Hrtimers and Beyond: Transforming the Linux Time Subsystems

Thomas Gleixner
linutronix
tglx@linutronix.de

Douglas Niehaus
University of Kansas
niehaus@eecs.ku.edu

Abstract

Several projects have tried to rework Linux time and timers code to add functions such as high-precision timers and dynamic ticks. Previous efforts have not been generally accepted, in part, because they considered only a subset of the related problems requiring an integrated solution. These efforts also suffered significant architecture dependence creating complexity and maintenance costs. This paper presents a design which we believe provides a generally acceptable solution to the complete set of problems with minimum architecture dependence.

The three major components of the design are hrtimers, John Stulz's new timeofday approach, and a new component called clock events. Clock events manages and distributes clock events and coordinates the use of clock event handling functions. The hrtimers subsystem has been merged into Linux 2.6.16. Although the name implies "high resolution" there is no change to the tick based timer resolution at this stage. John Stultz's timeofday rework addresses jiffy and architecture independent time keeping and has been identified as a fundamental preliminary for high resolution timers and tickless/dynamic tick solutions. This paper provides details on the hrtimers implementation and describes how the clock events component will complement and complete the hrtimers and timeofday components to create a solid foundation for architecture independent support of high-resolution timers and dynamic ticks.

1 Introduction

Time keeping and use of clocks is a fundamental aspect of operating system implementation, and thus of Linux. Clock related services in operating systems fall into a number of different categories:

- time keeping
- clock synchronization
- time-of-day representation
- next event interrupt scheduling
- process and in-kernel timers
- process accounting
- process profiling

These service categories exhibit strong interactions among their semantics at the design level and tight coupling among their components at the implementation level.

Hardware devices capable of providing clock sources vary widely in their capabilities, accuracy, and suitability for use in providing the desired clock services. The ability to use a given hardware device to provide a particular clock service also varies with its context in a uniprocessor or multi-processor system.

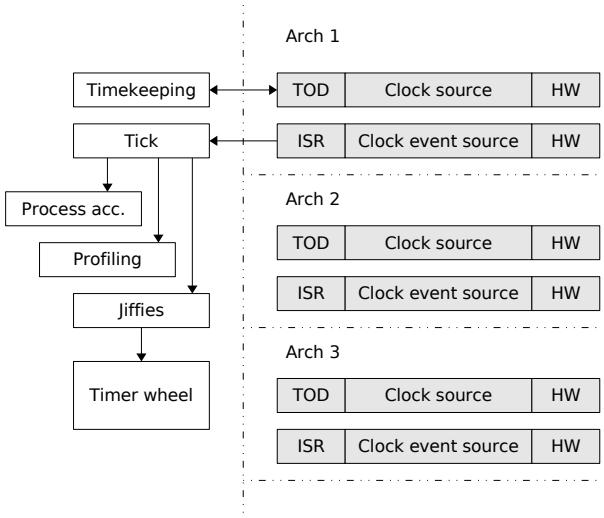

Figure 1: Linux Time System

Figure 1 shows the current software architecture of the clock related services in a vanilla 2.6 Linux system. The current implementation of clock related services in Linux is strongly associated with individual hardware devices, which results in parallel implementations for each architecture containing considerable amounts of essentially similar code. This code duplication across a large number of architectures makes it difficult to change the semantics of the clock related services or to add new features such as high resolution timers or dynamic ticks because even a simple change must be made in so many places and adjusted for so many implementations. Two major factors make implementing changes to Linux clock related services difficult: (1) the lack of a generic abstraction layer for clock services and (2) the assumption that time is tracked using periodic timer ticks (jiffies) that is strongly integrated into much of the clock and timer related code.

2 Previous Efforts

A number of efforts over many years have addressed various clock related services and functions in Linux including various approaches to high resolution time keeping and event scheduling. However, all of these efforts have encountered significant difficulty in gaining broad acceptance because of the breadth of their impact on the rest of the kernel, and because they generally addressed only a subset of the clock related services in Linux.

Interestingly, all those efforts have a common design pattern, namely the attempt to integrate new features and services into the existing clock and timer infrastructure without changing the overall design.

There are no projects to our knowledge which attempt to solve the *complete* problem as we understand and have described it. All existing efforts, in our view, address only a part of the whole problem as we see it, which is why, in our opinion, the solutions to their target problems are more complex than under our proposed architecure, and are thus less likely to be accepted into the main line kernel.

3 Required Abstractions

The attempt to integrate high resolution timers into Ingo Molnar's real-time preemption patch led to a thorough analysis of the Linux timer and clock services infrastructure. While the comprehensive solution for addressing the overall problem is a large-scale task it can be separated into different problem areas.

- clock sources management for time keeping

- clock synchronization
- time-of-day representation
- clock event management for scheduling next event interrupts
- eliminating the assumption that timers are supported by periodic interrupts and expressed in units of jiffies

These areas of concern are largely independent and can thus be addressed more or less independently during implementation. However, the important points of interaction among them must be considered and supported in the overall design.

3.1 Clock Source Management

An abstraction layer and associated API are required to establish a common code framework for managing various clock sources. Within this framework, each clock source is required to maintain a representation of time as a monotonically increasing value. At this time, nanoseconds are the favorite choice for the time value units of a clock source. This abstraction layer allows the user to select among a range of available hardware devices supporting clock functions when configuring the system and provides necessary infrastructure. This infrastructure includes, for example, mathematical helper functions to convert time values specific to each clock source, which depend on properties of each hardware device, into the common human-oriented time units used by the framework, i.e. nanoseconds. The centralization of this functionality allows the system to share significantly more code across architectures. This abstraction is already addressed by John Stultz's work on a Generic Time-of-day subsystem [5].

3.2 Clock Synchronization

Crystal driven clock sources tend to be imprecise due to variation in component tolerances and environmental factors, such as temperature, resulting in slightly different clock tick rates and thus, over time, different clock values in different computers. The Network Time Protocol (NTP) and more recently GPS/GSM based synchronization mechanisms allow the correction of system time values and of clock source drift with respect to a selected standard. Value correction is applied to the monotonically increasing value of the hardware clock source. This is an optional functionality as it can only be applied when a suitable reference time source is available. Support for clock synchronization is a separate component from those discussed here. There is work in progress to rework the current mechanism by John Stultz and Roman Zippel.

3.3 Time-of-day Representation

The monotonically increasing time value provided by many hardware clock sources cannot be set. The generic interface for time-of-day representation must thus compensate for drift as an offset to the clock source value, and represent the time-of-day (calendar or wall clock time) as a function of the clock source value. The drift offset and parameters to the function converting the clock source value to a wall clock value can set by manual interaction or under control of software for synchronization with external time sources (e.g. NTP).

It is important to note that the current Linux implementation of the time keeping component is the reverse of the proposed solution. The internal time representation tracks the time-of-day time fairly directly and derives the monotonically increasing nanosecond time value from it.

This is a relic of software development history and the GTOD/NTP work is already addressing this issue.

3.4 Clock Event Management

While clock sources provide read access to the monotonically increasing time value, clock event sources are used to schedule the next event interrupt(s). The next event is currently defined to be periodic, with its period defined at compile time. The setup and selection of the event sources for various event driven functionalities is hardwired into the architecture dependent code. This results in duplicated code across all architectures and makes it extremely difficult to change the configuration of the system to use event interrupt sources other than those already built into the architecture. Another implication of the current design is that it is necessary to touch all the architecture-specific implementations in order to provide new functionality like high resolution timers or dynamic ticks.

The clock events subsystem tries to address this problem by providing a generic solution to manage clock event sources and their usage for the various clock event driven kernel functionalities. The goal of the clock event subsystem is to minimize the clock event related architecture dependent code to the pure hardware related handling and to allow easy addition and utilization of new clock event sources. It also minimizes the duplicated code across the architectures as it provides generic functionality down to the interrupt service handler, which is almost inherently hardware dependent.

3.5 Removing Tick Dependencies

The strong dependency of Linux timers on using the the periodic tick as the time source and representation was one of the main problems faced when implementing high resolution timers and variable interval event scheduling. All attempts to reuse the cascading timer wheel turned out to be incomplete and inefficient for various reasons. This led to the implementation of the *hrtimers* (former ktimers) subsystem. It provides the base for precise timer scheduling and is designed to be easily extensible for high resolution timers.

4 hrtimers

The current approach to timer management in Linux does a good job of satisfying an extremely wide range of requirements, but it cannot provide the quality of service required in some cases precisely because it must satisfy such a wide range of requirements. This is why the essential first step in the approach described here is to implement a new timer subsystem to complement the existing one by assuming a subset of its existing responsibilities.

4.1 Why a New Timer Subsystem?

The Cascading Timer Wheel (CTW), which was implemented in 1997, replaced the original time ordered double linked list to resolve the scalability problem of the linked list's O(N) insertion time. It is based on the assumption that the timers are supported by a periodic interrupt (jiffies) and that the expiration time is also represented in jiffies. The difference in time value (delta) between now (the current system time) and the timer's expiration value is used as an index into the CTW's logarithmic array of arrays. Each array within the CTW represents the set of timers placed within a region of the system time line, where the size of the array's regions grow exponentially. Thus, the further into the

array	start	end	granularity
1	1	256	1
2	257	16384	256
3	16385	1048576	16384
4	1048577	67108864	1048576
5	67108865	4294967295	67108864

Table 1: Cascading Timer Wheel Array Ranges

future a timer's expiration value lies, the larger the region of the time line represented by the array in which it is stored. The CTW groups timers into 5 categories. Note that each CTW array represents a range of jiffy values and that more than one timer can be associated with a given jiffy value.

Table 1 shows the properties of the different timer categories. The first CTW category consists of n1 entries, where each entry represents a single jiffy. The second category consists of n2 entries, where each entry represents n1*n2 jiffies. The third category consists of n3 entries, where each entry represents n1*n2*n3 jiffies. And so forth. The current kernel uses n1=256 and n2..n5 = 64. This keeps the number of hash table entries in a reasonable range and covers the future time line range from 1 to 4294967295 jiffies.

The capacity of each category depends on the size of a jiffy, and thus on the periodic interrupt interval. While the 10 ms tick period in 2.4 kernels implied 2560ms for the CTW first category, this was reduced to 256ms in the early 2.6 kernels (1 ms tick) and readjusted to 1024ms when the HZ value was set to 250. Each CTW category maintains an time index counter which is incremented by the "wrapping" of the lower category index which occurs when its counter increases to the point where its range overlaps that of the higher category. This triggers a "cascade" where timers from the matching entry in the higher category have to be removed and reinserted into the lower category's finer-grained entries. Note that in the first CTW category the timers are time-sorted with jiffy resolution.

While the CTW's O(1) insertion and removal is very efficient, timers with an expiration time larger than the capacity of the first category have to be cascaded into a lower category at least once. A single step of cascading moves many timers and it has to be done with interrupts disabled. The cascading operation can thus significantly increase maximum latencies since it occasionally moves very large sets of timers. The CTW thus has excellent average performance but unacceptable worst case performance. Unfortunately the worst case performance determines its suitability for supporting high resolution timers.

However, it is important to note that the CTW is an excellent solution (1) for timers having an expiration time lower than the capacity of the primary category and (2) for timers which are removed before they expire or have to be cascaded. This is a common scenario for many long-term protocol-timeout related timers which are created by the networking and I/O subsystems.

The KURT-Linux project at the University of Kansas was the first to address implementing high resolution timers in Linux [4]. Its concentration was on investigating various issues related to using Linux for real-time computing. The UTIME component of KURT-Linux experimented with a number of data structures to support high resolution timers, including both separate implementations and those integrated with general purpose timers. The HRT project began as a fork of UTIME code [1]. both projects added a sub-jiffy time tracking component to increase resolution, and when integrating support with the CTW, sorted timers within a given jiffy on the basis of the subjiffy value.

This increased overhead involved with cascading due to the O(N) sorting time. The experience of both projects demonstrated that timer management overhead was a significant factor, and that the necessary changes in the timer code were quite scattered and intrusive. In summary, the experience of both projects demonstrated that separating support for high resolution and longer-term generic (CTW) timers was necessary and that a comprehensive restructuring of the timer-related code would be required to make future improvements and additions to timer-related functions possible. The *hrtimers* design and other aspects of the architecture described in this paper was strongly motivated by the lessons derived from both previous projects.

4.2 Solution

As a first step we categorized the timers into two categories:

Timeouts: Timeouts are used primarily by networking and device drivers to detect when an event (I/O completion, for example) does not occur as expected. They have low resolution requirements, and they are almost always removed before they actually expire.

Timers: Timers are used to schedule ongoing events. They can have high resolution requirements, and usually expire. Timers are mostly related to applications (user space interfaces)

The timeout related timers are kept in the existing timer wheel and a new subsystem optimized for (high resolution) timer requirements *hrtimers* was implemented.

hrtimers are entirely based on human time units: nanoseconds. They are kept in a time sorted, per-CPU list, implemented as a red-black tree. Red-black trees provide O(log(N)) insertion and removal and are considered to be efficient enough as they are already used in other performance critical parts of the kernel e.g. memory management. The timers are kept in relation to time bases, currently `CLOCK_MONOTONIC` and `CLOCK_REALTIME`, ordered by the absolute expiration time. This separation allowed to remove large chunks of code from the POSIX timer implementation, which was necessary to recalculate the expiration time when the clock was set either by settimeofday or NTP adjustments.

hrtimers went through a couple of revision cycles and were finally merged into Linux 2.6.16. The timer queues run from the normal timer softirq so the resulting resolution is not better than the previous timer API. All of the structure is there to do better once the other parts of the overall timer code rework are in place.

After adding *hrtimers* the Linux time(r) system looks like this:

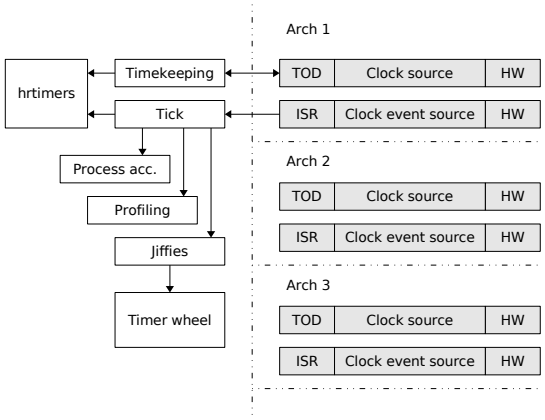

Figure 2: Linux time system + htimers

4.3 Further Work

The primary purpose of the separate implementation for the high resolution timers, dis-

cussed in Section 7, is to improve their support by eliminating the overhead and variable latency associated with the CTW. However, it is also important to note that this separation also creates an opportunity to improve the CTW behavior in supporting the remaining timers. For example, using a coarser CTW granularity may lower overhead by reducing the number of timers which are cascaded, given that an even larger percentage of CTW timers would be canceled under an architecture supporting high resolution timers separately. However, while this is an interesting possibility, it is currently a speculation that must be tested.

5 Generic Time-of-day

The Generic Time-of-day subsystem (GTOD) is a project led by John Stultz and was presented at OLS 2005. Detailed information is available from the OLS 2005 proceedings [5]. It contains the following components:

- Clock source management
- Clock synchronization
- Time-of-day representation

GTOD moves a large portion of code out of the architecture-specific areas into a generic management framework, as illustrated in Figure 3. The remaining architecture-dependent code is mostly limited to the direct hardware interface and setup procedures. It allows simple sharing of clock sources across architectures and allows the utilization of non-standard clock source hardware. GTOD is work in progress and intends to produce set of changes which can be adopted step by step into the main-line kernel.

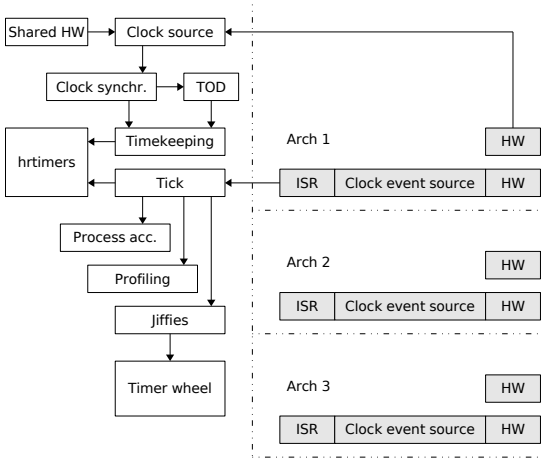

Figure 3: Linux time system + htimers + GTOD

6 Clock Event Source Abstraction

Just as it was necessary to provide a general abstraction for clock sources in order to move a significant amount of code into the architecture independent area, a general framework for managing clock event sources is also required in the architecture independent section of the source under the architecture described here. Clock event sources provide either periodic or individual programmable events. The management layer provides the infrastructure for registering event sources and manages the distribution of the events for the various clock related services. Again, this reduces the amount of essentially duplicate code across the architectures and allows cross-architecture sharing of hardware support code and the easy addition of non-standard clock sources.

The management layer provides interfaces for *hrtimers* to implement high resolution timers and also builds the base for a generic dynamic tick implementation. The management layer supports these more advanced functions only when appropriate clock event sources have been registered, otherwise the traditional periodic tick based behavior is retained.

6.1 Clock Event Source Registration

Clock event sources are registered either by the architecture dependent boot code or at module insertion time. Each clock event source fills a data structure with clock-specific property parameters and callback functions. The clock event management decides, by using the specified property parameters, the set of system functions a clock event source will be used to support. This includes the distinction of per-CPU and per-system global event sources.

System-level global event sources are used for the Linux periodic tick. Per-CPU event source are used to provide local CPU functionality such as process accounting, profiling, and high resolution timers. The `clock_event` data structure contains the following elements:

- `name`: clear text identifier
- `capabilities`: a bit-field which describes the capabilities of the clock event source and hints about the preferred usage
- `max_delta_ns`: the maximum event delta (offset into future) which can be scheduled
- `min_delta_ns`: the minimum event delta which can be scheduled
- `mult`: multiplier for scaled math conversion from nanoseconds to clock event source units
- `shift`: shift factor for scaled math conversion from nanoseconds to clock event source units
- `set_next_event`: function to schedule the next event
- `set_mode`: function to toggle the clock event source operating mode (periodic / one shot)
- `suspend`: function which has to be called before suspend
- `resume`: function which has to be called before resume
- `event_handler`: function pointer which is filled in by the clock event management code. This function is called from the event source interrupt
- `start_event`: function called before the `event_handler` function in case that the clock event layer provides the interrupt handler
- `end_event`: function called after the `event_handler` function in case that the clock event layer provides the interrupt handler
- `irq`: interrupt number in case the clock event layer requests the interrupt and provides the interrupt handler
- `priv`: pointer to clock source private data structures

The clock event source can delegate the interrupt setup completely to the management layer. It depends on the type of interrupt which is associated with the event source. This is possible for the PIT on the i386 architecture, for example, because the interrupt in question is handled by the generic interrupt code and can be initialized via setup_irq. This allows us to completely remove the timer interrupt handler from the i386 architecture-specific area and move the modest amount of hardware-specific code into appropriate source files. The hardware-specific routines are called before and after the event handling code has been executed.

In case of the Local APIC on i386 and the Decrementer on PPC architectures, the interrupt handler must remain in the architecture-specific code as it can not be setup through the standard interrupt handling functions. The clock management layer provides the function which has to be called from the hardware level

handler in a function pointer in the clock source description structure. Even in this case the shared code of the timer interrupt is removed from the architecture-specific implementation and the event distribution is managed by the generic clock event code. The Clock Events subsystem also has support code for clock event sources which do not provide a periodic mode; e.g. the Decrementer on PPC or match register based event sources found in various ARM SoCs.

6.2 Clock Event Distribution

The clock event layer provides a set of predefined functions, which allow the association of various clock event related services to a clock event source.

The current implementation distributes events for the following services:

- periodic tick
- process accounting
- profiling
- next event interrupt (*e.g.* high resolution timers, dynamic tick)

6.3 Interfaces

The clock event layer API is rather small. Aside from the clock event source registration interface it provides functions to schedule the next event interrupt, clock event source notification service, and support for suspend and resume.

6.4 Existing Implementations

At the time of this writing the base framework code and the conversion of i386 to the clock event layer is available and functional.

The clock event layer has been successfully ported to ARM and PPC, but support has not been continued due to lack of human resources.

6.5 Code Size Impact

The framework adds about 700 lines of code which results in a 2KB increase of the kernel binary size.

The conversion of i386 removes about 100 lines of code. The binary size decrease is in the range of 400 bytes.

We believe that the increase of flexibility and the avoidance of duplicated code across architectures justifies the slight increase of the binary size.

The first goal of the clock event implementation was to prove the feasibility of the approach. There is certainly room for optimizing the size impact of the framework code, but this is an issue for further development.

6.6 Further Development

The following work items are planned:

- Streamlining of the code
- Revalidation of the clock distribution decisions
- Support for more architectures
- Dynamic tick support

6.7 State of Transformation

The clock event layer adds another level of abstraction to the Linux subsystem related to time keeping and time-related activities, as illustrated in Figure 4. The benefit of adding the abstraction layer is the substantial reduction in architecture-specific code, which can be seen most clearly by comparing Figures 3 and 4.

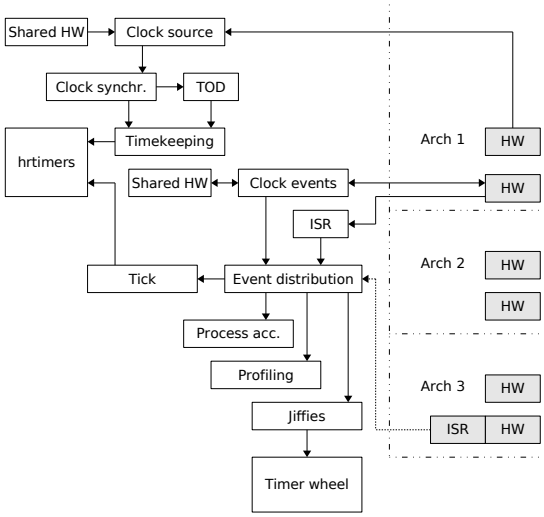

Figure 4: Linux time system + htimers + GTOD + clock events

7 High Resolution Timers

The inclusion of the clock source and clock event source management and abstraction layers provides now the base for high resolution support for *hrtimers*.

While previous attempts of high resolution timer implementations needed modification all over the kernel source tree, the *hrtimers* based implementation only changes the *hrtimers* code itself. The required change to enable high resolution timers for an architecture which is supported by the Generic Time-of-day and the clock event framework is the inclusion of a single line in the architecture specific Kconfig file.

The next event modifications remove the implicit but strong binding of hrtimers to jiffy tick boundaries. When the high resolution extension is disabled the clock event distribution code works in the original periodic mode and hrtimers are bound to jiffy tick boundaries again.

8 Implementation

While the base functionality of *hrtimers* remains unchanged, additional functionality had to be added.

- Management function to switch to high resolution mode late in the boot process.
- Next event scheduling
- Next event interrupt handler
- Separation of the *hrtimers* queue from the timer wheel softirq

During system boot it is not possible to use the high resolution timer functionality, while making it possible would be difficult and would serve no useful function. The initialization of the clock event framework, the clock source framework and *hrtimers* itself has to be done and appropriate clock sources and clock event sources have to be registered before the high resolution functionality can work. Up to the point where hrtimers are initialized, the system works in the usual low resolution periodic mode. The clock source and the clock event source layers provide notification functions which inform *hrtimers* about availability of new hardware. *hrtimers* validates the usability of the registered clock sources and clock

event sources before switching to high resolution mode. This ensures also that a kernel which is configured for high resolution timers can run on a system which lacks the necessary hardware support.

The time ordered insertion of *hrtimers* provides all the infrastructure to decide whether the event source has to be reprogrammed when a timer is added. The decision is made per timer base and synchronized across timer bases in a support function. The design allows the system to utilize separate per-CPU clock event sources for the per-CPU timer bases, but mostly only one reprogrammable clock event source per-CPU is available. The high resolution timer does not support SMP machines which have only global clock event sources.

The next event interrupt handler is called from the clock event distribution code and moves expired timers from the red-black tree to a separate double linked list and invokes the softirq handler. An additional mode field in the *hrtimer* structure allows the system to execute callback functions directly from the next event interrupt handler. This is restricted to code which can safely be executed in the hard interrupt context and does not add the timer back to the red-black tree. This applies, for example, to the common case of a wakeup function as used by nanosleep. The advantage of executing the handler in the interrupt context is the avoidance of up to two context switches—from the interrupted context to the softirq and to the task which is woken up by the expired timer. The next event interrupt handler also provides functionality which notifies the clock event distribution code that a requested periodic interval has elapsed. This allows to use a single clock event source to schedule high resolution timer and periodic events e.g. jiffies tick, profiling, process accounting. This has been proved to work with the PIT on i386 and the Incrementer on PPC.

The softirq for running the *hrtimer* queues and executing the callbacks has been separated from the tick bound timer softirq to allow accurate delivery of high resolution timer signals which are used by itimer and POSIX interval timers. The execution of this softirq can still be delayed by other softirqs, but the overall latencies have been significantly improved by this separation.

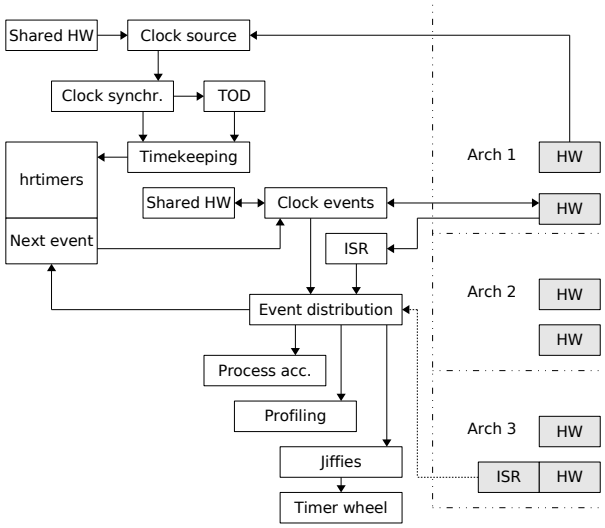

Figure 5: Linux time system + htimers + GTOD + clock events + high resolution timers

8.1 Accuracy

All tests have been run on a Pentium III 400MHz based PC. The tables show comparisons of vanilla Linux 2.6.16, Linux-2.6.16-hrt5 and Linux-2.6.16-rt12. The tests for intervals less than the jiffy resolution have not been run on vanilla Linux 2.6.16. The test thread runs in all cases with SCHED_FIFO and priority 80.

Test case: `clock_nanosleep(TIME_ABSTIME)`, Interval 10000 microseconds, 10000 loops, no load.

Kernel	min	max	avg
2.6.16	24	4043	1989
2.6.16-hrt5	12	94	20
2.6.16-rt12	6	40	10

Test case: `clock_nanosleep(TIME_ABSTIME)`, Interval 10000 micro seconds, 10000 loops, 100% load.

Kernel	min	max	avg
2.6.16	55	4280	2198
2.6.16-hrt5	11	458	55
2.6.16-rt12	16		

Test case: POSIX interval timer, Interval 10000 micro seconds, 10000 loops, no load.

Kernel	min	max	avg
2.6.16	21	4073	2098
2.6.16-hrt5	22	120	35
2.6.16-rt12	20	60	31

Test case: POSIX interval timer, Interval 10000 micro seconds, 10000 loops, 100% load.

Kernel	min	max	avg
2.6.16	82	4271	2089
2.6.16-hrt5	31	458	53
2.6.16-rt12	21	70	35

Test case: `clock_nanosleep(TIME_ABSTIME)`, Interval 500 micro seconds, 100000 loops, no load.

Kernel	min	max	avg
2.6.16-hrt5	5	108	24
2.6.16-rt12	5	48	7

Test case: `clock_nanosleep(TIME_ABSTIME)`, Interval 500 micro seconds, 100000 loops, 100% load.

Kernel	min	max	avg
2.6.16-hrt5	9	684	56
2.6.16-rt12	10	60	22

Test case: POSIX interval timer, Interval 500 micro seconds, 100000 loops, no load.

Kernel	min	max	avg
2.6.16-hrt5	8	119	22
2.6.16-rt12	12	78	16

Test case: POSIX interval timer, Interval 500 micro seconds, 100000 loops, 100% load.

Kernel	min	max	avg
2.6.16-hrt5	16	489	58
2.6.16-rt12	12	95	29

The real-time preemption kernel results are significantly better under high load due to the general low latencies for high priority real-time tasks. Aside from the general latency optimizations, further improvements were implemented specifically to optimize the high resolution timer behavior.

Separate threads for each softirq. Long lasting softirq callback functions e.g. in the networking code do not delay the delivery of *hrtimer* softirqs.

Dynamic priority adjustment for high resolution timer softirqs. Timers store the priority of the task which inserts the timer and the next event interrupt code raises the priority of the *hrtimer* softirq when a callback function for a high priority thread has to be executed. The softirq lowers its priority automatically after the execution of the callback function.

9 Dynamic Ticks

We have not yet done a dynamic tick implementation on top of the existing framework, but we considered the requirements for such an implementation in every design step.

The framework does not solve the general problem of dynamic ticks: how to find the next expiring timer in the timer wheel. In the worst

case the code has to walk through a large number of hash buckets. This can not be changed without changing the basic semantics and implementation details of the timer wheel code.

The next expiring *hrtimer* is simply retrieved by checking the first timer in the time ordered red-black tree.

On the other hand, the framework will deliver all the necessary clock event source mechanisms to reprogram the next event interrupt and enable a clean, non-intrusive, out of the box, solution once an architecture has been converted to use the framework components.

The clock event functionalities necessary for dynamic tick implementations are available whether the high resolution timer functionality is enabled or not. The framework code takes care of those use cases already.

With the integration of dynamic ticks the transformation of the Linux time related subsystems will become complete, as illustrated in Figure 6.

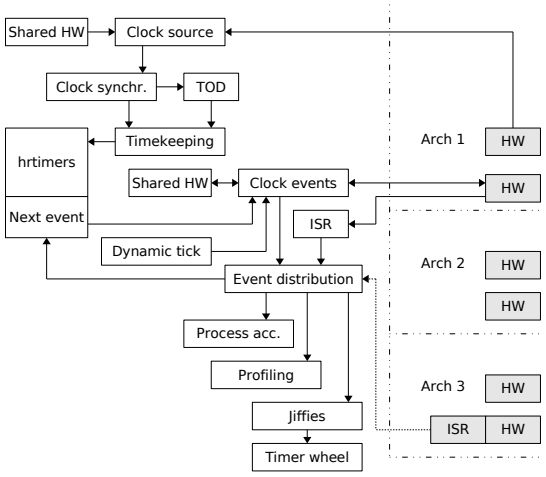

Figure 6: Transformed Linux Time Subsystem

10 Conclusion

The existing parts and pieces of the overall solution have proved that a generic solution for high resolution timers and dynamic tick is feasible and provides a valuable benefit for the Linux kernel.

Although most of the components have been tested extensively in the high resolution timer patch and the real-time preemption patch there is still a way to go until a final inclusion into the mainline kernel can be considered.

In general this can only be achieved by a step by step conversion of functional units and architectures. The framework code itself is almost self contained so a not converted architecture should not have any impacts.

We believe that we provided a clear vision of the overall solution and we hope that more developers get interested and help to bring this further in the near future.

10.1 Acknowledgments

We sincerely thank all those people who helped us to develop this solution. Help has been provided in form of code contributions, code reviews, testing, discussion, and suggestions. Especially we want to thank Ingo Molnar, John Stultz, George Anzinger, Roman Zippel, Andrew Morton, Steven Rostedt and Benedikt Spranger. A special thank you goes to Jonathan Corbet who wrote some excellent articles about hrtimers (and the previous ktimers) implementation [2, 3].

References

[1] George Anzinger and Monta Vista. High resolution timers home page.

http://high-res-timers.
sourceforge.net.

[2] J. Corbet. Lwn article: A new approach to kernel timers. http:
//lwn.net/Articles/152436.

[3] J. Corbet. Lwn article: The high resolution timer api. http:
//lwn.net/Articles/167897.

[4] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the shelf hardware and free software. In 4^{th} *Real-Time Technology and Applications Symposium*, Denver, June 1998.

[5] J. Stulz. We are not getting any younger: A new approach to timekeeping and timers. In *Ottawa Lnux Symposium*, Ottawa, Ontario, Canada, July 2005.