A Custom Thread Library Built on Native Linux Threads for Faster Embedded System Simulation

Tony Mathew, Rainer Dömer

Technical Report CECS-11-10
December 14, 2011

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
(949) 824-8919

tmathew@uci.edu
doemer@uci.edu

A Custom Thread Library Built on Native Linux Threads for Faster Embedded System Simulation

Tony Mathew, Rainer Dömer

Technical Report CECS-11-10
December 14, 2011

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
(949) 824-8919

tmathew@uci.edu
doemer@uci.edu

Abstract

Embedded system simulation has become very time expensive in recent times due to the increasing complexity of system models. Efficient and fast simulations play a crucial role in the embedded system development. The current SpecC simulator uses the Posix thread library, Quickthreads or Win32 libraries to achieve multithreading. This report proposes the usage of a custom thread library based on native linux primitives to achieve the same functionalities. Our proposed custom thread library named Litethreads is developed with the goal of having the advantages of both user-level threads, like Quick threads, and kernel-level threads, like Pthreads. Preliminary simulation results indicate significant improvement in simulation time for some design samples including an mp3 decoder.

Contents

List of Figures

List of Tables

List of Listings

A Custom Thread Library Built on Native Linux Threads for Faster Embedded System Simulation

T. Mathew, R. Dömer
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
tmathew@uci.edu
doemer@uci.edu

Abstract

Embedded system simulation has become very time expensive in recent times due to the increasing complexity of system models. Efficient and fast simulations play a crucial role in the embedded system development. The current SpecC simulator uses the Posix thread library, Quickthreads or Win32 libraries to achieve multithreading. This report proposes the usage of a custom thread library based on native linux primitives to achieve the same functionalities. Our proposed custom thread library named Litethreads is developed with the goal of having the advantages of both user-level threads, like Quick threads, and kernel-level threads, like Pthreads. Preliminary simulation results indicate significant improvement in simulation time for some design samples including an mp3 decoder.

1 Introduction

In this report, we discuss a custom thread library developed using native linux primitives. TheSpecC [5] simulator currently uses two types of thread libraries. Quickthread library has very low overhead, but as these are only userlevel threads on simultaneous multiprocessing machines (SMP), it becomes inefficient as it runs in context of a single process. Posix threads [10] carry more overhead while creating and controlling threads. Whereas many options are available to the user like setting behavior and type of mutexes. It has the advantage of schedulability to different cores on SMP machines. We didn't consider an improvisation on win32 [9] threads instead concentrated more on linux based environment.

Our proposed custom thread library named Litethreads utilizes some primitives provided by Linux like futex (Fast User Space Mutex), and clone system call to achieve the features offered by other thread libraries. The motivation is to merge the good components of both Quickthreads [7] and Pthreads, hence obtaining a thread library with low overhead and multi-core schedulability. Cutting down features of Posix thread library and tuning it for use in SpecC simulator is complex as Pthreads is integrated with glibc which has large code space and dependencies.

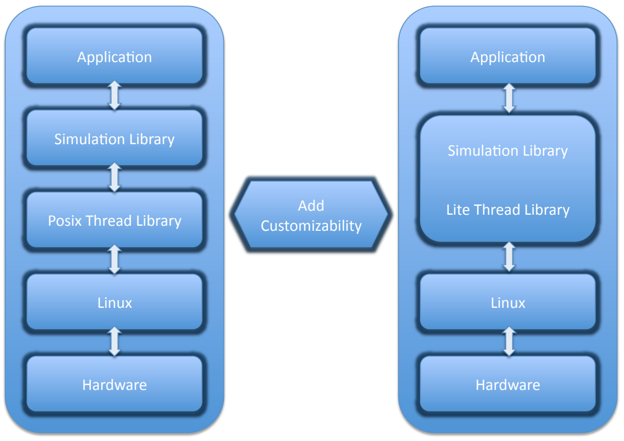

Figure 1: System Level Design Environment

The software stack of a system level simulation environment [2] is depicted as shown in the Figure 1. The application uses simlib library provided by

SpecC development environment. The simulation library in turn invokes Pthread APIs to achieve various functionalities like waiting, and creating of threads. The Pthread library uses Linux APIs to obtain the thread functions. Our custom Litethread library will help to enhance the efficiency of simulation library by providing more visibility of the environment and ways of tweaking the environment depending upon the behavior of the application that is being simulated.

2 Thread synchronization

Thread synchronization for Litethreads is achieved using futexes [4]. SpecC simulator uses mainly mutexes and condition variables, which are implemented at this point.

2.1 Futex - Fast user space mutex

Futexes are light-weight, linux constructs that can be used to implement synchronization primitives like user level locks, semaphores, condition variables. Futexes [3] in non contended case acquires and releases lock without kernel intervention. Futexes are basis for several mutual exclusion constructs used in multi threaded programming including Native Posix Thread Library (NPTL) [10]. Futexes are implemented using sytem call `SYS_Futex`. The Linux manual pages denes futex as:

> ”The `futex()` system call provides a method for a program to wait for a value at a given address to change, and a method to wake up anyone waiting on a particular address (while the addresses for the same memory in separate processes may not be equal, the kernel maps them internally so the same memory mapped in different locations will correspond for `futex()` calls).”

Futexes can be used to store the state of a lock and provide a kernel wait queue for tasks blocking on the lock. To minimize system call overhead, this state should allow for atomic lock acquisition when the lock is uncontended.

The syntax of futex call is shown in Listing 1.The function parameters comprises of `addr1`, which points to a user space address, that correspond to the futex. The `op` parameter can take different values that decide on whether to wake up or wait for the futex at `addr1`. Parameter `val1` decides on when to perform an action on the futex at `addr1` by comparing to the value at `addr1`.

```
int sys_futex(void *addr1, int op,

  int val1, struct timespec *timeout,

    void *addr2, int val3)
{

   return syscall(SYS_futex, addr1, op,
          val1, timeout, addr2, val3);
}
```

Listing 1: Function prototype for futex system call.

The working of futex can be summarised as below:

1. User level thread can acquire the futex when the lock is free. While releasing lock, if the futex is uncontended, the thread can release the lock. Hence, in both cases we can avoid invoking the system call.

2. A lock request is made by setting the argument `op` value to `FUTEX_WAIT`. `addr1` parameter is set to virtual address of the user space lock. The value which has to be be compared against the value at `addr1` is passed in as `val1` argument. If it is different, then the queue in the futex hash table is evaluated using the address as key. The thread is then queued in the table until it is woken up by some other thread.

3. An unlock request is made by passing the virtual address of the lock and `FUTEX_WAKE` as `op` parameter. The number of threads to be woken up can also be set using the parameter `val1`. The threads are removed from the wait queue in the kernel when this system call is invoked.

4. Detailed information about the working of futex can be found in futex man page.

We can find that Futex provides many advantages. There are no explicit limits on how many futexes one

```
union mutex
{
  unsigned u;
  Struct
  {
   unsigned char locked;
   unsigned char contended;
  } b;
}
```

Listing 2: Basic mutex datatype.

can create, nor can one futex user starve other users of futexes. This is because futex is merely a memory location like any other until the system call is invoked. Also, by invoking `sys_futex` call, we are pinning maximum only one page per process in the worst case.

2.2 Implementation

The main synchronization primitives that we implemented are mutex and condition variables. For implementing the mutex we use a union which has a prototype as shown in Listing 2. A value of 0 in the union denotes an unlocked and not contended mutex. Value of 1 denotes locked mutex that is not contended. Value of 256 denotes an unlocked but contended mutex, and 257 denotes that the mutex is both locked and contended.

The mutex lock and unlock functions are implemented as shown in Listing 3 and Listing 4. It can be noticed that in both functions we have two loops which makes the thread spin for some time in the user space before invoking the system call. These parameters can be optimized further depending upon the behavior and level of contention of the application that is using the futex. It could also be found that we are using atomic instructions that makes the switching of the lock fast and safe. These are currently specific to the x86 instruction set architecture, hence have to be adapted to be used in other architectures.

In mutex lock, we first try to grab the lock by trying to set locked byte. If this is not successful, then a futex system call is invoked on the address location and marks the location as contended also. In unlock operation, we first check if the lock is not contended

```
int litethread_mutex_lock(
                    litethread_mutex *m)
{
   int i;

   /* Try to grab lock */
   for (i = 0; i < 100; i++)
   {
      if (!xchg_8(&m->b.locked, 1))
      return 0;

      cpu_relax();
   }

   /* Have to sleep */
   while (xchg_32(&m->u, 257) & 1)
   {
      litethread_sys_futex(
        m, FUTEX_WAIT, 257, NULL, NULL, 0);
   }

   return 0;
}
```

Listing 3: Basic mutex lock function.

and unlock it, if uncontended. If it is contended, then we unlock it and spin, and check if somebody grabs the lock in the meantime. If nobody grabs the lock during the spin time, then we go on to invoke a futex call to wake any thread who is waiting on the mutex.

Condition variables are implemented using futex system calls by having an extra counter variable for each conditional variable object. For implementing conditional wait, the thread waits on the value of the counter by invoking `FUTEX_WAIT`. For conditional signal, the thread increments the counter value to wake up some other thread, if any one was waiting for that counter variable with some other value. The code snippets for both functions are as shown in Listing 5.

3 Thread control in System Level Simulator

Thread control in SpecC simulator mainly comprises of creation, deletion and joining of individual threads. For this, we have to implement something similar to `pthread_create` and `pthread_destroy`. For

```
int litethread_mutex_unlock(
                    litethread_mutex *m)
{
    int i;

    /* Locked and not contended */
    if ((m->u == 1) &&
        (cmpxchg(&m->u, 1, 0) == 1))
        return 0;

    /* Unlock */
    m->b.locked = 0;

    barrier();

/*Spin and hope someone takes the lock*/
    for (i = 0; i < 200; i++)
    {
        if (m->b.locked) return 0;

        cpu_relax();
    }

    /* We need to wake someone up */
    m->b.contended = 0;

    litethread_sys_futex(
      m, FUTEX_WAKE, 1,NULL, NULL, 0);

    return 0;
}
```

Listing 4: Basic mutex unlock function.

```

int litethread_cond_wait(litethread_cv *c,
                    litethread_mutex *m)
{
    int seq = c->seq;

    litethread_mutex_unlock(m);

    litethread_sys_futex(
    &c->seq, FUTEX_WAIT, seq, NULL, NULL, 0);

    litethread_mutex_lock(m);

    return 0;
}
int litethread_cond_signal(litethread_cv *c)
{
    /* We are waking someone up */
    atomic_inc(&c->seq);

    /* Wake up a thread */
    litethread_sys_futex(
        &c->seq, FUTEX_WAKE, 1, NULL, NULL, 0);

    return 0;
}
```

Listing 5: Basic condition variable with wait and signal functions.

creation, we use a Linux system call named clone. According to Linux man page

> ”clone creates a new process similar to fork, it is actually a library function layered on top of `SYS_clone` system call.”

The difference between clone and fork is that it lets the user decide on how much sharing should be there between the parent and child process. Multiple threads, that share the memory space can, be implemented using clone. Before invoking clone system call, we need to allocate a stack space which has to be used for the child process. This is passed as a parameter to the clone system call. Linux also provides many other flags which can be used to obtain different functionality using clone system call.

3.1 Thread Creation

The prototype thread creation function using clone system call is shown in Listing 6. We make use of flags like below which controls different factors between parent and child thread.

- `CLONE_IO` - Share IO descriptors
- `CLONE_FS` - Share file system information
- `CLONE_FILES` - Share file descriptor table
- `CLONE_SIGHAND` - Share signal handle table
- `CLONE_VM` - Share same memory space
- `CLONE_CHILD_SETTID` - Store the child thread id at location refered by childpidptr
- `CLONE_CHILD_CLEARTID` - Clear the thread id from the childpidptr location when child exits
- `CLONE_THREAD` - Create independent threads rather than processes with new process ids.
- `CLONE_THREAD` - The new thread created is in the same thread group as that of the parent process.

```
int litethread_create(int (*fn)(void *),

void *args, void ** stack, int *stackStat,

                            int *child_pidptr)
{
  void * stacka =
  malloc(SIM_THREAD_STACK_SIZE);
  *stack=stacka;

  if (stacka){

  *stackStat=1;

  return clone(

  fn, (char*)stacka+SIM_THREAD_STACK_SIZE,

  CLONE_CHILD_CLEARTID | CLONE_SIGHAND |

  CLONE_VM | CLONE_THREAD | CLONE_FILES |

  CLONE_DETACHED | CLONE_IO | CLONE_FS |

  CLONE_CHILD_SETTID , args, NULL, NULL,

                              child_pidptr);
  }
}
```

Listing 6: Thread creation using clone system call.

3.2 Thread Abortion and Join

SpecC has constructs like `try catch` that requires the feature of aborting an existing thread. In a Pthread environment, this can be implemented using thread cancel and thread join functions. In Litethreads, the join functionality is achieved by utilising the value stored in the `childptr` location. We probe on this address for the thread id. As we used the clear `tid` flag in clone call whenever the thread exits, contents of this address location are cleared. Hence, we can use a futex system call to wait until the contents of `childptr` are changed to 0 to achieve the functionality of join.

Another alternative method for achieving the same functionality of join was to use the `wait()` call invoked by the parent thread to join with the child. For achieving this behavior we should avoid using `CLONE_THREAD` option in the clone system call.

With this approach, every clone call spawns a new process resulting in the pid numbers growing with the number of threads in the system. This is not favorable. Also, if some running thread calls exit, this has to cleanly exit the whole application rather than just the calling thread. But as in this approach, each thread itself is considered as a process by linux, a call to exit can result in abortion of the calling thread that leaves lots of unfinished threads in the system. To cleanly exit all these remaining threads, we have to have separate exit handling mechanism using `at_exit` construct. This is an overhead. Hence, in Litethreads we follow earlier method using futex to implement join.

With the use of `CLONE_THREAD` flag in Litethreads, we have a flat structure for the threads in the system. Hence for achieving thread cancelling we investigated the use of `tgkill` function which wasn't successful. Hence, to achieve abortion of threads, we used `longjmp` functionality. When abort is invoked on particular thread, a long jump is signalled on the thread to be killed and the calling thread will wait to join with the killed thread. The thread to be killed, upon signaled with long jump, does a far jump to the exit of the thread and returns from the clone system call hence signaling any thread who was waiting for the killed thread. The usage of `tgkill` will avoid this overhead but how to make `tgkill` work is not known at this point.

We can construct higher level functions like threadcreate, threadrun, threadabort, etc. using above mentioned thread synchronization and control mechanisms that can be in turn used by SpecC simulator.

4 Experiments and Results

At this stage, we have performed only minimal measurements integrating the Litethread library to the SpecC simulator. All measurements are performed on a Intel(R) Core(TM)2 Quad CPU, 3.00GHz, machine.

Currently we have tested Litethread library for three main applications. First example project tested is MP3 decoder [8]. Table 1 shows the timing comparison between the performance of the simulation. It could be found that there is an improvement of around 5% when using litethreads. The cpu utilisation of lite threads is always found to be higher for mp3 decoder example.

Next example code which was used for measurement is JPEG encoder [11] Table 2 summarises the timings obtained for this case while using Litethread and Pthread and Quickthread library. The percentage of CPU used by Litethreads is again found to be higher. But not much of timing improvisation is found for this case except for the cases where Litethreads achieves more cpu utilisation.

Third example to that was investigated was Vocoder for GSM [6].In this case, it could be found that Litethread performs better than Pthread at times, but not consistently. More analysis has to be done for this kind of behavior. Table 3 provides the timing measurements for this case.

5 Conclusion and Future Work

How a custom thread library can be built around native linux primitives is discussed in this document. It could be found that we were successful in implementing such a custom thread library and have got preliminary results using the thread library on various examples. Currently, the testing has been done just for the available examples. More examples which are specifically designed to test the multithreading efficiency, that involves less I/O, has to be used to test how much much more efficient our proposed thread library is.

Also one constraint which we plan to address in future is attaching the thread to a particular core to get more efficiency. To achieve this, we have to use the parallel simulator [1] extended with options of setting affinity to particular processor cores with the threads.

Finally, the right amount of spin locking in the mutex and condition variable implementations depends on the nature of the application. This is also a possible area of further investigation to make these parameters, that are currently fixed, to be dependent upon the program nature.

6 Acknowledgement

The authors would like to thanks Jeff White for his initial exploration of using native Linux threads as an alternative to Posix threads.

Table 1: Mp3 Decoder Example(time in seconds)

Pthreads				LiteThreads				QuickThreads				Litethread vs(%)	
User	Sys	Total	Cpu%	User	Sys	Total	Cpu%	User	Sys	Total	Cpu%	Posix	Quick
0.35	0.13	0.59	81	0.31	0.14	0.53	85	0.28	0.01	0.30	95	+10	-43
0.34	0.13	0.58	82	0.32	0.12	0.52	84	0.28	0.01	0.29	98	+10	-46
0.36	0.12	0.58	82	0.29	0.15	0.53	84	0.28	0.00	0.29	98	+8	-44
0.33	0.14	0.57	83	0.31	0.14	0.53	85	0.29	0.00	0.30	99	+8	-43
0.35	0.13	0.58	83	0.29	0.16	0.53	85	0.28	0.01	0.29	99	+8	-45

Table 2: Jpeg Encoder Example(time in seconds)

Pthreads				LiteThreads				QuickThreads				Litethread vs(%)	
User	Sys	Total	Cpu%	User	Sys	Total	Cpu%	User	Sys	Total	Cpu%	Posix	Quick
0.04	0.13	0.23	77	0.04	0.10	0.21	67	0.01	0.00	0.02	81	+8	-90
0.04	0.12	0.23	74	0.04	0.09	0.19	66	0.01	0.00	0.02	72	+15	-89
0.04	0.13	0.24	75	0.05	0.10	0.22	68	0.01	0.00	0.02	80	+8	-91
0.03	0.08	0.12	91	0.03	0.08	0.14	83	0.01	0.00	0.02	90	-15	-85
0.05	0.11	0.21	80	0.03	0.09	0.20	66	0.01	0.00	0.02	81	+4	-90

This work has been supported in part by funding from the National Science Foundation under research grant NSF Award #0747523. We thank the NSF for this valuable support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] R. Dömer, W. Chen, X. Han, and A. Gerstlauer. Multi-core parallel simulation of system-level description languages. In *Proceedings of the Asia and South Pacific Design Automation Conference 2011*, Yokohama, Japan, January 2011.

[2] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. Gajski. System-on-chip environment: A specc-based framework for heterogeneous mpsoc design. In *EURASIP Journal on Embedded Systems, vol. 2008, article ID 647953*, Irvine, USA, July 2008.

[3] Ulrich Drepper. Futexes are tricky. In *Futexes are Tricky*, Red Hat Inc, Japan, December 2005.

[4] Mutexes and Condition Variables using Futexes. http://locklessinc.com/articles/mutex_cv_futex/.

[5] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[6] Andreas Gerstlauer, Shuqing Zhao, Daniel D. Gajski, and Arkady M. Horak. SpecC system-level design methodology applied to the design of a GSM vocoder. In *Proceedings of the Workshop of Synthesis and System Integration of Mixed Information Technologies*, Kyoto, Japan, April 2000.

[7] D. Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, May 1993.

[8] R. Dömer P. Chandraiah. Specification and design of a mp3 audio decoder. Technical Report CECS-TR-05-04, May 2005.

[9] J. Richter. Advanced windows nt: The developer's guide to the win32 application programming interface. Technical Report Microsoft-Press, January 1994.

Table 3: Vocoder Example(time in seconds)

Pthreads				LiteThreads				Litethread vs(%)
User	Sys	Total	Cpu%	User	Sys	Total	Cpu%	Posix
1.28	0.41	01.88	89	1.35	0.45	01.84	97	+2
1.31	0.47	02.04	87	1.35	0.69	02.34	87	-15
1.37	0.83	02.91	75	1.32	0.60	02.06	93	+29
1.32	0.89	02.94	75	1.34	0.92	02.43	93	+17
1.29	0.42	01.93	87	1.30	0.57	02.08	90	-7

[10] Ingo Molnar Ulrich Drepper. The native posix thread library for linux. In *The Native POSIX Thread Library for Linux*, Red Hat Inc, February 2005.

[11] Hanyu Yin, Haito Du, Tzu-Chia Lee, and Daniel D. Gajski. Design of a JPEG encoder using SpecC methodology. Technical Report ICS-TR-00-23, July 2000.