Report: studio sincronizzazione variabili condivise e istruzioni atomiche su architettura RISC-V

Esame di Embedded systems - 056899 A.A. 2023/2024

Francesco Maria Tranquillo
Gioele Peltrera

Introduzione

In questo report analizzeremo i metodi software per la sincronizzazione multi-thread e la loro implementazione hardware, tra cui mutex, semafori, spinlock e compare exchange. Inoltre affronteremo anche il tema delle istruzioni atomiche e illustreremo una soluzione architetturale per una cpu RISC-V che possa supportare tali soluzioni.
I test eseguiti sono frutto di simulazioni attraverso la piattaforma gem5, utilizzando una cpu timing o O3 a due core, con un protocollo di coerenza MESI a due livelli implementato attraverso Ruby, e sistema operativo Linux.

```
import m5
from m5.objects import Root

from gem5.components.boards.riscv_board import RiscvBoard
from gem5.components.cachehierarchies.ruby.mesi_two_level_cache_hierarchy import (
    MESITwoLevelCacheHierarchy,
)
from gem5.coherence_protocol import CoherenceProtocol
from gem5.components.memory import SingleChannelDDR3_1600
from gem5.components.processors.cpu_types import CPUTypes
from gem5.components.processors.simple_switchable_processor import (
    SimpleSwitchableProcessor,
)
from gem5.isas import ISA
from gem5.resources.resource import DiskImageResource, obtain_resource, CheckpointResource
from gem5.simulate.simulator import Simulator
from gem5.utils.requires import requires

# Run a check to ensure the right version of gem5 is being used.
requires(
    isa_required=ISA.RISCV,
    coherence_protocol_required=CoherenceProtocol.MESI_TWO_LEVEL,
)

# Setup the cache hierarchy.
# For classic, PrivateL1PrivateL2 and NoCache have been tested.
# For Ruby, MESI_Two_Level and MI_example have been tested.
cache_hierarchy = MESITwoLevelCacheHierarchy(
    l1d_size="32kB",
    l1d_assoc=8,
    l1i_size="32kB",
    l1i_assoc=8,
    l2_size="256kB",
    l2_assoc=16,
    num_l2_banks=2,
)
# Setup the system memory.
memory = SingleChannelDDR3_1600()

# Setup a single core Processor.
processor = SimpleSwitchableProcessor(
    starting_core_type = CPUTypes.ATOMIC,
    switch_core_type=CPUTypes.O3,#CPUTypes.TIMING,
    isa=ISA.RISCV,
    num_cores=2
)

# Setup the board.
board = RiscvBoard(
    clk_freq="1GHz",
    processor=processor,
    memory=memory,
    cache_hierarchy=cache_hierarchy,
)

# Set the Full System workload.
board.set_kernel_disk_workload(

    kernel=obtain_resource("riscv-bootloader-vmlinux-5.10","/home/user/Documents/POLIMI/HD/binaries/"
),
    disk_image=DiskImageResource("/home/user/Documents/POLIMI/HD/ubuntu_ARM/riscv-disk-img"),
    readfile="/home/user/Documents/POLIMI/HD/sources/acquire_release",
    checkpoint = CheckpointResource("/home/user/Documents/POLIMI/HD/gem5_installation/gem5/m5out/cpt.
281422923000")
)
```

```
104
105     simulator = Simulator(board=board)
106     print("Beginning simulation!")
107     # Note: This simulation will never stop. You can access the terminal upon boot
108     # using m5term (`./util/term`): `./m5term localhost <port>`. Note the `<port>`
109     # value is obtained from the gem5 terminal stdout. Look out for
110     # "system.platform.terminal: Listening for connections on port <port>".
111     simulator.run()
112
```

Per ottenere dei risultati più significativi, all'inizio del codice di ogni thread verrà messo un ciclo che continuerà a ripetersi finché entrambi i thread non si troveranno in quello stesso ciclo, così che il primo thread creato non inizi subito l'esecuzione della zona critica prima che il secondo thread venga creato dal main.

```
pthread_spinlock_t sp_lock;

void *func(void *par)
{
        int increm = *((int *)par);
        unsigned int old_val = 0;
        int local_start = 0;

        while(!local_start)
        {
                pthread_spin_lock(&sp_lock);
                if(start!=0 && start!=increm)
                        local_start=1;
                start = increm;
                pthread_spin_unlock(&sp_lock);
        }
        ...
        ...

 }
```

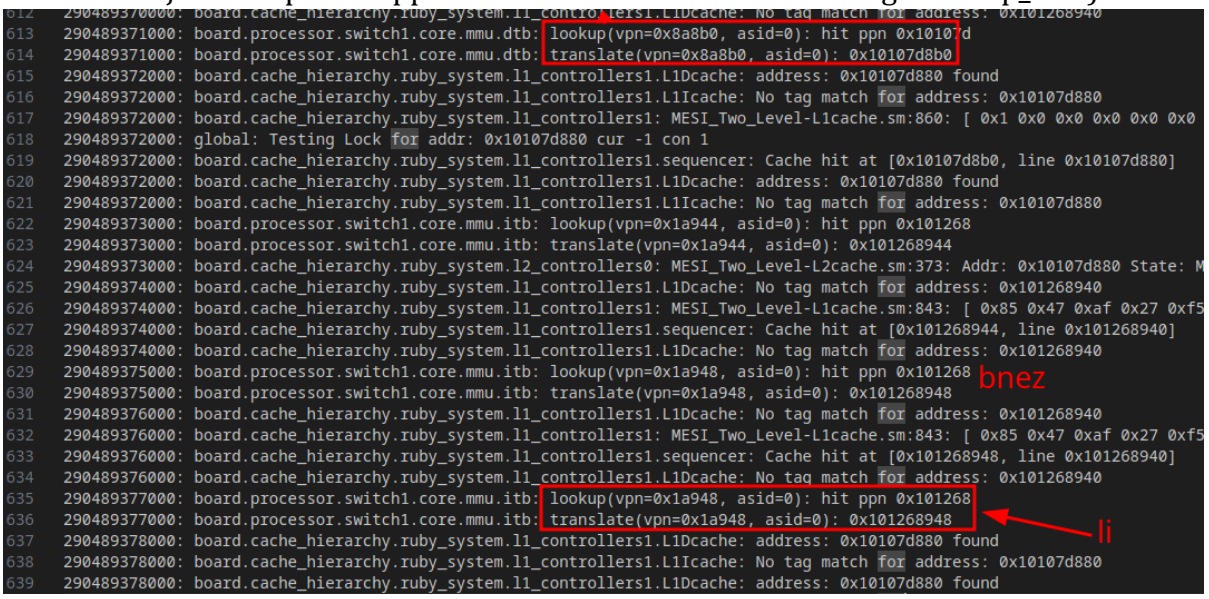
SpinLock

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
//#include <time.h>
#include <gem5/m5ops.h>
#define N_TIMES 10
#define N_INCREM 3

pthread_spinlock_t sp_lock;
unsigned int shared_value = 0;
int start = 0;
int change = 0;

void *func(void *par)
{
        int increm = *((int *)par);
        unsigned int old_val = 0;
        int local_start = 0;

        while(!local_start)
        {
                pthread_spin_lock(&sp_lock);
                if(start!=0 && start!=increm)
                        local_start=1;
                start = increm;
                pthread_spin_unlock(&sp_lock);
        }

        for(int i=0; i<N_TIMES; i++)
        {
                pthread_spin_lock(&sp_lock);
                if(shared_value!=old_val)
                        change++;
                for(int j=0; j<N_INCREM; j++)
                {
                        shared_value = shared_value + increm;
                }
                old_val=shared_value;
                pthread_spin_unlock(&sp_lock);
        }
        return NULL;
}

int main()
{
        pthread_t thread_1, thread_2;
        int param_1= 1;
        int param_2= 2;

        if(pthread_spin_init(&sp_lock, PTHREAD_PROCESS_PRIVATE) != 0)
        {
                printf("ERROR in initializing spin lock!\n");
                return -1;
        }
        m5_switch_cpu();
        pthread_create(&thread_1, NULL, func, (void *)&param_1);
        pthread_create(&thread_2, NULL, func, (void *)&param_2);

        pthread_join(thread_1, NULL);
        pthread_join(thread_2, NULL);
        if(pthread_spin_destroy(&sp_lock) != 0)
        {
                printf("ERROR in initializing spin lock!\n");
                return -1;
        }
        printf("Number of changes in thread lock owning: %d\n", change);
        fflush(stdout);
//      m5_exit(0);
        return 0;
}
```

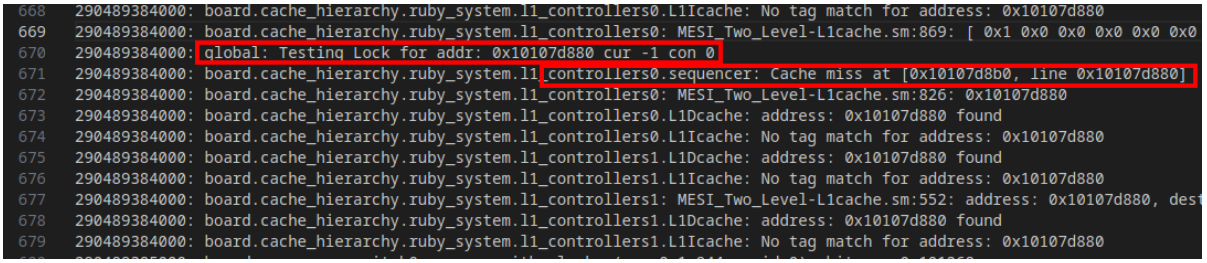
Lo spin lock esegue un'operazione di atomic-swap (marchiata come acquire, ergo tutte le operazioni successive alla swap non potranno essere riordinate ed eseguite prima di questa operazione atomica).
Successivamente se l'atomic swap dovesse aver fallito, quindi nel caso in cui il valore contenuto all'indirizzo di memoria della variabile globale di lock fosse già 1, grazie alla "bnez" invece che concludere la funzione con una ret, passiamo ad un loop, che precede la logica di load reserve-store conditional.

```
000000000001a940 <__pthread_spin_lock>:
   1a940:   4785                li a5,1
   1a942:   0cf527af            amoswap.w.aq   a5,a5,(a0)
   1a946:   2781                sext.w   a5,a5
   1a948:   e399                bnez  a5,1a94e <__pthread_spin_lock+0xe>
   1a94a:   4501                li a0,0  PRESO LOCK
   1a94c:   8082                ret
   1a94e:   4705                li a4,1  NON PRESO LOCK
   1a950:   411c                lw a5,0(a0)
   1a952:   fffd                bnez  a5,1a950 <__pthread_spin_lock+0x10>
   1a954:   100527af            lr.w  a5,(a0)
   1a958:   e781                bnez  a5,1a960 <__pthread_spin_lock+0x20>
   1a95a:   1ce526af            sc.w.aq  a3,a4,(a0)
   1a95e:   fafd                bnez  a3,1a954 <__pthread_spin_lock+0x14>
   1a960:   2781                sext.w   a5,a5
   1a962:   f7fd                bnez  a5,1a950 <__pthread_spin_lock+0x10>
   1a964:   4501                li a0,0   PRESO LOCK
   1a966:   8082                ret

000000000001a968 <__pthread_spin_unlock>:
   1a968:   0f50000f            fence iorw,ow
   1a96c:   0805202f            amoswap.w   zero,zero,(a0)
   1a970:   4501                li a0,0
   1a972:   8082                ret
```

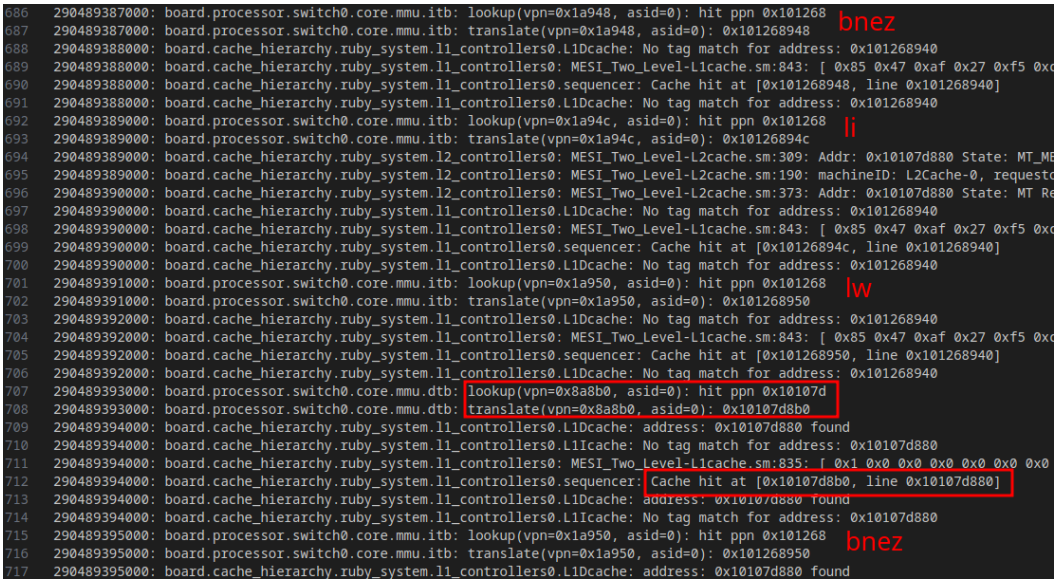
Il core switch1, a riga 618, è il primo a concludere correttamente l'atomic swap, quindi tutte le successive atomic swap non ritorneranno più 0, ma 1 (l'indirizzo 0x8a8b0 si vede dal file di object dump che rappresenta l'indirizzo della variabile globale sp_lock).



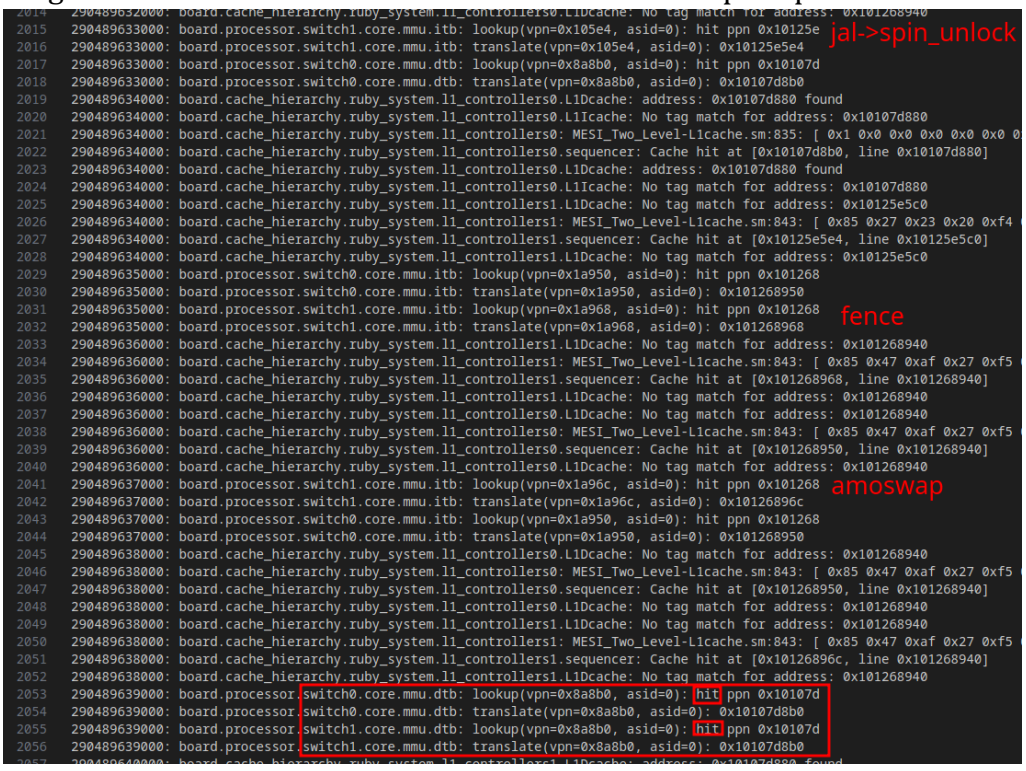
Il core switch0, a riga 670, ritorna dall'atomic swap con un risultato diverso da 0 dopo aver fatto una miss sulla variabile sp_lock, poichè la sua entry (presente nella cache L1 del core) è stata invalidata dalla prima atomic swap del core 1.



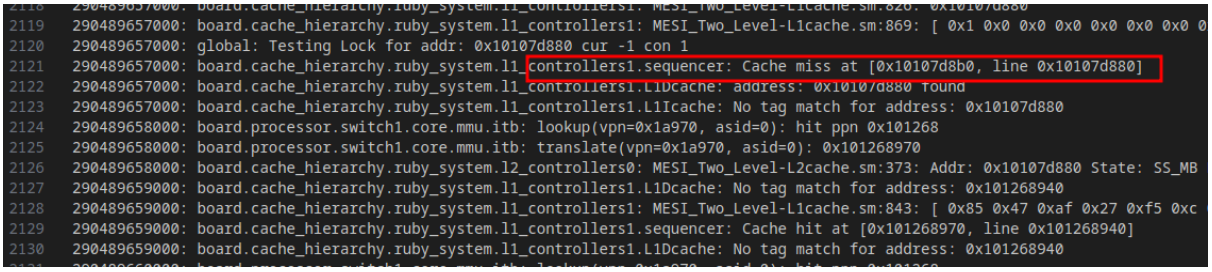
A riga 701 si vede che il core switch0 che ha eseguito la jump iniziare a ciclare sulla variabile di spin lock prima della load reserve, tanto è vero che a riga 707 switch0 richiede l'indirizzo di memoria per la variabile sp_lock, questa volta risultando in una hit.



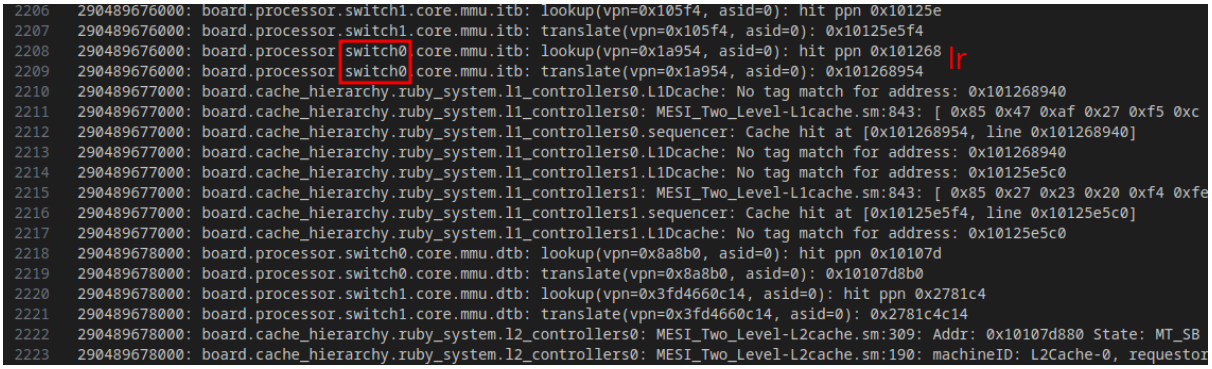
A riga 2015 inizia la fase di unlock del core switch1, che recupera la variabile sp_lock facendo anche lui hit.
Entrambi i core riescono a fare hit poiché la variabile sp_lock è usata da entrambi in sola lettura.
A riga 2055 lo switch1 libera con un'altra atomic-swap lo spin lock.



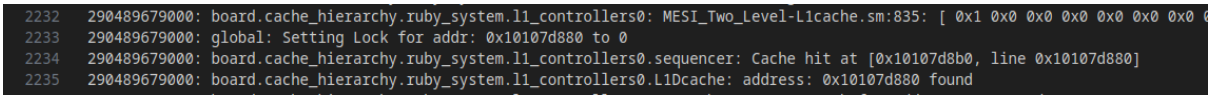
A riga 2121, quando si conclude l'unlock, il core 1 fa miss sull'indirizzo della variabile sp_lock, poiché la modifica fatta dall'atomic-swap invalida tutti i blocchi della cache.



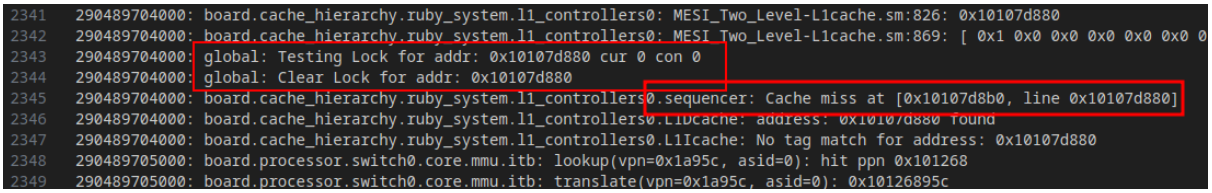
A riga 2208 il core 0 esegue la load reserve



e a riga 2233 grazie al messaggio "global: Setting Lock":



la load reserve scrive sul lock globale il valore univoco corrispondente al suo contesto e, a riga 2343-44, con i messaggi:



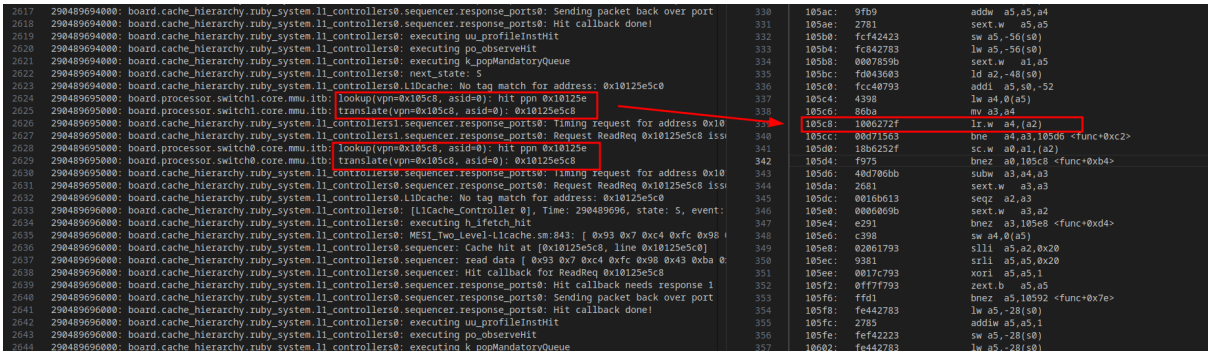
la store conditional si conclude correttamente, seguita da una miss, sempre a causa dell'invalidazione del blocco di memoria data da una modifica.

Compare exchange

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdatomic.h>
#include <gem5/m5ops.h>
#define N 50

pthread_spinlock_t sp_lock;
int shared_value = 0;
int start = 0;
int change = 0;

void *func(void *par)
{
        int increm = *((int *)par);
        unsigned int old_val = 0;
        int local_start = 0;
        int curr = 0;
        int expected = 0;
        int n_retry = 0;

        while(!local_start)
        {
                pthread_spin_lock(&sp_lock);
                if(start!=0 && start!=increm)
                        local_start=1;
                start = increm;
                pthread_spin_unlock(&sp_lock);
        }

        for(int i=0; i<N; i++)
        {
                curr = expected;
                while(!atomic_compare_exchange_weak_explicit(&shared_value, &expected,
                                curr+increm, memory_order_relaxed, memory_order_relaxed))
                        curr = expected;
        }
        return NULL;
}

int main()
{
        pthread_t thread_1, thread_2;
        int param_1= 1;
        int param_2= -1;

        if(pthread_spin_init(&sp_lock, PTHREAD_PROCESS_PRIVATE) != 0)
        {
                printf("ERROR in initializing spin lock!\n");
                return -1;
        }
        m5_switch_cpu();
        pthread_create(&thread_1, NULL, func, (void *)&param_1);
        pthread_create(&thread_2, NULL, func, (void *)&param_2);

        pthread_join(thread_1, NULL);
        pthread_join(thread_2, NULL);
        if(pthread_spin_destroy(&sp_lock) != 0)
        {
                printf("ERROR in initializing spin lock!\n");
                return -1;
        }
        printf("Final value: %d\n", shared_value);
        fflush(stdout);
//      m5_exit(0);
        return 0;
}
```

Per quanto riguarda l'atomic_compare_exchange(), quello che si cerca di fare è modificare il valore contenuto nella variabile condivisa dai thread, gli si passa il valore atteso che dev'essere conenuto al momento della chiamata a funzione e: in caso il valore fosse corretto si modifica la variabile condivisa con il valore desiderato, mentre in caso contrario si modifica semplicemente il valore atteso con il valore contenuto nella variabile condivisa in questo momento.
La funzione atomic_compare_exchange(), viene tradotta dal compilatore direttamente come una load reserve, store conditional.

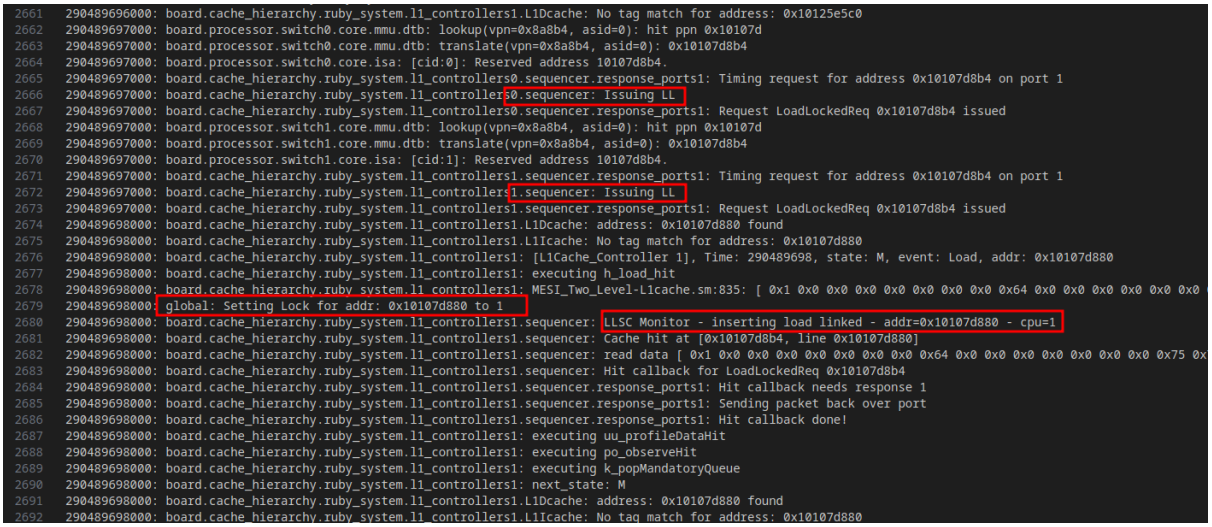
A riga 2624: entrambi i thread arrivano alla prima load reserve ed entrambi hanno come next state S (poiché quella pagina contiene un'istruzione, quindi non da modificare)



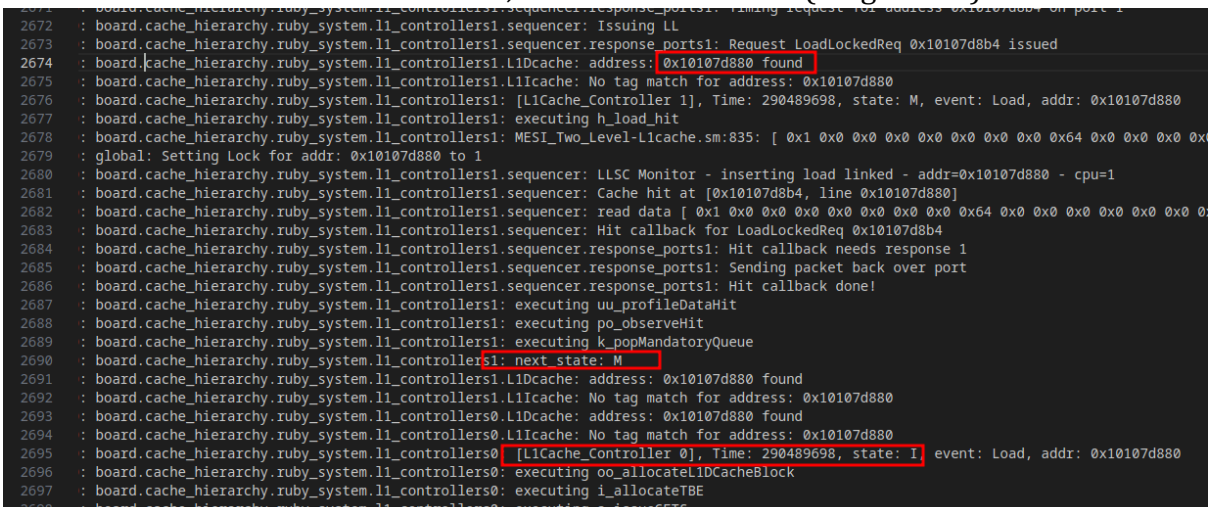
A riga 2672: il core 1 è il primo ad acquisire il lock con:

- Issuing LL
- LLSC Monitor - inserting load linked - addr=0x10107d880 - cpu=1
- Cache hit at [0x10107d8b4, line 0x10107d880]
- next_state: M

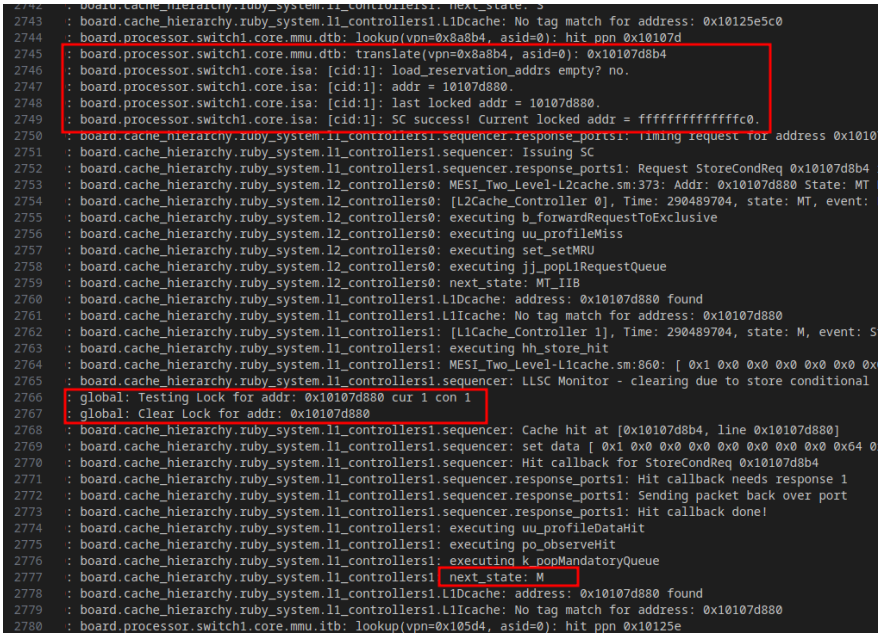
qui next state è M, nonostante basti lo stato S, poiché già si trovava in M prima della load (e non è ancora stata eseguita la write back).



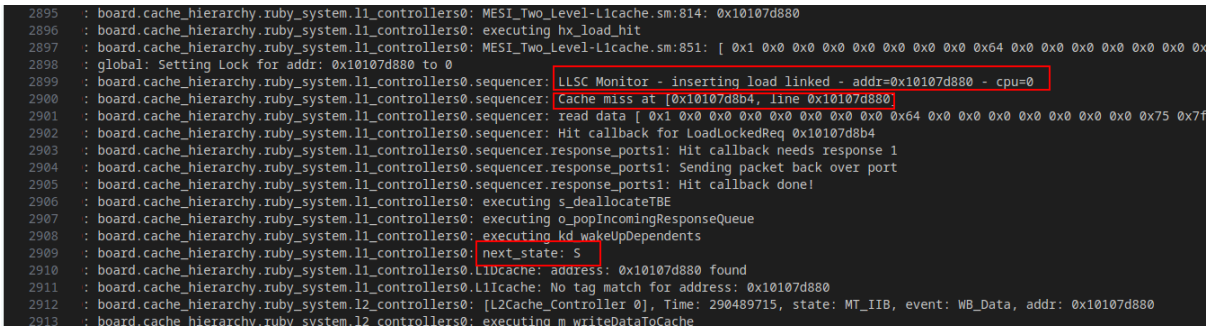
Il core 1 ottiene il lock a seguito della found (a riga 2674) per la pagina della shared variable.
Successivamente a riga 2676 c'è la risposta (come detto prima in stato M), mentre a riga 2693 anche il core 0 fa address found, ma si trova in stato I (a riga 2695).



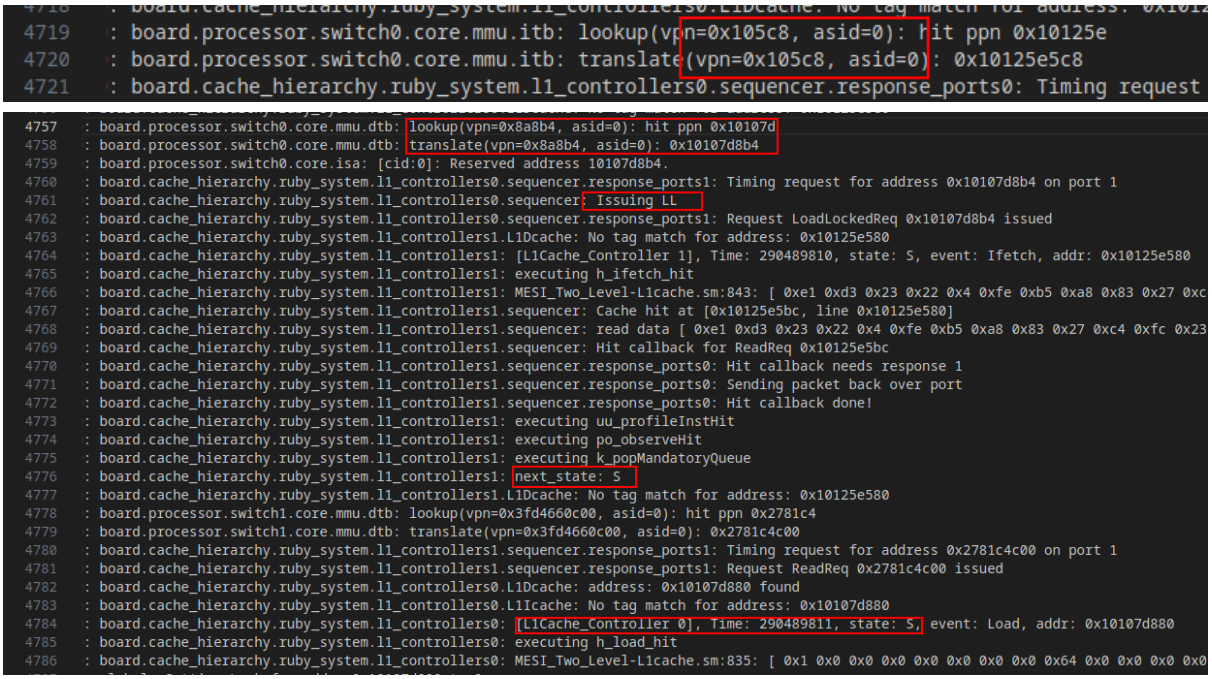
Da riga 2744 a riga 2749: il core 1 finisce con la store conditional e verifica che la reservation fosse ancora valida (riga 2766, 2767), concludendo con lo stato M.



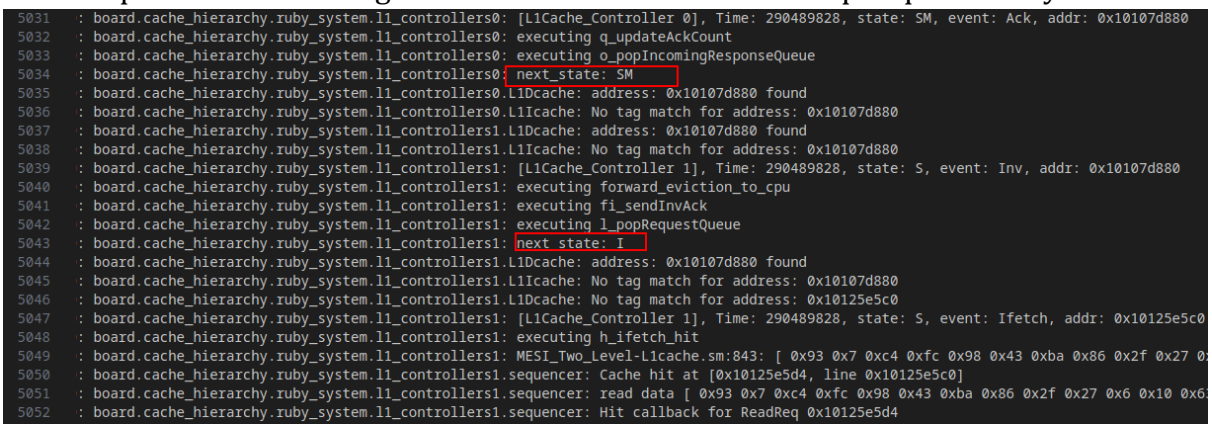
A riga 2899: il core 0 fa miss sulla entry di cache della shared variable, quindi recupera la entry che adesso può tornare in stato S. Infatti a riga 2909 viene fatta fare la write back dal core 1.



A riga 4719 il core 0 raggiunge la load reserve, fa lookup sulla shared variable (4757) e riesce a concludere la load reserve (4784).
Da qui in poi quando prendo la load reserve sulla shared variable lo stato sarà S e passerà in stato M solo dopo la store conditional (riga 5122).



A riga 5034: il core 0 sta per eseguire la store conditional, quindi la entry di shared variable della sua cache passa dallo stato S a SM, invalidando così le entry di tutte le altre cache.
Infatti si può notare che a riga 5043 il core 1 si trova in stato I per quella entry.



N.B: quando passa da stato S a M, invalida anche la entry nella propria cache, tanto è vero che la prima volta, in cui ci si trovava già in stato M, dopo i messaggi

LLSC Monitor - clearing due to store conditional - addr=0x10107d880 - cpu=0
global: Testing Lock for addr: 0x10107d880 cur 0 con 0
global: Clear Lock for addr: 0x10107d880

veniva comunque fatta una hit. Mentre da quel punto in poi, dopo la transizione da S a M, abbiamo sempre una miss.

Semaphores e Mutex

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <semaphore.h>
#include <gem5/m5ops.h>
#define N_TIMES 100
#define N_INCREM 3

sem_t my_sem;
pthread_spinlock_t sp_lock;
unsigned int shared_value = 0;
int start = 0;
int change = 0;

void *func(void *par)
{
        int increm = *((int *)par);
        unsigned int old_val = 0;
        int local_start = 0;

        while(!local_start)
        {
                pthread_spin_lock(&sp_lock);
                if(start!=0 && start!=increm)
                        local_start=1;
                start = increm;
                pthread_spin_unlock(&sp_lock);
        }

        for(int i=0; i<N_TIMES; i++)
        {
                sem_wait(&my_sem);
                if(shared_value!=old_val)
                        change++;
                for(int j=0; j<N_INCREM; j++)
                {
                        shared_value = shared_value + increm;
                }
                old_val=shared_value;
//              shared_value += increm;
                sem_post(&my_sem);
        }
        return NULL;
}

int main()
{
        pthread_t thread_1, thread_2;
        int param_1= 1;
        int param_2= -1;

        printf("Testing semaphore...\n");
        if(sem_init(&my_sem, 0, 1)!=0 || pthread_spin_init(&sp_lock, PTHREAD_PROCESS_PRIVATE)!=0)
        {
                printf("ERROR in initializing!\n");
                return -1;
        }
        m5_switch_cpu();
        pthread_create(&thread_1, NULL, func, (void *)&param_1);
        pthread_create(&thread_2, NULL, func, (void *)&param_2);

        pthread_join(thread_1, NULL);
        pthread_join(thread_2, NULL);
        if(sem_destroy(&my_sem)!=0 || pthread_spin_destroy(&sp_lock)!=0)
        {
                printf("ERROR in cleaning!\n");
                return -1;
        }
        printf("Number of changes in thread lock owning: %d\n", change);
        printf("Final value: %d\n", shared_value);
        fflush(stdout);
//      m5_exit(0);
        return 0;
}
```

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <gem5/m5ops.h>
#define N_TIMES 10
#define N_INCREM 3

pthread_mutex_t my_mutex;
pthread_spinlock_t sp_lock;
unsigned int shared_value = 0;
int start = 0;
int change = 0;

void *func(void *par)
{
        int increm = *((int *)par);
        unsigned int old_val = 0;
        int local_start = 0;

        while(!local_start)
        {
                pthread_spin_lock(&sp_lock);
                if(start!=0 && start!=increm)
                        local_start=1;
                start = increm;
                pthread_spin_unlock(&sp_lock);
        }

        //printf("Entro %d\n", increm);

        for(int i=0; i<N_TIMES; i++)
        {
                if(pthread_mutex_lock(&my_mutex)!=0)
                        return (void *) -1;
                if(shared_value!=old_val)
                        change++;
                for(int j=0; j<N_INCREM; j++)
                {
                        shared_value = shared_value + increm;
                }
                old_val=shared_value;
//              shared_value += increm;
                if(pthread_mutex_unlock(&my_mutex)!=0)
                        return (void *) -1;
        }

//      printf("Esco %d\n", increm);
        return 0;
}

int main()
{
        pthread_t thread_1, thread_2;
        int param_1= 1;
        int param_2= -1;
        void *ret;

        printf("Testing mutex...\n");
        if(pthread_mutex_init(&my_mutex, NULL)!=0 || pthread_spin_init(&sp_lock, PTHREAD_PROCESS_PRIVATE)!=0)
        {
                printf("ERROR in initializing!\n");
                return -1;
        }
        m5_switch_cpu();
        pthread_create(&thread_1, NULL, func, (void *)&param_1);
        pthread_create(&thread_2, NULL, func, (void *)&param_2);

        pthread_join(thread_1, &ret);
        if((int)ret!=0)
                printf("IL THREAD 1 HA FALLITO!\n");
        pthread_join(thread_2, &ret);
        if((int)ret!=0)
                printf("IL THREAD 2 HA FALLITO!\n");
        if(pthread_mutex_destroy(&my_mutex)!=0 || pthread_spin_destroy(&sp_lock)!=0)
        {
                printf("ERROR in cleaning!\n");
                return -1;
        }
        printf("Number of changes in thread lock owning: %d\n", change);
        printf("Final value: %d\n", shared_value);
        fflush(stdout);
//      m5_exit(0);
        return 0;
}
```

Dopo aver realizzato un implementazione per la sincronizzazione fra thread con i semafori e i mutex, ci siamo resi conto che venivano tradotti dal compilatore di nuovo come delle load reserve store conditional. Quindi il loro funzionamento era analogo alle implementazioni, più semplici e facilmente analizzabili dal simulatore, svolte precedentemente.

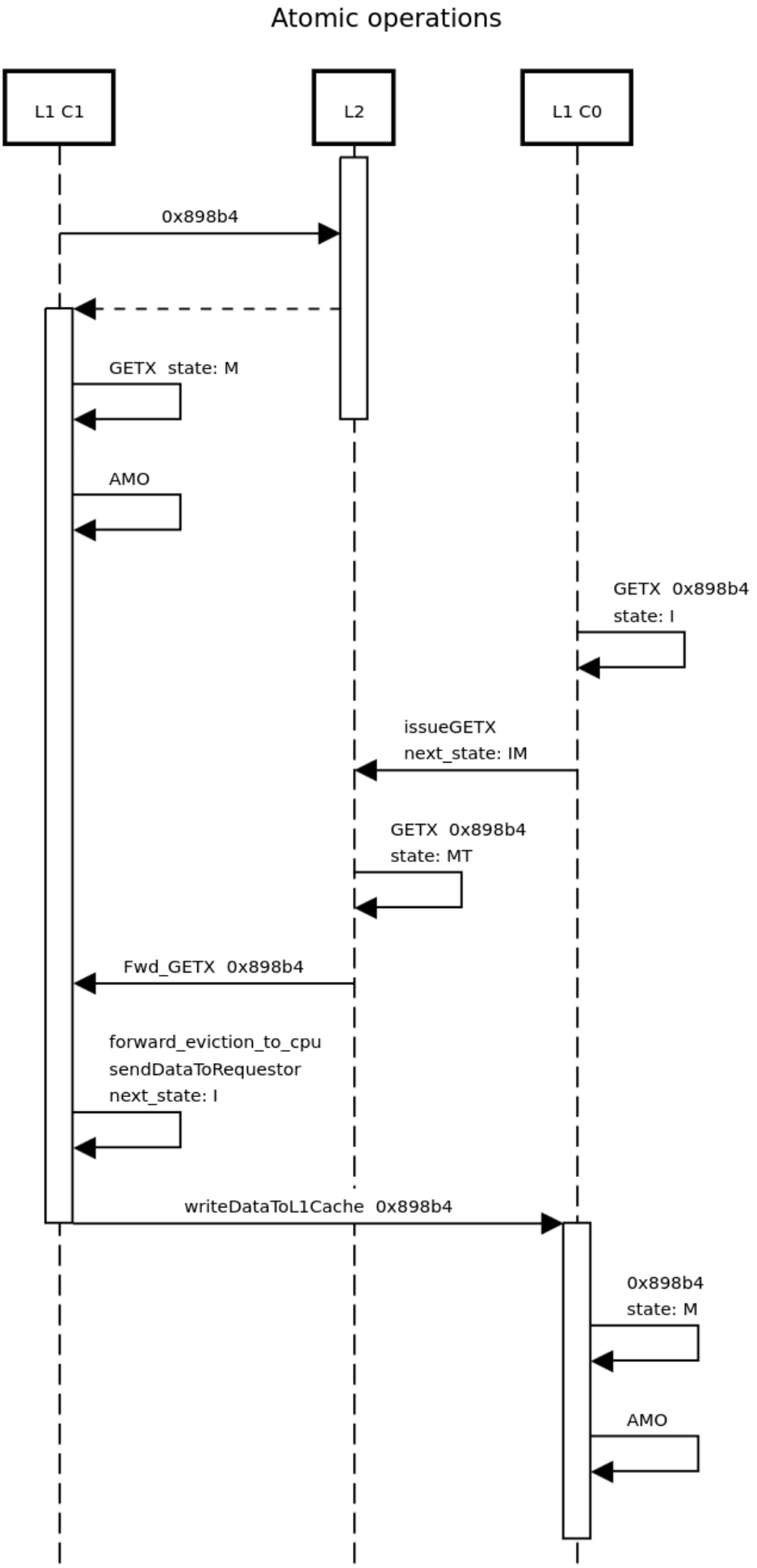
Ad esempio per il semaforo:

```
000000000001ab16 <__new_sem_wait>:
   1ab16:       1141                    addi    sp,sp,-16
   1ab18:       e022                    sd      s0,0(sp)
   1ab1a:       e406                    sd      ra,8(sp)
   1ab1c:       842a                    mv      s0,a0
   1ab1e:       2e0200ef                jal     3adfe <___pthread_testcancel>
   1ab22:       601c                    ld      a5,0(s0)
   1ab24:       02079713                slli    a4,a5,0x20
   1ab28:       9301                    srli    a4,a4,0x20
   1ab2a:       c30d                    beqz    a4,1ab4c <__new_sem_wait+0x36>
   1ab2c:       fff78693                addi    a3,a5,-1
   1ab30:       1004372f                lr.d    a4,(s0)
   1ab34:       00f71563                bne     a4,a5,1ab3e <__new_sem_wait+0x28>
   1ab38:       1cd4362f                sc.d.aq a2,a3,(s0)
   1ab3c:       fa75                    bnez    a2,1ab30 <__new_sem_wait+0x1a>
   1ab3e:       00f71763                bne     a4,a5,1ab4c <__new_sem_wait+0x36>
   1ab42:       60a2                    ld      ra,8(sp)
   1ab44:       6402                    ld      s0,0(sp)
   1ab46:       4501                    li      a0,0
   1ab48:       0141                    addi    sp,sp,16
   1ab4a:       8082                    ret
   1ab4c:       8522                    mv      a0,s0
   1ab4e:       6402                    ld      s0,0(sp)
   1ab50:       60a2                    ld      ra,8(sp)
   1ab52:       0141                    addi    sp,sp,16
   1ab54:       bf39                    j       1aa72 <__new_sem_wait_slow64.constprop.0>
```

Atomic operations

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdatomic.h>
#include <gem5/m5ops.h>
#include <pthread.h>
#define N_TIMES 10

pthread_spinlock_t sp_lock;
atomic_int my_index = ATOMIC_VAR_INIT(0);
//int my_index=0;
int start = 0;

void *func(void *par)
{
        int increm = *((int *)par);
        int n_times = *(((int *)par)+1);
        int local_start = 0;

        while(!local_start)
        {
                pthread_spin_lock(&sp_lock);
                if(start!=0 && start!=increm)
                        local_start=1;
                start = increm;
                pthread_spin_unlock(&sp_lock);
        }

        for(int i=0; i<n_times; i++)
        {
                atomic_fetch_add(&my_index, increm);
                atomic_fetch_add(&my_index, increm);
                atomic_fetch_add(&my_index, increm);
                atomic_fetch_add(&my_index, increm);
                atomic_fetch_add(&my_index, increm);
                atomic_fetch_add(&my_index, increm);
                atomic_fetch_add(&my_index, increm);
                atomic_fetch_add(&my_index, increm);
                atomic_fetch_add(&my_index, increm);
                //my_index = my_index+increm;
        }
        return NULL;
}

int main()
{
        pthread_t thread_1, thread_2;
        int param_1[]={1, N_TIMES};
        int param_2[]={-1, N_TIMES};

        if(pthread_spin_init(&sp_lock, PTHREAD_PROCESS_PRIVATE) != 0)
        {
                printf("ERROR in initializing spin lock!\n");
                return -1;
        }
        m5_switch_cpu();
        pthread_create(&thread_1, NULL, func, (void *)&param_1);
        pthread_create(&thread_2, NULL, func, (void *)&param_2);
        pthread_join(thread_1, NULL);
        pthread_join(thread_2, NULL);
        if(pthread_spin_destroy(&sp_lock) != 0)
        {
                printf("ERROR in initializing spin lock!\n");
                return -1;
        }
        printf("%d\n", my_index);
        return 0;
}
```

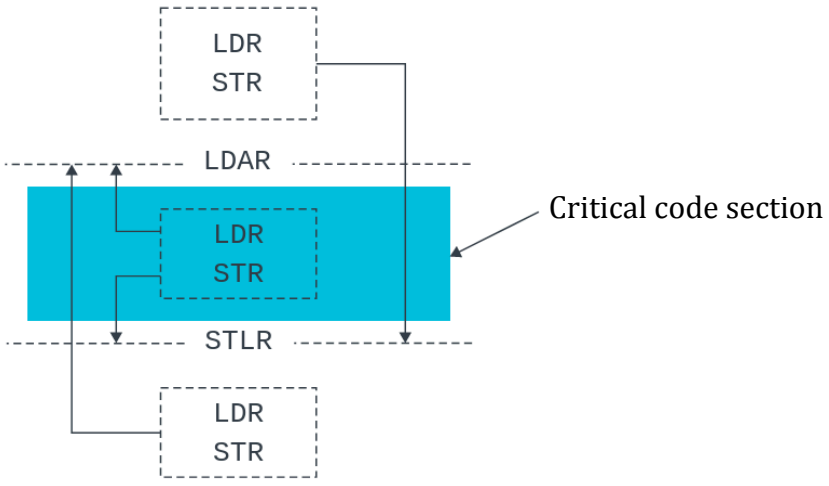
Per le operazioni atomiche abbiamo creato 2 threads che eseguono in un ciclo 10 incrementi ad una variabile condivisa, in modo che la zona critica fosse più lunga e permettesse ai 2 thread di sovrapporsi.
Dopo aver analizzato la simulazione di gem5, ci siamo resi conto che il loro funzionamento era strettamente legato al modello di memoria delle cache, basandosi sul fatto che un entry nello stato M invalidasse quelle di tutti gli altri e la tenesse nello stato M fino al completamento dell'operazione atomica, seguendo una logica riportata nel sequence diagram seguente.


Atomic operations
L1 C1
L2
L1 C0
0x898b4
GETX state: M
AMO
GETX 0x898b4
state: I
issueGETX
next_state: IM
GETX 0x898b4
state: MT
Fwd_GETX 0x898b4
forward_eviction_to_cpu
sendDataToRequestor
next_state: I
writeDataToL1Cache 0x898b4
0x898b4
state: M
AMO

Fences

```c
#include <stdio.h>
#include <atomic>
#include <pthread.h>
#include <gem5/m5ops.h>
#define DIM 500

using namespace std;
pthread_spinlock_t sp_lock;
int start = 0;
long long prec_elements[DIM] = {0};
//atomic<long long> My_values[DIM] = {0};
long long my_values[DIM] = {0};
unsigned int i = 0;

void do_stuff()
{
        int a=1;
        int b=2;
        for(int j=0; j<100; j++)
        {
                b = (a-b)*(a+b);
        }
}

void *producer(void *par)
{
        int local_start = -1;
        while(local_start != 1)
        {
                pthread_spin_lock(&sp_lock);
                if(local_start==-1)
                {
                        local_start=0;
                        ++start;
                }
                if(start>=2)
                        local_start=1;
                pthread_spin_unlock(&sp_lock);
        }
        printf("Inizio\n");

        for(i=0; i<DIM; i++)
        {
                do_stuff();
                prec_elements[i] = i/1.321;
                //My_values[i].store(i/1.321, memory_order_release);
                my_values[i] = i/1.321;
        }
        return NULL;
}

void *consumer(void *par)
{
        unsigned int counter=0;
        long long prec = -1;
        long long val = -1;
        unsigned int index = 0;
        int local_start = -1;
        while(local_start != 1)
        {
                pthread_spin_lock(&sp_lock);
                if(local_start==-1)
                {
                        local_start=0;
                        ++start;
                }
                if(start>=2)
                        local_start=1;
                pthread_spin_unlock(&sp_lock);
        }

        index = i;
        while(index < DIM)
        {
                val = my_values[index];
                //val = My_values[index].load(memory_order_acquire);
                prec = prec_elements[index];
                if(val!=0 && prec==0)
                {
                        ++counter;
                        printf("prec_element  |  new_value:  %lld  |  %lld\n", prec, val);
                }
                index = i;
        }
        printf("Number of memory order inconsistency: %lld\n", counter);
        return NULL;
}

int main()
{
        pthread_t thread_1, thread_2;

        if(pthread_spin_init(&sp_lock, PTHREAD_PROCESS_PRIVATE) != 0)
        {
                printf("ERROR in initializing spin lock!\n");
                return -1;
        }

        m5_switch_cpu();
        pthread_create(&thread_1, NULL, producer, NULL);
        pthread_create(&thread_2, NULL, consumer, NULL);
        pthread_join(thread_1, NULL);
        pthread_join(thread_2, NULL);
        fflush(stdout);
        if(pthread_spin_destroy(&sp_lock) != 0)
        {
                printf("ERROR in initializing spin lock!\n");
                return -1;
        }
//      m5_exit(0);
        return 0;
}
```

Per testare le fences abbiamo sfruttato la logica di acquire release così che si potessero testare anche delle fence più deboli oltre alla classica istruzione “fence” in grado di impedire il reordering in entrambi i versi.



LDAR e STLR rappresentano l’acquire e la release, mentre LDR e STR rappresentano la load e la store.

In assembly la store acquire e load release vengono tradotte tramite in fence e decoratori di istruzione (.aq e .rl):

- acquire viene tradotto come una fence seguita da un'operazione atomica marchiata come acquire, la quale impedisce qualsiasi operazione successiva di venire riordinata e completata prima dell'operazione atomica.
  In questo modo si ottiene un ordine completo, al pari di un'operazione circondata da 2 fences

```
10696:       0f50000f                fence   iorw,ow
1069a:       0ce7b02f                amoswap.d.aq    zero,a4,(a5)
```

- release viene tradotta come una semplice load preceduta e seguita da 2 fences, così da forzare un ordine completo per quest'operazione

```
10794:       0ff0000f                fence
10798:       639c                    ld      a5,0(a5)
1079a:       0ff0000f                fence
```

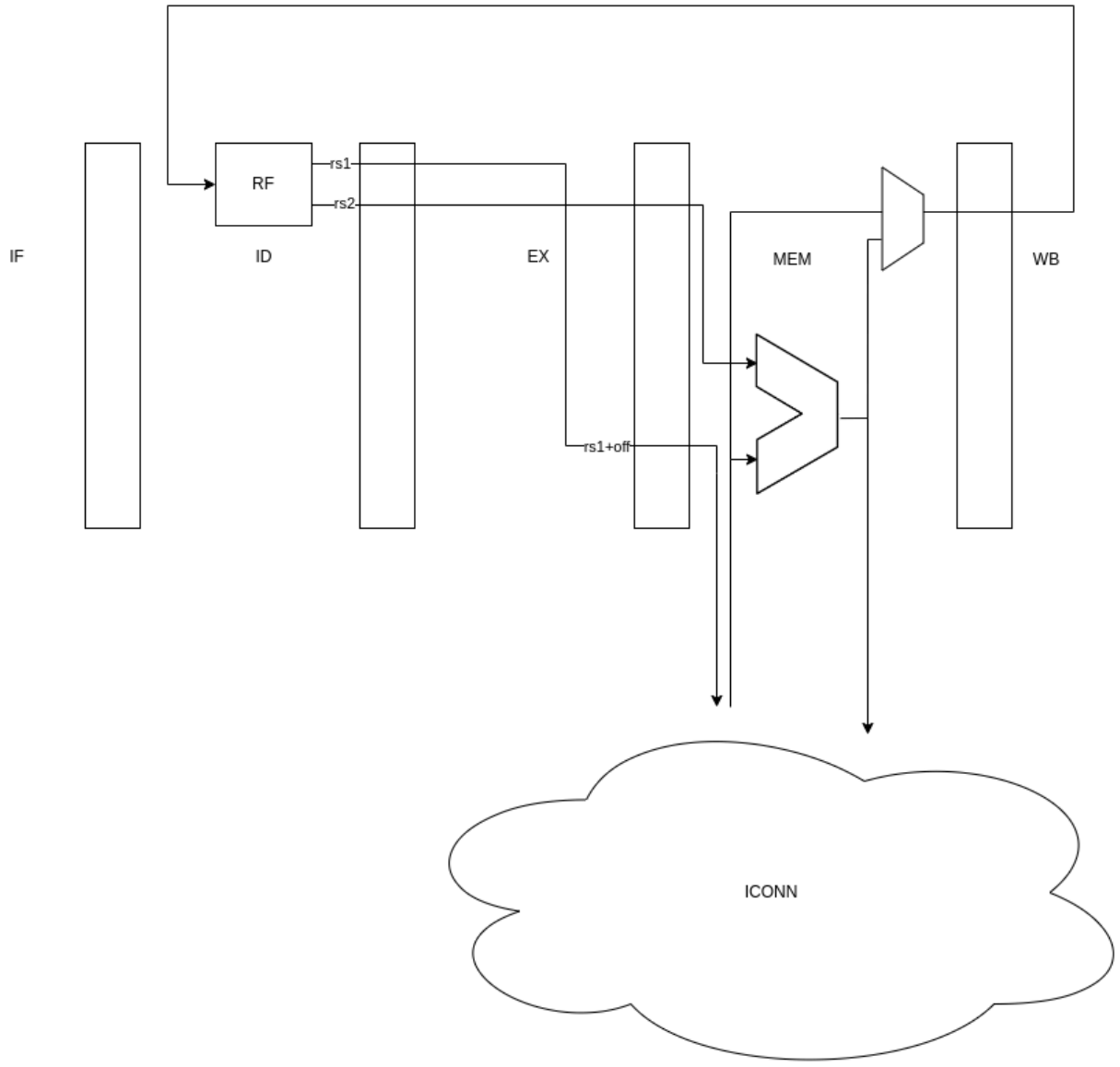
Dopo aver eseguito vari test si può notare come senza e fences create dalla logica di acquire-release il thread consumer è rileva numerose inconsistenze di memoria

```
==== m5 terminal: Terminal 0 ====
Inizio
prec_element  |  new_value:  0  |  182
prec_element  |  new_value:  0  |  191
prec_element  |  new_value:  0  |  196
prec_element  |  new_value:  0  |  200
prec_element  |  new_value:  0  |  204
prec_element  |  new_value:  0  |  208
prec_element  |  new_value:  0  |  215
prec_element  |  new_value:  0  |  219
prec_element  |  new_value:  0  |  227
prec_element  |  new_value:  0  |  230
prec_element  |  new_value:  0  |  239
prec_element  |  new_value:  0  |  246
prec_element  |  new_value:  0  |  252
prec_element  |  new_value:  0  |  254
prec_element  |  new_value:  0  |  256
prec_element  |  new_value:  0  |  258
prec_element  |  new_value:  0  |  288
prec_element  |  new_value:  0  |  290
prec_element  |  new_value:  0  |  300
prec_element  |  new_value:  0  |  302
prec_element  |  new_value:  0  |  305
prec_element  |  new_value:  0  |  307
prec_element  |  new_value:  0  |  312
prec_element  |  new_value:  0  |  314
prec_element  |  new_value:  0  |  318
prec_element  |  new_value:  0  |  367
prec_element  |  new_value:  0  |  369
Number of memory order inconsistency: 27
```

Implementazioni

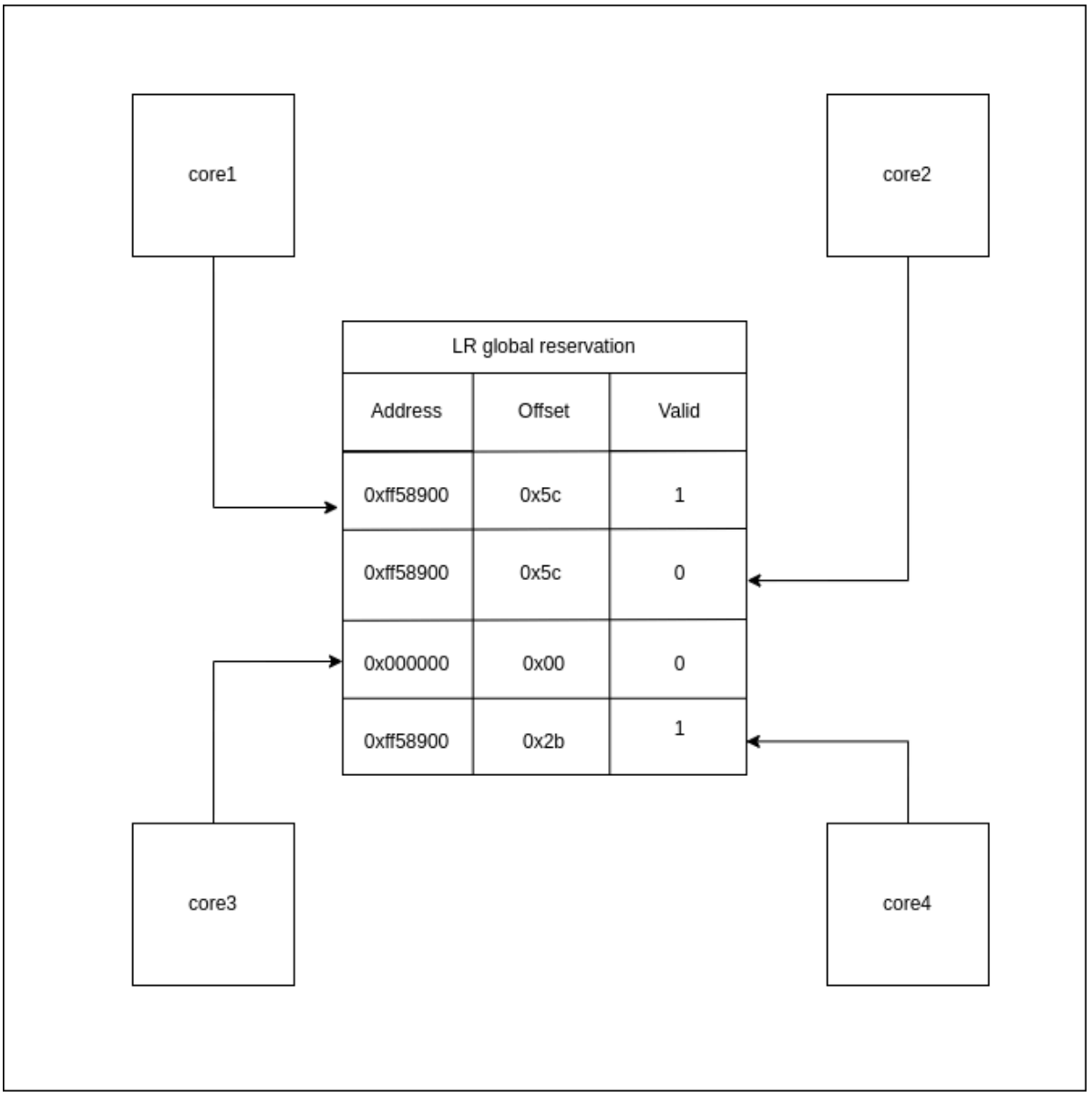
Atomic

La difficoltà nell'implementare istruzioni atomiche in una CPU sta nel riuscire a eseguire multiple operazioni senza stravolgere la pipeline delle istruzioni ordinarie.
Una soluzione è inserire una piccola ALU, capace di eseguire solamente un subset di istruzioni, che rappresenta il numero di operazioni atomiche concesse da RISC-V, ovvero: SWAP, ADD, AND, OR, XOR, MAX e MIN.
Per garantire coerenza all'interno del processore, la pipeline modificata sarà in grado di eseguire nello stesso ciclo di pipeline un accesso a memoria per recuperare il dato che verrà successivamente elaborato dalla ALU e salvato in memoria tramite una store.
Come si può notare dall'esempio riportato, la soluzione riesce a calcolare rs1+offset nello stadio di EX, accedere alla memoria tramite l'interconnect, nello stadio di MEM, e utilizzare il risultato come input della nuova ALU, per poi andare a salvare finalmente il risultato in memoria ancora tramite l'interconnect.



LL SC

Per porter supportare una architettura che faccia uso di istruzioni Load-link/store-conditional, abbiamo pensato ad una soluzione che prevede una nuova struttura di memoria, condivisa tra i vari core del processore.
Questa struttura prevede una sola entry per ogni core, e ognuna contiene l'indirizzo di una pagina di memoria, un offset che si riferisce all'indirizzo su cui la LL ha preso una reservation, e un bit di validità per indicare se il core in questione ha ancora la reservation su quella regione di memoria.
Ogni volta che un core esegue una LL, questo andrà ad aggiornare la propria entry nella tabella, e provvede ad invalidare tutte le righe degli altri core che avevano privilegi sullo stesso indirizzo.
A questo punto, prima di eseguire la store conditional, il core dovrà controllare di avere il bit di validità settato ad 1, altrimenti la store conditional fallirà e la branch che la segue riporterà l'esecuzione del codice alla load link.



Fences

Per implementare le fence instructions abbiamo pensato di aggiungere un flip-flop il quale viene settato a 1 non appena viene fatta la fetch di un'istruzione di tipo fence, e non rimane settato a 1 finché la pipeline e tutte il buffer contenente tutte le operazioni di memoria, non siano vuoti.
In questo modo riusciamo ad assicurare che tutte le istruzioni antecedenti alla fence vengano completate prima dell'esecuzione di una qualsiasi istruzione successiva, impedendo così qualsiasi tipo di reordering a livello di memoria e di istruzioni.

```
if(!(fence_instr ^ fetcher_out))
begin
      fence_n =1;
end
else
begin
      fence_n = fence_r && !(if_id_free && id_ex_free && ex_mem_free &&
                             mem_wb_free && mem_op_buf_free);
end
```