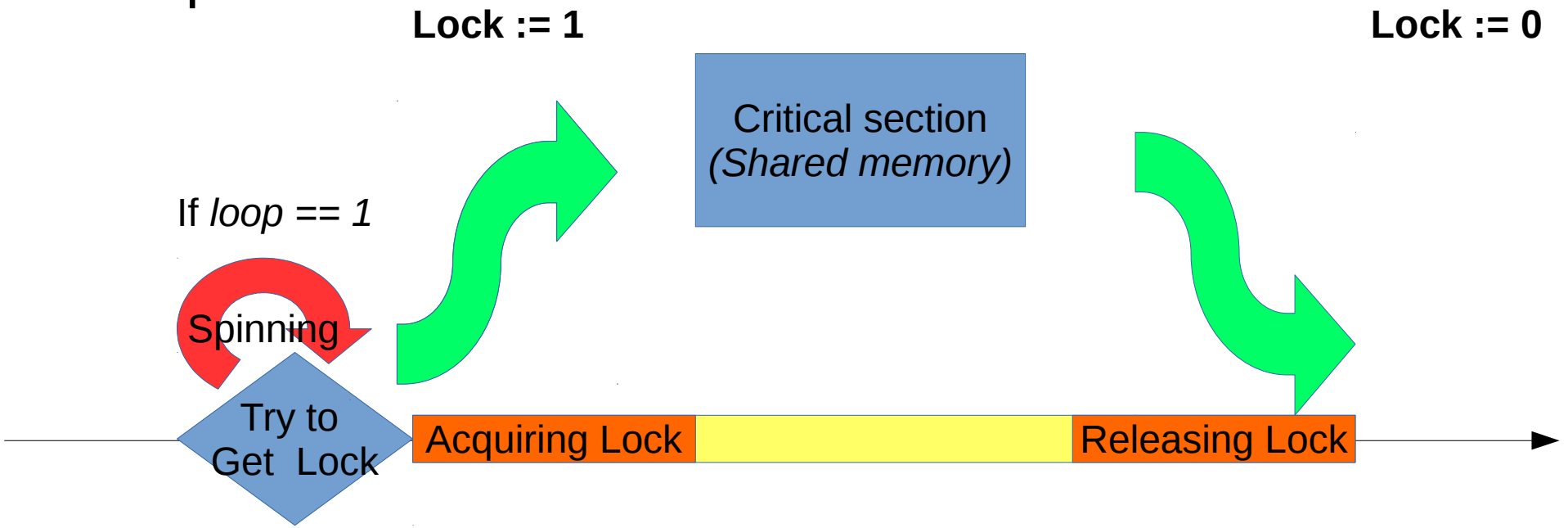
Implementing Transactional Memory in Kernel space

Breno Leitão
Linux Technology Center

leitao@debian.org
leitao@FreeBSD.org

Problem Statement

- Mutual exclusion concurrency control (Lock)
- Locks type:
  - Semaphore
  - Mutex
  - Spinlocks

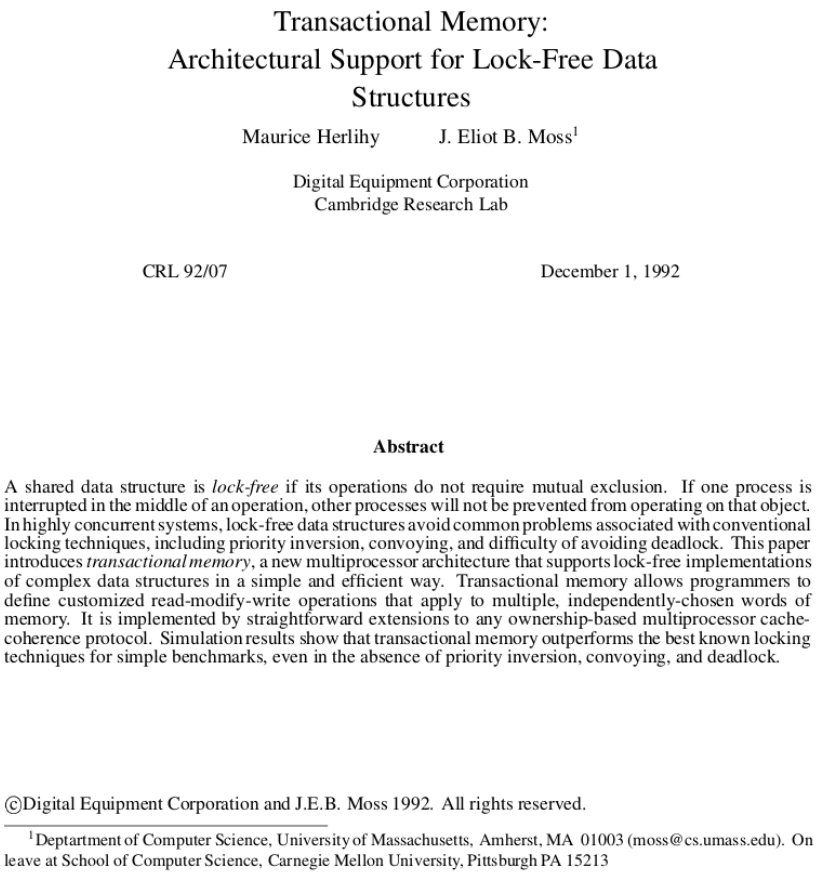


Motivation

- Optimistic critical region access
  - Commit changes as long as a conflict is not present
- No penalties if data does not conflict
- Lock Free accesses
- Atomic operation
- Lock vs HTM
- No serialization if no conflict

Concept

- Concept is not new
- Implementations:
  - Intel Transactional Synchronization Extensions (TSX)
  - POWER8 and newer
- Takes advantage of the first level cache and the cache coherency protocol



Transactional Memory:
Architectural Support for Lock-Free Data Structures

Maurice Herlihy J. Eliot B. Moss[1]

Digital Equipment Corporation
Cambridge Research Lab

CRL 92/07 December 1, 1992

Abstract

A shared data structure is *lock-free* if its operations do not require mutual exclusion. If one process is interrupted in the middle of an operation, other processes will not be prevented from operating on that object. In highly concurrent systems, lock-free data structures avoid common problems associated with conventional locking techniques, including priority inversion, convoying, and difficulty of avoiding deadlock. This paper introduces *transactional memory*, a new multiprocessor architecture that supports lock-free implementations of complex data structures in a simple and efficient way. Transactional memory allows programmers to define customized read-modify-write operations that apply to multiple, independently-chosen words of memory. It is implemented by straightforward extensions to any ownership-based multiprocessor cache-coherence protocol. Simulation results show that transactional memory outperforms the best known locking techniques for simple benchmarks, even in the absence of priority inversion, convoying, and deadlock.

[1]Deptartment of Computer Science, University of Massachusetts, Amherst, MA 01003 (moss@cs.umass.edu). On leave at School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213

http://www.hpl.hp.com/techreports/Compaq-DEC/CRL-92-7.pdf

Implementation

- HTM is a CPU feature
- Ideally we want to avoid any kernel work
- The hardware knows how to start, commit and abort a transaction.

Userspace implementation

(Intel Assembly)

```
#include "immintrin.h”

..

if (_xbegin() ==_XBEGIN_STARTED)
{
          /* transaction */

          _xend();

} else

          /* fallback*/
```

Figure 1: C example

```
        or eax, 0FFFFFFFFh

        xbegin L0

        <transaction>

        xend

        jmp  L2

L0:

        <fallback>

L2:
```

Figure 2: ASM Example

Lock Elision (Powerpc Assembly)

Lock = 0 /1

```
tle_entry:
   tbegin.            #Start TLE transaction
   beq- tle_abort     #Handle TLE transaction abort
   lwz r6,0(r3)       #Read lock
   cmpw r6,r4         #Check if lock is free
   bne- busy_lock     #If not, handle lock busy case

critical_section1:
```

Critical section instructions

```
tle_abort:
   mfspr r4, TEXASRU    # Read high-order half
                        # of TEXASR
   andis. r5,r4,0x0100  # determine whether failure
                       # is likely to be persistent
   bne tle_acquire_lock   #Persistent, acquire lock
                          #enter critical sec
   b tle_entry            #Transient, try TLE again
```

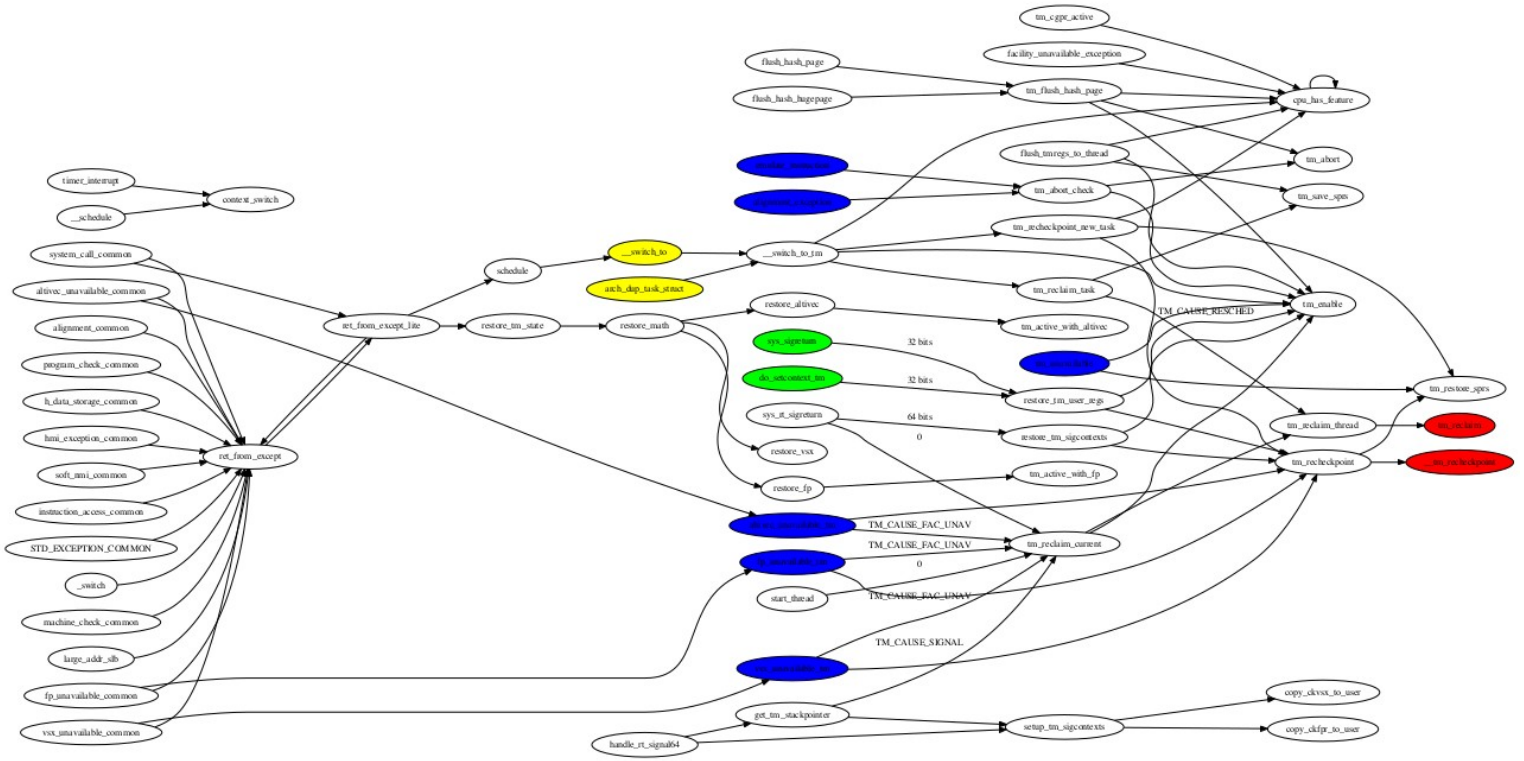
```
busy_lock:
   li r3, TLE_BUSY_LOCK
   tabort r3              #Abort TLE transaction
```

Inner implementation

- Three special purpose registers help you to debug and understand why it failed.
  - General status about failure (TEXASR)
  - Transaction Failure Instruction Address Register (TFIAR)
  - Transaction Failure Handler Address Register (TFHAR)

Kernel Implementation

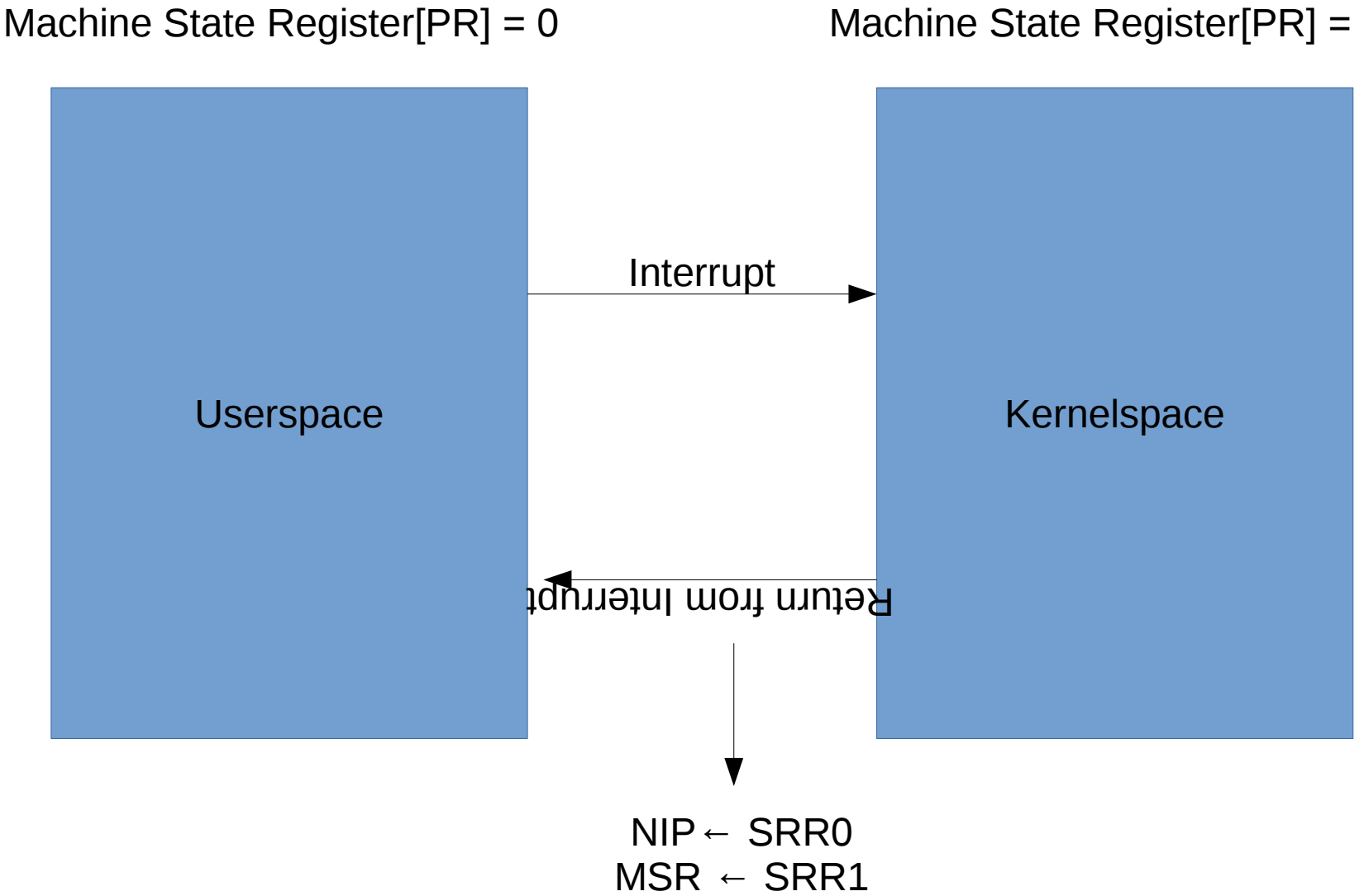
- Why do we need to about it in Kernel space?
- Kernel does not use HTM at all
  - https://github.com/leitao/htm_flow/blob/master/pdf/complete.pdf



Why kernel implementation is required?

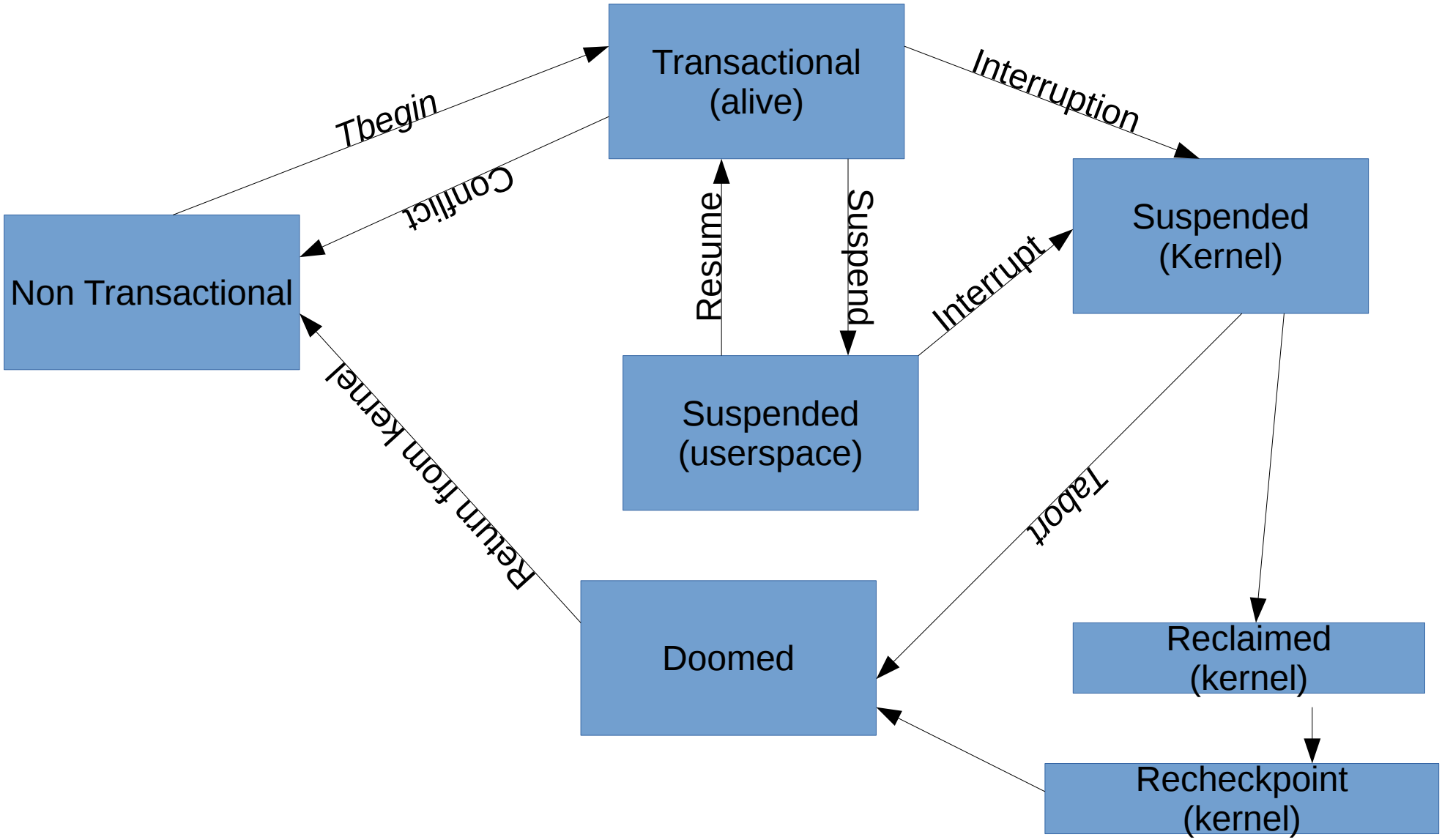
- Interruption
  - What would happen when there is an interrupt inside a transaction?
    - Page fault, Illegal instruction, Facility unavailable, etc
- Context switches
  - What to do if the scheduling quantum is over and the transaction is active?

“*Ring levels*”

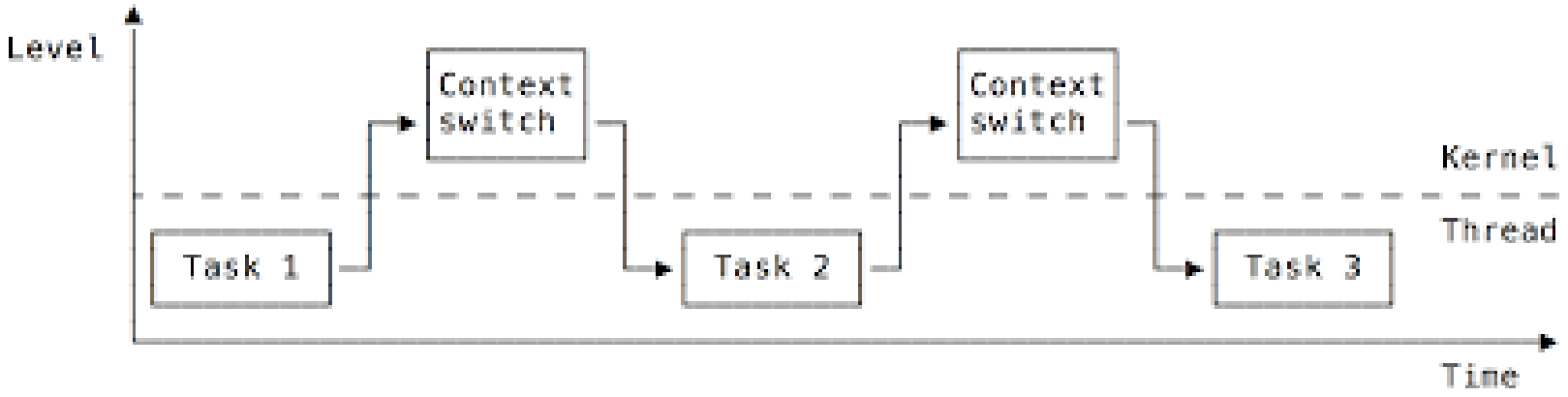


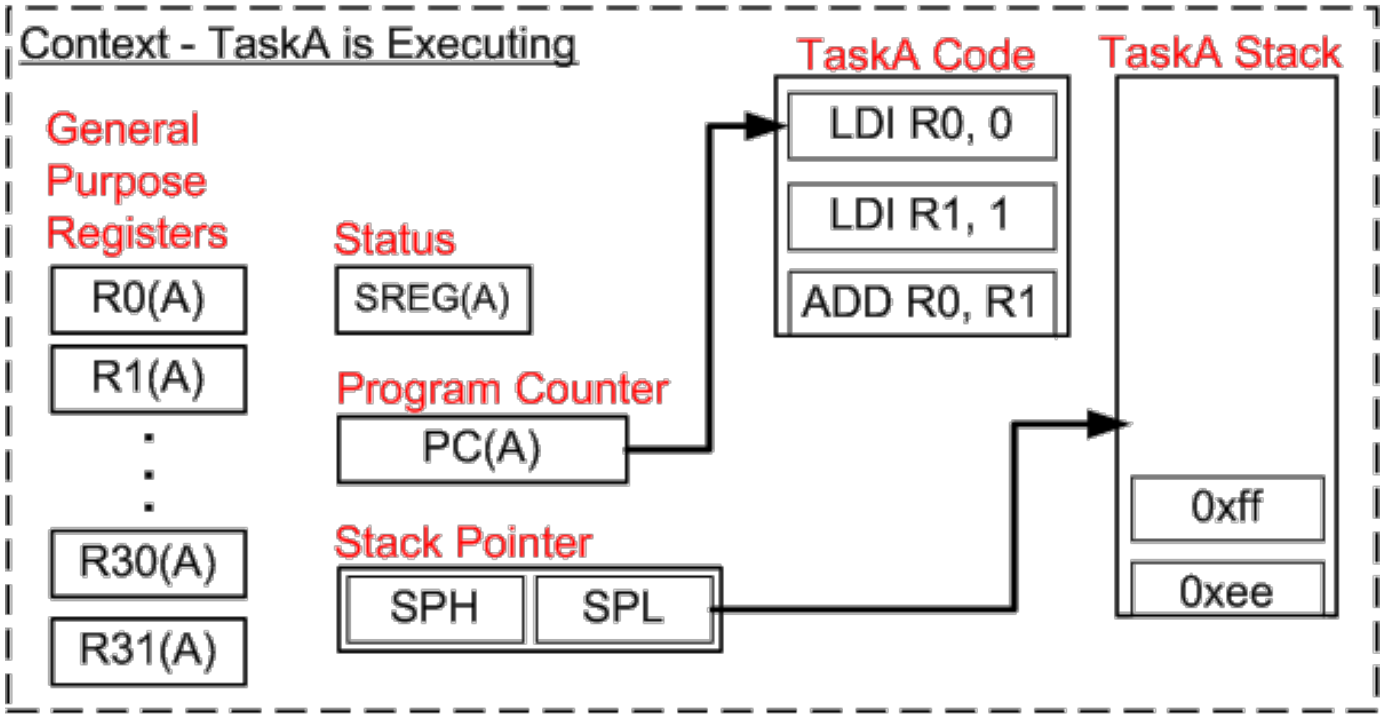
What would

Machine state (CPU)

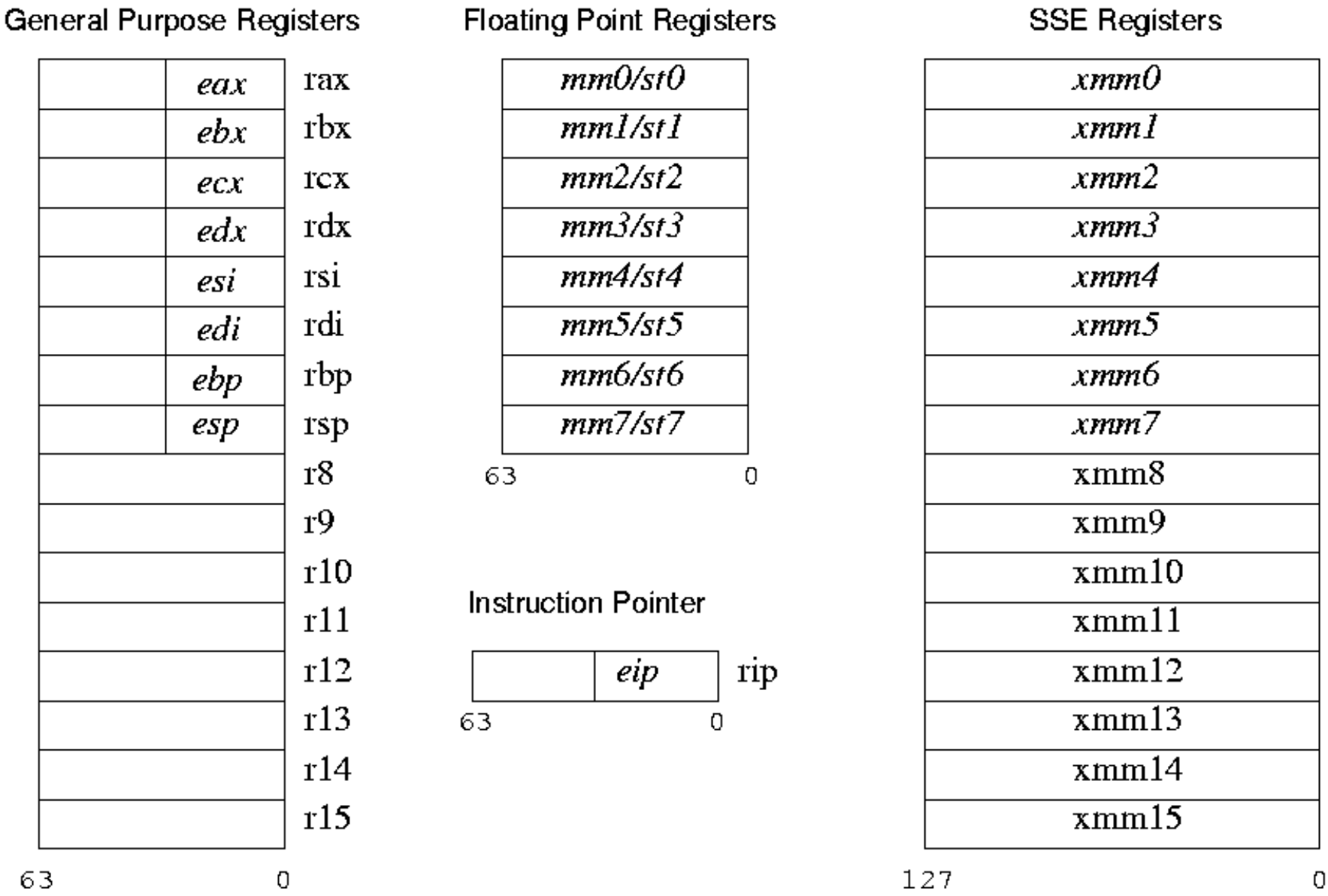


Context Switch

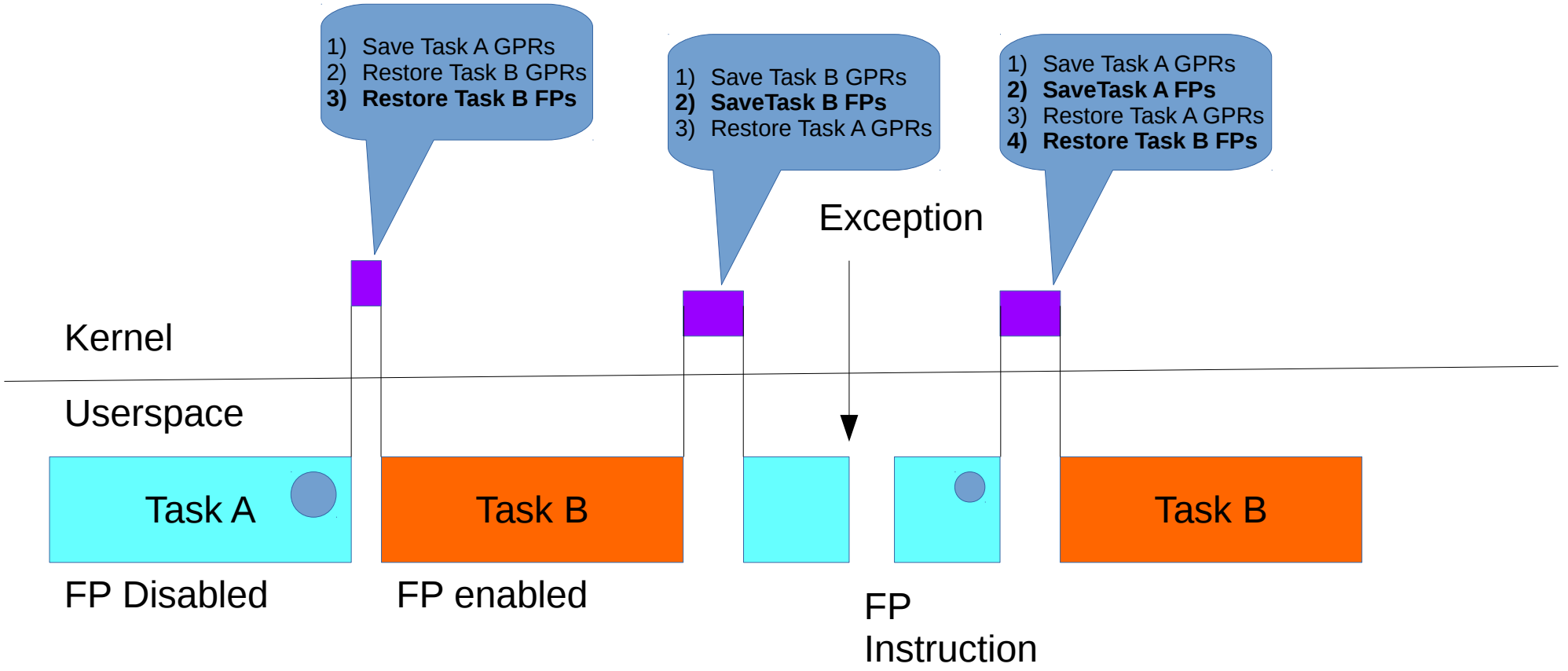




Machine register set



Lazy Save and restore



Lazy Facility enablement

- All facilities (Float point, HTM, etc) are disabled by default when a thread starts
  - First time you execute a facility instruction, as a float operation, it will cause an interrupt.
  - Goes to kernel space, zero the registers, and return to kernel space
  - Re-execute the same instruction.
- Now we need to save and restore all the FP registers.
- After a while (255 context switches), the facility is disabled again

Implementation

- Save and restore special purpose registers
- Interrupted to kernel space when a transaction is active.
  - CPU migration is impossible without cleaning the CPU transaction state.
  - Cleaning the CPU is not easy

FreeBSD implementation

- https://reviews.freebsd.org/D15243

```
@@ -303,9 +333,12 @@ trap(struct trapframe *frame)

                case EXC_FAC:
                        fscr = mfspr(SPR_FSCR);
+#ifdef HTM
                        if ((fscr & FSCR_IC_MASK) == FSCR_IC_HTM) {
-                               CTR0(KTR_TRAP, "Hardware Transactional Memory subsystem disabled");
+                               enable_htm_thread(td);
+                               break;
                        }
+#endif
```

```c
/* Enable HTM on a thread for the very first time*/
void
enable_htm_thread(struct thread *td)
{
        struct  pcb *pcb;
        struct  trapframe *tf;

        /* Transaction already active */
        if (htm_active(mfmsr())) {
                return;
        }

        pcb = td->td_pcb;
        tf = trapframe(td);

        /*
        * Save the thread's HTM CPU number, and set the CPU's current
        * vector thread
        */
        PCPU_SET(htmthread, td);

        /*
        * Enable the HTM feature unit for when the thread returns from the
        * exception. If this is the first time the unit has been used by
        * the thread, initialise the SPR registers to 0, and
        * set the flag to indicate that the vector unit is in use.
        */
        tf->srr1 |= PSL_HTM;

        if (!(pcb->pcb_flags & PCB_HTM)) {
                memset(&pcb->pcb_htm, 0, sizeof pcb->pcb_htm);
                pcb->pcb_flags |= PCB_HTM;
        }

        /*
        * Enable HTM on current MSR since we are going to access HTM SPRs
        */
        enable_htm_current_cpu();

        htm_restore_sprs(pcb);
}
```

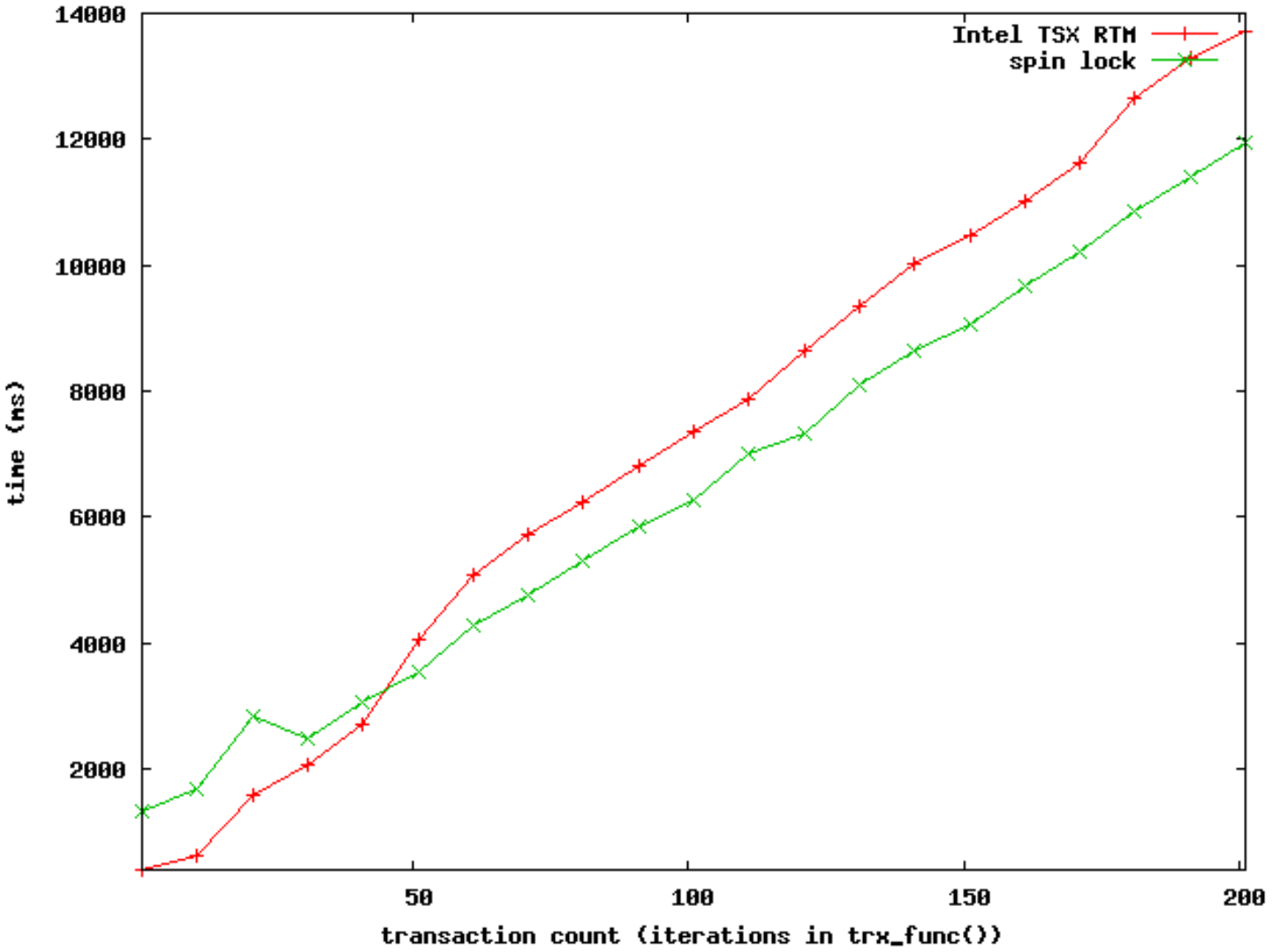
```
@@ -145,10 +146,20 @@ ENTRY(cpu_switch)
        lwz     %r7,PCB_FLAGS(%r17)
        /* Save Altivec context if needed */
        andi.   %r7, %r7, PCB_VEC
-       beq     .L2
+       beq     .L11
        bl      save_vec
        nop

+.L11:
+#if defined(HTM)
+       mr      %r3,%r14                /* restore old thread ptr */
+       lwz     %r7,PCB_FLAGS(%r17)
+       /* Save HTM SPR context if needed */
+       andi.   %r7, %r7, PCB_HTM
+       beq     .L2
+       bl      save_htm
+       nop
+#endif
 .L2:
        mr      %r3,%r14                /* restore old thread ptr */
        bl      pmap_deactivate /* Deactivate the current pmap */
@@ -206,10 +217,21 @@ blocked_loop:
        lwz     %r6, PCB_FLAGS(%r17)
        /* Restore Custom DSCR if needed */
        andi.   %r6, %r6, PCB_CDSCR
-       beq     .L4
+       beq     .L32
        ld      %r6, PCB_DSCR(%r17)     /* Load the DSCR register*/
        mtspr   SPR_DSCR, %r6

+.L32:
+#if defined(HTM)
+       lwz     %r6, PCB_FLAGS(%r17)
+       /* Restore HTM context if needed */
+       andi.   %r6, %r6, PCB_HTM
+       beq     .L4
+       mr      %r3, %r13               /* Pass curthread to restore_htm*/
+       bl      restore_htm
+       nop
+
+#endif
        /* thread to restore is in r3 */
 .L4:
        addi    %r1,%r1,48
```

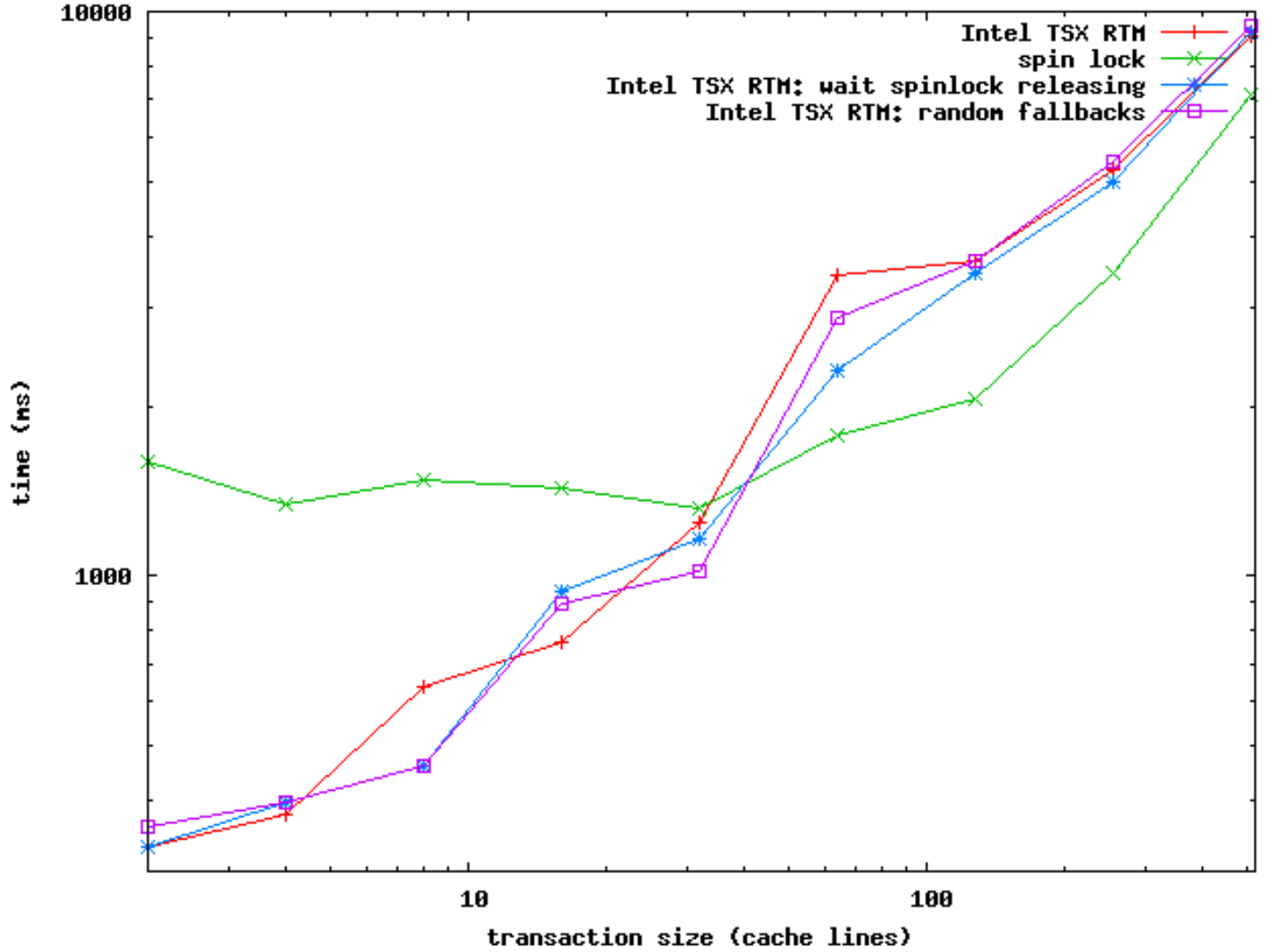
Backup

Performance Data



Source: http://natsys-lab.blogspot.com/2013/11/studying-intel-tsx-performance.html

Performance



Source: http://natsys-lab.blogspot.com/2013/11/studying-intel-tsx-performance.html

Locking problem

- Always Pessimistic
- Low data contention
  - Few collisions when accessing the critical section
  - Needs to emulate a atomic operation (Load and reserve)
- Serialization
- In the best case scenario, the critical access will be the sum of the lock and unlock calls plus the function that changes the critical area.
- Hardware Transactional Memory wants to be more optimistic when locking.
- Lock memory uncacheable.

Implementing locks

```
static inline unsigned long __arch_spin_trylock(arch_spinlock_t *lock)

{
        unsigned long tmp, token;

        token = LOCK_TOKEN;
        __asm__ __volatile__(
"1:     " PPC_LWARX(%0,0,%2,1) "\n\
        cmpwi           0,%0,0\n\
        bne-            2f\n\
        stwcx.          %1,0,%2\n\
        bne-            1b\n"
        PPC_ACQUIRE_BARRIER
"2:"
        : "=&r" (tmp)
        : "r" (token), "r" (&lock->slock)
        : "cr0", "memory");

        return tmp;
}
```