

# [PATCH 4/8] adaptive real-time lock support

---

*Source:* <http://linux.derkeiler.com/Mailing-Lists/Kernel/2008-05/msg08810.html>

---

- *From:* Gregory Haskins <[ghaskins@xxxxxxxxxxx](mailto:ghaskins@xxxxxxxxxxx)>
  - *Date:* Mon, 19 May 2008 13:07:33 -0400
- 

The Real Time patches to the Linux kernel converts the architecture specific SMP-synchronization primitives commonly referred to as "spinlocks" to an "RT mutex" implementation that support a priority inheritance protocol, and priority-ordered wait queues. The RT mutex implementation allows tasks that would otherwise busy-wait for a contended lock to be preempted by higher priority tasks without compromising the integrity of critical sections protected by the lock. The unintended side-effect is that the -rt kernel suffers from significant degradation of IO throughput (disk and net) due to the extra overhead associated with managing pi-lists and context switching. This has been generally accepted as a price to pay for low-latency preemption.

Adaptive real-time lock technology restores some of the benefits lost in the conversion to RT mutexes by adaptively spinning or sleeping while retaining both the priority inheritance protocol as well as the preemptive nature of RT mutexes. Essentially, the RT Mutex has been modified to busy-wait under contention for a limited (and configurable) time. This works because most locks are typically held for very short time spans. Too often, by the time a task goes to sleep on a mutex, the mutex is already being released on another CPU. The effect (on SMP) is that by polling a mutex for a limited time we reduce context switch overhead by up to 90%, and therefore eliminate CPU cycles as well as massive hot-spots in the scheduler / other bottlenecks in the Kernel - even though we busy-wait (using CPU cycles) to poll the lock.

The net result is a significant performance boost (300%+) in many aspects of the -rt kernel. These performance increases apply to disk IO as well as netperf UDP benchmarks, without compromising RT preemption latency. For more complex applications, overall the I/O throughput seems to approach the throughput on a PREEMPT\_VOLUNTARY or PREEMPT\_DESKTOP Kernel, as is shipped by most distros.

Signed-off-by: Gregory Haskins <[ghaskins@xxxxxxxxxxx](mailto:ghaskins@xxxxxxxxxxx)>  
Signed-off-by: Peter Morreale <[pmorreale@xxxxxxxxxxx](mailto:pmorreale@xxxxxxxxxxx)>  
Signed-off-by: Sven Dietrich <[sdietrich@xxxxxxxxxxx](mailto:sdietrich@xxxxxxxxxxx)>

---

Documentation/adaptive-locks.txt | 38 ++++++

## [PATCH 4/8] adaptive real-time lock support

```
kernel/Kconfig.preempt | 21 ++++++++
kernel/rtmutex.c | 15 ++++
kernel/rtmutex_adaptive.h | 106 +++++
4 files changed, 177 insertions(+), 3 deletions(-)
```

```
diff --git a/Documentation/adaptive-locks.txt b/Documentation/adaptive-locks.txt
```

```
new file mode 100644
```

```
index 0000000..bc9abf0
```

```
--- /dev/null
```

```
+++ b/Documentation/adaptive-locks.txt
```

```
@@ -0,0 +1,38 @@
```

```
+ Adaptive real-time locks
```

```
+-----
```

```
+ 
```

```
+The Real Time patches to the Linux kernel converts the architecture  
+specific SMP-synchronization primitives commonly referred to as  
+"spinlocks" to an "RT mutex" implementation that support a priority  
+inheritance protocol, and priority-ordered wait queues. The RT mutex  
+implementation allows tasks that would otherwise busy-wait for a  
+contended lock to be preempted by higher priority tasks without  
+compromising the integrity of critical sections protected by the lock.  
+The unintended side-effect is that the -rt kernel suffers from  
+significant degradation of IO throughput (disk and net) due to the  
+extra overhead associated with managing pi-lists and context switching.  
+This has been generally accepted as a price to pay for low-latency  
+preemption.
```

```
+ 
```

```
+Adaptive real-time lock technology restores some of the benefits lost  
+in the conversion to RT mutexes by adaptively spinning or sleeping  
+while retaining both the priority inheritance protocol as well as the  
+preemptive nature of RT mutexes. Essentially, the RT Mutex has been  
+modified to busy-wait under contention for a limited (and configurable)  
+time. This works because most locks are typically held for very short  
+time spans. Too often, by the time a task goes to sleep on a mutex,  
+the mutex is already being released on another CPU. The effect (on SMP)  
+is that by polling a mutex for a limited time we reduce context switch  
+overhead by up to 90%, and therefore eliminate CPU cycles as well as  
+massive hot-spots in the scheduler / other bottlenecks in the  
+Kernel - even though we busy-wait (using CPU cycles) to poll the lock.
```

```
+ 
```

```
+The net result is a significant performance boost (300%+) in many aspects  
+of the -rt kernel. These performance increases apply to disk IO as well  
+as netperf UDP benchmarks, without compromising RT preemption latency.  
+For more complex applications, overall the I/O throughput seems to  
+approach the throughput on a PREEMPT_VOLUNTARY or PREEMPT_DESKTOP Kernel,  
+as is shipped by most distros.
```

```
+ 
```

```
+ 
```

```
+ 
```

```
diff --git a/kernel/Kconfig.preempt b/kernel/Kconfig.preempt
```

```
index aea638b..45d00dc 100644
```

## [PATCH 4/8] adaptive real-time lock support

```
--- a/kernel/Kconfig.preempt
+++ b/kernel/Kconfig.preempt
@@ -160,3 +160,24 @@ config RCU_TRACE

Say Y here if you want to enable RCU tracing
Say N if you are unsure.
+
+config ADAPTIVE_RTLOCK
+ bool "Adaptive real-time locks"
+ default y
+ depends on PREEMPT_RT && SMP
+ help
+ PREEMPT_RT allows for greater determinism by transparently
+ converting normal spinlock_ts into preemptible rtmutexes which
+ sleep any waiters under contention. However, in many cases the
+ lock will be released in less time than it takes to context
+ switch. Therefore, the "sleep under contention" policy may also
+ degrade throughput performance due to the extra context switches.
+
+ This option alters the rtmutex derived spinlock_t replacement
+ code to use an adaptive spin/sleep algorithm. It will spin
+ unless it determines it must sleep to avoid deadlock. This
+ offers a best of both worlds solution since we achieve both
+ high-throughput and low-latency.
+
+ If unsure, say Y.
+
diff --git a/kernel/rtmutex.c b/kernel/rtmutex.c
index 223557f..f50ec55 100644
--- a/kernel/rtmutex.c
+++ b/kernel/rtmutex.c
@@ -7,6 +7,8 @@
 * Copyright (C) 2005-2006 Timesys Corp., Thomas Gleixner <tglx@xxxxxxxxxxxx>
 * Copyright (C) 2005 Kihon Technologies Inc., Steven Rostedt
 * Copyright (C) 2006 Esben Nielsen
+ * Copyright (C) 2008 Novell, Inc., Sven Dietrich, Peter Morreale,
+ * and Gregory Haskins
 *
 * See Documentation/rt-mutex-design.txt for details.
 */
@@ -17,6 +19,7 @@
#include <linux/hardirq.h>

#include "rtmutex_common.h"
+#include "rtmutex_adaptive.h"

/*
 * lock->owner state tracking:
@@ -689,6 +692,7 @@ rt_spin_lock_slowlock(struct rt_mutex *lock)
{
struct rt_mutex_waiter waiter;
```

[PATCH 4/8] adaptive real-time lock support

```
unsigned long saved_state, state, flags;
+ DECLARE_ADAPTIVE_WAITER(adaptive);

debug_rt_mutex_init_waiter(&waiter);
waiter.task = NULL;
@@ -734,6 +738,8 @@ rt_spin_lock_slowlock(struct rt_mutex *lock)
continue;
}

+ prepare_adaptive_wait(lock, &adaptive);
+
+ /*
+ * Prevent schedule() to drop BKL, while waiting for
+ * the lock ! We restore lock_depth when we come back.
+ */
@@ -745,9 +751,12 @@ rt_spin_lock_slowlock(struct rt_mutex *lock)

debug_rt_mutex_print_deadlock(&waiter);

- update_current(TASK_UNINTERRUPTIBLE, &saved_state);
- if (waiter.task)
- schedule_rt_mutex(lock);
+ /* adaptive_wait() returns 1 if we need to sleep */
+ if (adaptive_wait(lock, &waiter, &adaptive)) {
+ update_current(TASK_UNINTERRUPTIBLE, &saved_state);
+ if (waiter.task)
+ schedule_rt_mutex(lock);
+ }

spin_lock_irqsave(&lock->wait_lock, flags);
current->flags |= saved_flags;
diff --git a/kernel/rtmutex_adaptive.h b/kernel/rtmutex_adaptive.h
new file mode 100644
index 0000000..8329e3c
--- /dev/null
+++ b/kernel/rtmutex_adaptive.h
@@ -0,0 +1,106 @@
+ /*
+ * Adaptive RT lock support
+ *
+ * See Documentation/adaptive-locks.txt
+ *
+ * Copyright (C) 2008 Novell, Inc.,
+ * Sven Dietrich, Peter Morreale, and Gregory Haskins
+ *
+ * This program is free software; you can redistribute it and/or
+ * modify it under the terms of the GNU General Public License
+ * as published by the Free Software Foundation; version 2
+ * of the License.
+ */
+
+
+ #ifndef __KERNEL_RTMUTEX_ADAPTIVE_H
```

[PATCH 4/8] adaptive real-time lock support

```
+ #define __KERNEL_RTMutex_ADAPTIVE_H
+
+ #include "rtmutex_common.h"
+
+
+ #ifdef CONFIG_ADAPTIVE_RTLOCK
+ struct adaptive_waiter {
+     struct task_struct *owner;
+ };
+
+ /*
+  * Adaptive-rtlocks will busywait when possible, and sleep only if
+  * necessary. Note that the busyloop looks racy, and it is....but we do
+  * not care. If we lose any races it simply means that we spin one more
+  * time before seeing that we need to break-out on the next iteration.
+  *
+  * We realize this is a relatively large function to inline, but note that
+  * it is only instantiated 1 or 2 times max, and it makes a measurable
+  * performance difference to avoid the call.
+  *
+  * Returns 1 if we should sleep
+  *
+  */
+ static inline int
+ adaptive_wait(struct rt_mutex *lock, struct rt_mutex_waiter *waiter,
+              struct adaptive_waiter *adaptive)
+ {
+     int sleep = 0;
+
+     for (;;) {
+         /*
+          * If the task was re-awoken, break out completely so we can
+          * reloop through the lock-acquisition code.
+          */
+         if (!waiter->task)
+             break;
+
+         /*
+          * We need to break if the owner changed so we can reloop
+          * and safely acquire the owner-pointer again with the
+          * wait_lock held.
+          */
+         if (adaptive->owner != rt_mutex_owner(lock))
+             break;
+
+         /*
+          * Sleep if the owner is not running....we could deadlock
+          * if they are waiting for our cpu, and it just doesn't
+          * make sense to anyway.
+          */
+         if (!adaptive->owner->oncpu) {
```

## [PATCH 4/8] adaptive real-time lock support

```
+ sleep = 1;
+ break;
+ }
+
+ cpu_relax();
+ }
+
+ put_task_struct(adaptive->owner);
+
+ return sleep;
+}
+
+static inline void
+prepare_adaptive_wait(struct rt_mutex *lock, struct adaptive_waiter *adaptive)
+{
+ /*
+ * We must acquire/lock the owner pointer while holding
+ * the wait_lock, or we risk racing against the owner
+ * exiting.
+ */
+ adaptive->owner = rt_mutex_owner(lock);
+ get_task_struct(adaptive->owner);
+}
+
+#define DECLARE_ADAPTIVE_WAITER(name) \
+ struct adaptive_waiter name = { \
+ .owner = NULL, \
+}
+
+#else
+
+#define DECLARE_ADAPTIVE_WAITER(name)
+
+#define adaptive_wait(lock, waiter, busy) (1)
+#define prepare_adaptive_wait(lock, busy) do { } while (0)
+
+#endif /* CONFIG_ADAPTIVE_RTLOCK */
+
+#endif /* __KERNEL_RTMUTEX_ADAPTIVE_H */
```

—  
To unsubscribe from this list: send the line "unsubscribe linux-kernel" in  
the body of a message to majordomo@xxxxxxxxxxxxxxxxx  
More majordomo info at <http://vger.kernel.org/majordomo-info.html>  
Please read the FAQ at <http://www.tux.org/lkml/>