but has two severe downsides: First of all, this timer tick happens periodically irrespective of the processor state (idle Vs busy) and if the processor is idle, it has to wake up from its power saving sleep state every 1, 4, or 10 milliseconds, which costs quite a bit of energy (and hence battery life in laptops). Second of all, in a virtualization environment, if a system has 50 guests that each have a 1000 Hz timer tick, the total system will have a 50,000 effective ticks. This is not workable and highly limits the number of possible guests.

Tickless(/Dynticks) kernel is designed to solve these downsides. This paper will briefly look into the current tickless kernel implementation and then look in detail about the various add-ons the authors added to make the tickless kernel more effective. Section 2 will look into the current tickless kernel infrastructure. Section 3 will look at the impact of different timers in the kernel and its impact on tickless kernel. Section 4 will address the hardware challenges that were exposed by the requirement of per-CPU timer.

Section 5 will talk about the impact on process load balancing in tickless idle kernels and the problem resolution. Section 6 will look at the impact of the different timers (and polling) in user level daemons and drivers on the power savings aspect of the tickless kernel.

2 Tickless Kernel

Tickless kernel, as the name suggests, is a kernel without the regular timer tick. Tickless kernel depends on architecture independent generic clocksource and clockevent management framework, which got into the mainline in the recent past [7]. Generic timeofday and clocksource management framework moved lot of timekeeping code into architecture independent portion of code, with architecture portion reduced to defining and managing low level hardware pieces of clocksources. While clock sources provide read access to the monotonically increasing time value, clock event devices are used to schedule the next event interrupt(s). The setup and selection of the event devices is hardwired into the architecture dependent code. The clock events provides a generic framework to manage clock event devices and their usage for the various clock event driven kernel functionalities.

Linux kernel typically provided timers which are of tick (HZ) based timer resolution. clockevents framework in the newer kernels enabled the smooth implementation of high resolution timers across architectures. Depending on the clock source and clock event devices that are available in the system, kernel switches to the hrtimer [8] mode and per CPU periodic timer tick functionality is also provided by the per CPU hrtimers (managed by per-CPU clockevent device).

Hrtimer based periodic tick enabled the functionality of tickless idle kernel. When a CPU goes into idle state, timer framework evaluates the next scheduled timer event and in case that the next event is further away than the next periodic tick, it reprograms the per-CPU clockevent device to this future event. This will allow the idle CPU to go into longer idle sleeps without the unnecessary interruption by the periodic tick.

The current solution in 2.6.21 [4] eliminates the tick during idle and as such it is not a full tickless or dynamic tick implementation. However, eliminating the tick during idle is a great first step forward. Future trend however is to completely remove the tick, irrespective of the busy or idle CPU state.

2.1 Dynticks effectiveness data

Effectiveness of dynticks can be measured using different system information like power consumption, idle time etc. Following sections of this paper looks at the dynticks effectiveness in terms of three measurements done on a totally idle system.

- The number of interrupts per second.
- The number of timer events per second.
- Average amount of time CPU stays in idle state upon each idle state entry.

These measurements, collectively, gives an easy approximation to actual system power and battery consumption.

These measurements reported were taken on a totally idle system, few minutes after reboot. The number of interrupts in the system, per second, is computed by taking the average from output of `vmstat`. The value reported is the number of interrupts per second on the whole system (all CPUs). The number of events reported is from `/proc/timer_stats` and the value reported is events per second. Average CPU idle time per call is the average amount of time spent in a CPU idle state per entry into idle. The time reported is in uS and this data is obtained by using `/proc/acpi/processor/CPU*/power`. All these three measurements are at the system level and includes the activity of all CPUs.

The system used for the measurement is a Mobile reference system with Intel® Core™ 2 Duo CPUs (2 CPU cores), running i386 kernel with a HZ rate of 1000.

2.1.1 Baseline measurement

Table 1 show the data with and without dynticks enabled.

As the data indicates, dyntick eliminates the CPU timer ticks when CPUs idle and hence reduces the number of interrupts in the system drastically. This reduction in number of interrupts also increases the amount of time CPU spends in an idle state, once entering that idle state.

Is this the best that can be done or is there scope to do more changes within the kernel to reduce the number of interrupts and/or reduce the number of events? Following sections addresses this specific question.

	# interrupts	#events	Avg CPU idle residency (uS)
With ticks	2002	59.59	651
Tickless	118	60.60	10161

Table 1: System activity during idle with and without periodic ticks

3 Keeping Kernel Quiet

One of the noticeable issue with the use of timers, inside core kernel, in drivers and also in userspace, is that of staggered timers. Each user of a timer does not know about other timers that may be setup on the system at the particular time and picks a particular time based on its own usage restrictions. Most of the time, this specific usage restriction does not mandate a strict hard time value. But, each timer user inadvertently end up setting the timer on their own and hence resulting in a bunch of staggered timers at the system level.

To prevent this in kernel space, a new API `__round_jiffies()` is introduced [6] in the kernel. This API rounds the jiffy timeout value to the nearest second. All users of timeout, who are really not interested in precise timeout should use this API while setting their timeout. This will cause all such timers to coalesce and expire at the same jiffy, preventing the staggered interrupts.

Another specific issue inside the kernel that needed special attention are the timer interrupts from drivers which are only important when CPU is busy and not really as important to wake the CPU from idle and service the timer. Such timer can tolerate the latency until some other important timer or interrupt comes along, at which time this timer can be serviced as well. Classic example of this usage model is cpufreq ondemand governor.

ondemand governor monitors each processor utilization at periodic intervals (many times per second) and tries to manage the processor frequency, keeping it close to the processor utilization. When a processor is totally idle, there is no pressing need for ondemand to periodically wakeup the processor just to look at its utilization. To resolve this issue, a new API of `deferrable timers` was introduced [1] in the recent kernel.

Deferrable timer is a timer that works as a normal timer when processor is active, but will be deferred to a later time when processor is idle. The timers thus deferred will be handled when processor eventually comes out of idle due to a non-deferrable timer or any other interrupts.

Deferrable timer is implemented using a special state bit in the timer structure, overloaded over one of the fields in the current structure, which maintains the nature of the timer (deferrable or not). This state bit is preserved as the timer moves around in various per-CPU queues and `__next_timer_interrupt()` skips over all the deferrable timers and picks the timer event for next non-deferrable timer in the timer queue.

This API works very well with ondemand, reducing the number of interrupts and number of events on an idle system as shown in the Table 2. As shown, this feature nearly doubles the average CPU idle residency time, on a totally idle system. This API may also have limited usages with other timers inside the kernel like machine check or cache reap timers and has the potential to reduce the number of wakeups on an idle system further down.

Note that this timer is only available for in kernel usage at this point and usage for user apps is not yet conceptualized. It is not straight forward to extend this to userspace as user timer are not directly associated with per-CPU timer queues and also there can be various dependencies across multiple timers, which can result in timer reordering depending on what CPU they are scheduled on and whether that CPU is idle or not.

4 Platform Timer Event Sources

Dynticks depends on having a per-CPU timer event source. On x86, LAPIC timer can be used as a dependable timer event source. But, when the platform supports low power processor idle states (ACPI C2, C3 states), on most current platforms LAPIC timer stops ticking while CPU is in one of the low power idle states. Dynticks uses a nice workaround to address this issue, with the concept of broadcast timer. Broadcast timer is an always working timer, that will be shared across a

	# interrupts	#events	Avg CPU idle residency (uS)
Ondemand	118	60.60	10161
Ondemand + deferrable timer	89	17.17	20312

Table 2: System activity during idle with and without deferrable timer usage in ondemand

pool of processors and has the responsibility to send a local APIC timer interrupt to wakeup any processor in the pool. Such broadcast timers are platform timers like PIT or HPET [3].

PIT/8254: is a platform timer that can run either in one-shot or periodic mode. It has a frequency of 1193182 Hz and can have a maximum timeout of 27462 uS.

HPET: Is based on newer standard, has a set of memory mapped timers. These timers are programmable to work in different modes and frequency of this timer is based on the particular hardware. On our system under test, HPET runs at 14318179 Hz freq and can have a max timeout of more than 3 seconds (with 32-bit timer).

HPET is superior than PIT, in terms of max timeout value and thus can reduce the number of interrupts when the system is idle. Most of the platforms today has built in HPET timers in the chipset. Unfortunately, very few of the platforms today enable HPET timer and/or advertise the existence of HPET timer to OS using the ACPI tables. As a result, Linux kernel ends up using PIT for broadcast timer. This is not an issue on platforms which do not support low power idle state (e.g., today's servers) as they have always working local APIC time. But, this does cause issue on laptops that typically support low power idle states.

4.1 Force detection of HPET

To resolve this issue of BIOS not enabling/advertising HPET, there is a kernel feature to forcefully detect and enable HPET on various chipsets, using chipset specific PCI quirks. Once that is done, HPET can be used instead of PIT to reduce the number of interrupts on an idle system further. On our test platform, data that we got after forcefully detecting and enabling HPET timer is in Table 3.

Note that this patch to force enable HPET was in proposal-review state at the time of writing this paper and was not available in any standard kernel yet.

4.2 HPET as a per-CPU timer

Linux kernel uses "legacy replacement" mode of HPET timer today to generate timer events. In this mode, HPET appears like legacy PIT and RTC to OS generating interrupts on IRQ0 and IRQ8 for HPET channel 0 and channel 1 timer respectively. There is a further optimization possible, where HPET can be programmed in "standard" interrupt delivery mode and use different channels of HPET to send "per-CPU" interrupt to different processors. This will help laptops that have 2 logical CPUs and at least 2 HPET timer channels available. Different channels of HPET can be used to program timer for each CPU, thereby avoiding the need for broadcast timer altogether and eliminating the LAPIC timers as well. This feature, which is still under development at the time of writing this paper, brings an incremental benefit on systems with more than one logical CPUs and that support deep idle states (most laptop systems with dual core processors). Table 4 shows the data with and without this feature on such a system.

In comparison to base tickless data (as shown in Table 1), features we have talked so far (as shown in Tables 2, 3, and 4) have demonstrated the increase in idle residency time by approximately 7 times.

5 Idle Process Load balancing

In the regular kernel, one of the usages of the periodic timer tick on each processor is to perform periodic process load balancing on that particular processor. If there is a process load imbalance, load balancer will pull the process load from a busiest CPU, ensuring that the load is equally distributed among the available processors in the system. Load balancing period and the subset of processors which will participate in the load balancing will depend on different factors, like the busy state of the processor doing the load balance, the hierarchy of scheduler domains that this processor is part of and the

	# interrupts	#events	Avg CPU idle residency (uS)
PIT	89	17.17	20312
HPET	32	15.15	56451

Table 3: System activity during idle with PIT Vs HPET

	# interrupts	#events	Avg CPU idle residency (uS)
global HPET	32	15.15	56451
percpu HPET	22	15.15	73514

Table 4: System activity during idle with global vs. per-CPU HPET channels

Garbage collector	Perf Regression
parallel	6.3%
gencon	7.7%

Table 5: SPECjbb2000 performance regression with Tickless kernel. Tickless Idle load balancing enhancements recovered this performance regression.

process load in the system. Compared to busy CPUs, idle CPUs perform load balancing more often (mainly because it has nothing else to do and can immediately start executing the pulled load, the load otherwise was waiting for CPU cycles on another busy CPU). In addition to the fairness, this will help improve the system throughput. As such, idle load balancing plays a very important role.

Because of the absence of the periodic timer tick in tickless kernel, idle CPU will potentially sleep for longer time. This extended sleep will delay the periodic load balancing and as such the idle load balancing in the system doesn't happen as often as it does in the earlier kernels. This will present a throughput and latency issue, especially for server workloads.

To measure the performance impact, some experiments were conducted on an 8 CPU core system (dual package system with quad-core) using SPECjbb2000 benchmark in a 512MB heap and 8 warehouses configuration. Performance regression with tickless 2.6.21-rc6-rt0 kernel [5] is shown in Table 5.

Recovering this performance regression in the tickless kernel with no impact on power savings is tricky and challenging. There were some discussions happened in the Linux kernel community on this topic last year, where two main approaches were discussed.

First approach is to increase the back off interval of periodic idle load balancing. Regular Linux kernel already does some sort of backoff (increasing the load balance period up to a maximum amount) when the CPU doing the load balance at a particular sched domain finds that the load at that level is already balanced. And if there is an imbalance, the periodic interval gets reset to the minimum level. Different sched domains in the scheduler domain hierarchy uses different minimum and maximum busy/idle intervals and this back off period increases as one goes up in the scheduler domain hierarchy. Current back off intervals are selected in such a fashion that there are not too many or too less load balancing attempts, so that there is no overdoing the work when the system is well balanced and also react in reasonable amount of time, when the load changes in the system.

To fix the performance regression, this approach suggests to further increase the backoff interval for all the levels in the scheduler domain hierarchy but still retaining the periodic load balancing on each CPU (by registering a new periodic timer event which will trigger the periodic load balancing). Defining the interval increase will be tricky and if it is too much, then the response time will also be high and won't be able to respond for sudden changes in the load. If it is small, then it won't be able to save power, as the periodic load balancing will wake up the idle CPU often.

Second mechanism is some sort of a watchdog mech-

anism where the busy CPU will trigger the load balance on an idle CPU. This mechanism will be making changes to the busy load balancing (which will be doing more load balancing work, while the current busy task on that CPU is eagerly waiting for the CPU cycles). Busy load balancing is quite infrequent compared to idle load balancing attempts. Similar to the first mechanism, this mechanism also won't be able to respond quickly to changes in load. And also figuring out that a CPU is heavily loaded and where that extra load need to moved, is some what difficult job, especially so in the case of hierarchical scheduler domains.

This paper proposes a third route which nominates an owner among the idle CPUs, which does the idle load balancing (ILB) on behalf of the other idle CPUs in the system. This ILB owner will have the periodic tick active in idle state and uses this tick to do load balancing on behalf of all the idle CPUs that it tracks, while the other idle CPUs will be in long tickless sleeps. If there is an imbalance, ILB owner will wakeup the appropriate CPU from its idle sleep. Once all the CPUs in the system are in idle state, periodic tick on the ILB owner will also stop (as there is no other CPU generating load and hence no reason for checking the load imbalance). New idle load balancing owner will be selected again, as soon as there is a busy CPU in the system.

This solution is in 2.6.21 -mm kernels and the experiments showed that this patch completely recovered the performance regression seen earlier (Table 5) in the tickless kernels with SPECjbb workload. If this ILB owner selection is done carefully (like an idle core in a busy package), one can minimize the power wasted also.

6 Keeping Userspace Quiet

Tickless idle kernel alone is not sufficient to enable the idle processor to go into long and deep sleeps. In addition to the kernel space, applications in the user space also need to quite down during idle periods, which will ensure that the whole system goes to long sleeps, ultimately saving power (and thus enhancing battery life in case of laptops).

Dave Jones' OLS2006 talk [9] entitled "Why Userspace Sucks" revealed to the Linux community that the userspace really doesn't quiet down as one would hope for, on an otherwise idle system. Number of applications and daemons wakeup at frequent intervals (even on a completely idle system) for performing a periodic activity like polling a device, cursor blinking, querying for a status change to modify the graphical icon accordingly and so on (for more information about the mischeivous application behaviors look into references [9, 10, 2]).

A Number of fixes went into applications and libraries over the course of the last year to fix these problems [2]. Instead of polling periodically for checking status changes, applications and deamons should use some sort of event notification where ever possible and perform the actions based on the triggered event. For example, the `hal` daemon used to poll very frequently to check for media changes and thus making the idle processor wakeup often. Newer SATA hardware supports a feature called Asynchronous Notification, which will notify the host at a media change event. With the recent changes in the community, `hal` daemon will avoid the polling on platforms which has the support of this asynchronous notification.

Even in the case where the application has to rely on periodic timers, application should use intelligent mechanisms to avoid/minimize the periodic timers when possible. Instead of having scattered timers across the period of time, it will be best to group them and expire the bunch of timers at the same instance. This grouping will minimize the number of instances a processor will be wokenup, while still servicing the same number of timers. This will enable the processor to sleep longer and go into the lowest power state that is possible.

For example all gnome applications use the glib timer `g_timeout_add()` API for their timers, which expire at scattered instances. A second API, `g_timeout_add_seconds()` has now been added which causes all recurring timers to happen at the start of the second, enabling the system wide grouping of timers. The start of the second is offset by a value which is system wide but system-specific to prevent all Linux machines on the internet doing network traffic at the same time.

New tools are getting developed [10] for identifying the applications (and even the kernel level components) which behave badly by wakingup more often than required. These tools use the kernel interfaces (like `/proc/timer_stats`, `/proc/acpi/processor/CPU*/power`, `/proc/interrupts`) and report the biggest offenders and also the average C-state residency information of the processor. Developers should use these tools to identify if their application is

on the hitlist and if so, fix them based on the above mentioned guidelines.

7 Conclusions

Tickless kernel infrastructure is the first and important step forward in making the idle system go into long and deep sleeps. Now that this is integrated into Linux kernel, enhancements mentioned in this paper will increase the mileage out of the tickless kernel, by minimizing the unnecessary wakeups in an otherwise idle system. Going forward, responsibility of saving power lies with both system and user level software. As an evolutionary step, in coming days we can expect the Linux kernel to be fully (both during idle and busy) dynamic tick capable.

8 Acknowledgments

Thanks to all the Linux community members who contributed, reviewed and provided feedback to various tickless add on features mentioned in this paper.

References

[1] Deferrable timers. http://lwn.net/Articles/228143/.

[2] Dependency tree for bug 204948: Userspace sucks (wakeups). https://bugzilla.redhat.com/bugzilla/showdependencytree.cgi?id=204948.

[3] High precision event timers specification. http://www.intel.com/technology/architecture/hpetspec.htm.

[4] Linux 2.6.21. http://www.kernel.org.

[5] Realtime preempt patches. http://people.redhat.com/mingo/realtime-preempt/.

[6] Round jiffies infrastructure. http://lkml.org/lkml/2006/10/10/189.

[7] Thomas Gleixner and Ingo Molnar. Dynamic ticks. http://lwn.net/Articles/202319/.

[8] Thomas Gleixner and Douglas Neihaus. High resolution timers ols 2006. https://ols2006.108.redhat.com/reprints/gleixner-reprint.pdf.

[9] Dave Jones. Why userspace sucks ols 2006. https://ols2006.108.redhat.com/reprints/jones-reprint.pdf.

[10] Arjan Van De Ven. Programming for low power usage. http://conferences.oreillynet.com/cs/os2007/view/e_sess/12958.