Center for Embedded Computer Systems
University of California, Irvine

Performance Evaluation and Optimization of A Custom Native Linux Threads Library

Guantao Liu and Rainer Dömer

Technical Report CECS-12-11
October 3, 2012

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{guantaol, doemer}@uci.edu
http://www.cecs.uci.edu/

Performance Evaluation and Optimization of A Custom Native Linux Threads Library

Guantao Liu and Rainer Dömer

Technical Report CECS-12-11
October 3, 2012

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{guantaol, doemer}@uci.edu
http://www.cecs.uci.edu

Abstract

The current SpecC simulator utilizes PosixThreads, QuickThreads or a custom native Linux thread library named LiteThreads to perform thread manipulation. While QuickThreads is very efficient as a user-level thread library and PosixThreads supports multithreading and the parallel simulator, the proposed LiteThreads library combines the advantages of both thread libraries and aimes to achieve a significant improvement in simulation time. In this report, we will present the performance evaluation of the LiteThreads library based on two featured benchmarks. In addition, more work is done on optimizations of context switching and stack space allocation. With these improvements, the LiteThreads library achieves better performance than PosixThreads for the sequential simulator. The same conclusion is also true on 64-bit Linux machines, as verified by our simulation results.

Contents

1 Introduction **1**

2 Performance and Optimizations on Context Switching **2**
2.1 Experiments and Results 2

3 Performance and Optimizations on Thread Creation and Deletion **6**
3.1 Stack Space Allocation 7
3.2 Thread Creation and Deletion Benchmark 11
3.3 Experiments and Results 13

4 Performance on 64-bit Architectures **14**

5 Conclusion and Future Work **15**

References **21**

List of Figures

1 Intel Pentium 4 architecture, 2.4 GHz (alpha) 3
2 Intel Pentium 4 architecture, 3.0 GHz (epsilon) 3
3 Intel Core 2 Quad architecture, Q9650 (mu) 4
4 Intel Xeon architecture, X5650 (xi) 4

List of Tables

1 Simulation Results of Producer-Consumer Model on alpha and epsilon (LiteThreads loops=100, 200) . . . 5
2 Simulation Results of Producer-Consumer Model on mu and xi (LiteThreads loops=100, 200) . . . 6
3 Simulation Results of Producer-Consumer Model (LiteThreads loops=0, 0) . . . 7
4 Simulation Results of Producer-Consumer Model (LiteThreads loops=0, 200) . . . 8
5 Simulation Results of Producer-Consumer Model (LiteThreads loops=100, 0) . . . 9
6 Simulation Results of Producer-Consumer Model (LiteThreads loops=1000, 0) . . . 10
7 Simulation Results of Producer-Consumer Model (LiteThreads loops=1000, 2000) 11
8 Simulation Results of TFMUL on alpha . . . 13
9 Simulation Results of TFMUL on epsilon . . . 14
10 Simulation Results of TFMUL on mu . . . 15
11 Simulation Results of TFMUL on xi . . . 16
12 Simulation Results of Producer-Consumer Model on 64-bit Architectures . . . 16
13 Simulation Results of TFMUL on 64-bit Architectures . . . 17
14 Simulation Results of Producer-Consumer Model (cores=0 1, loops=0 0) . . . 17
15 Simulation Results of Producer-Consumer Model (cores=0 1, loops=100 0) . . . 18
16 Simulation Results of Producer-Consumer Model (cores=0 1, loops=0 200) . . . 18
17 Simulation Results of Producer-Consumer Model (cores=0 1, loops=100 200) . . . 19
18 Simulation Results of Producer-Consumer Model on xi(same physical core, different logical cores, loops=0 0) . . . 19
19 Simulation Results of Producer-Consumer Model on xi(same physical core, different logical cores, loops=100 0) . . . 20
20 Simulation Results of Producer-Consumer Model on xi(same physical core, different logical cores, loops=0 200) . . . 20
21 Simulation Results of Producer-Consumer Model on xi(same physical core, different logical cores, loops=100 200) . . . 21

List of Listings

1 Thread Initialization with Constant-time Stack Space Allocation 7
2 Thread Creation with Constant-time Stack Space Allocation 8
3 Intensive Thread Creation/Deletion Benchmark 11

Performance Evaluation and Optimization of A Custom Native Linux Threads Library

Guantao Liu and Rainer Dömer

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

{guantaol, doemer}@uci.edu
http://www.cecs.uci.edu

Abstract

The current SpecC simulator utilizes PosixThreads, QuickThreads or a custom native Linux thread library named LiteThreads to perform thread manipulation. While QuickThreads is very efficient as a user-level thread library and PosixThreads supports multithreading and the parallel simulator, the proposed LiteThreads library combines the advantages of both thread libraries and aimes to achieve a significant improvement in simulation time. In this report, we will present the performance evaluation of the LiteThreads library based on two featured benchmarks. In addition, more work is done on optimizations of context switching and stack space allocation. With these improvements, the LiteThreads library achieves better performance than PosixThreads for the sequential simulator. The same conclusion is also true on 64-bit Linux machines, as verified by our simulation results.

1 Introduction

Nowadays, QuickThreads and PosixThreads are the two most popular thread libraries used on Linux machines. QuickThreads is user-level thread, which has very low overhead and is extremely efficient for a sequential simulator. As for the PosixThreads library, it is a kernel-level thread library and has more options to set behaviors and type of mutexes. Thus, Posix threads have the advantage of more schedulability on different cores of SMP machines. Basically speaking, PosixThreads library carries more overhead than QuickThreads.

In order to have the advantages of both thread libraries, a custom thread library named LiteThreads is built on native Linux threads primitives [1]. By utilizing the futex and clone system calls, other than the corresponding mutex and fork system call in PosixThreads, LiteThreads reduces the overhead of context switching and thread creation/deletion. In this way, LiteThreads could also be used in the parallel simulator, which is not supported by the QuickThreads library. Currently, all of these three thread libraries are used in the SpecC simulator.

In this report, we will first evaluate the performance of our custom LiteThreads library in the SpecC simulator, compared to the PosixThreads library and QuickThreads library. Two different kinds of benchmarks are used in the tests to evaluate the features of LiteThreads. With the premise of only using sequential simulator, we also optimize the LiteThreads library based on the simulation results. Finally, we carry out the same evaluation on a 64-bit machine to verify that LiteThreads library also achieves better performance than PosixThreads on this platform.

2 Performance and Optimizations on Context Switching

Both PosixThreads and QuickThreads have multiple features and options to support multithreading, but LiteThreads has two differences from PosixThreads: the clone system call and spinlocks used in the synchronization.

In the PosixThreads library, it is a time-consuming task to enter or exit the critical sections, which often spends lots of time in the context switching between the user level and kernel level. In order to reduce this overhead, LiteThreads makes use of spinlocks in the mutex_lock and mutex_unlock functions, which would avoid such context switching when some other threads are waiting to grab the lock.

In order to achieve better performance than PosixThreads, the spinlock in the LiteThreads library must be more efficient than the context switching between user level and kernel level. This overhead is decided by the loop iterations in the mutex_lock and mutex_unlock functions in LiteThreads. If the loop iteration (spin time) is too short, no other threads would grab the lock from the current thread, which means that the spin time is wasted and it still needs to switch to the kernel level; while the loop iteration (spin time) is too long, certain threads would finally grab the lock from the current thread, but the overhead of the spinlock would be larger than that of the context switching between user level and system level. In this case, the new feature is useless.

Therefore, the performance of the LiteThreads library is largely related to the two loop iterations in the mutex_lock and mutex_unlock. In our experiments, we will first compare the performance of the initial LiteThreads with PosixThreads and QuickThreads and then try to optimize the spin lock according to the simulation results.

2.1 Experiments and Results

For the current experiments, we use the Producer-Consumer model as benchmark and utilize the sequential simulator to run all the tests. The Producer-Consumer model in this case uses the double handshake protocol to communicate between the two agents and the Producer and Consumer locate in two different threads.

All the tests in these experiments are running on four 32-bit Linux machines, which have Intel(R) Pentium 4 architecture 2.40 GHz CPU (named alpha), Intel(R) Pentium 4 architecure 3.0 GHz CPU (named epsilon), Intel(R) Core(TM) 2 Quad architecture Q9650 3.0 GHz CPU (named mu) and Intel(R) Xeon(R) architecture X5650 2.66 GHz CPU (named xi), respectively.

The architectures of these four processors are indicated as Figure 1, 2 , 3 and 4. The dashed line in the figure means that the core has the hyperthreading feature enabled.

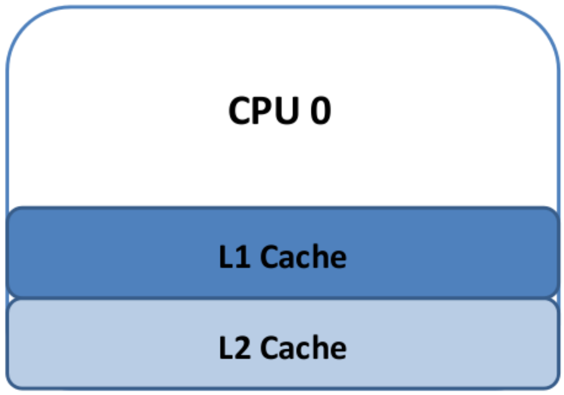

Figure 1: Intel Pentium 4 architecture, 2.4 GHz (alpha)

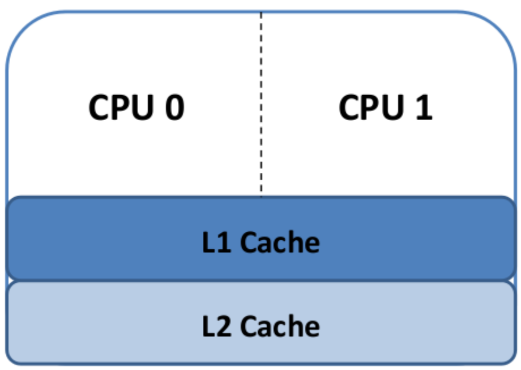

Figure 2: Intel Pentium 4 architecture, 3.0 GHz (epsilon)

As most of the processors have more than one core, the elapsed time of the simulation varies with the CPU affinity. In order to eliminate this variation in simulation time, we utilize the *taskset* Linux command to force the whole program to run on one logical core.

```
taskset -c 0 executable
```

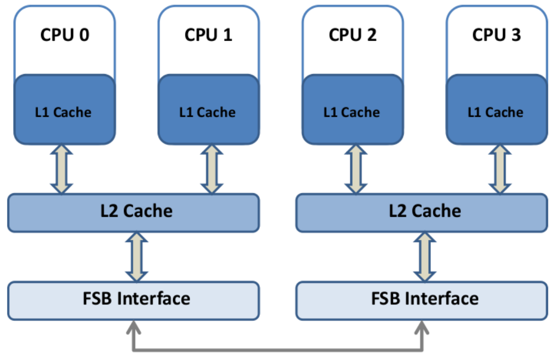

Figure 3: Intel Core 2 Quad architecture, Q9650 (mu)

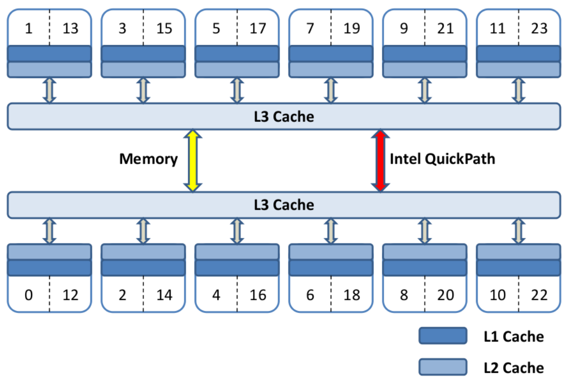

Figure 4: Intel Xeon architecture, X5650 (xi)

Using this command, we get consistent and reliable simulation results, as shown in Table 1, 2, 3, 4, 5, 6 and 7. The consistent simulation time and the 99% CPU loads indicate that the whole program indeed runs on one logical core.

The initial LiteThreads has the loop iterations of (100, 200) in the mutex_lock and mutex_unlock functions [1]. When compared with the other two thread libraries, it has slightly larger user time and elapsed time than PosixThreads, as indicated in Table 1 and 2. QuickThreads always has the best performance on the four servers, as long as we only use the sequential simulator. The zero system time and much smaller user time indicate that QuickThreads has no kernel-level overhead and its user-level scheduling is much more efficient than the other two.

For the initial LiteThreads, it seems that it is worse than PosixThreads. However, as we discussed in the previous section, the simulation time of LiteThreads largely relies on the loop iterations in the mutex_lock and mutex_unlock. Thus, with changes of the loop iterations, LiteThreads would achieve a smaller simulation time, as demonstrated in Table 3, and 4. In all the cases, the

Table 1: Simulation Results of Producer-Consumer Model on alpha and epsilon (LiteThreads loops=100, 200)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
alpha	10.1s	12.71s	22.87s	99.00%	LiteThreads
	10.45s	12.11s	22.62s	99.00%	
	9.41s	13.02s	22.49s	99.00%	
	9.75s	12.9s	22.83s	99.00%	
	10.38s	14.2s	24.65s	99.00%	
alpha	5.76s	13.5s	19.3s	99.00%	PosixThreads
	5.91s	13.24s	19.2s	99.00%	
	5.77s	13.25s	19.07s	99.00%	
	5.89s	13.36s	19.29s	99.00%	
	5.74s	13.22s	19s	99.00%	
alpha	0.69s	0	0.69s	99.00%	QuickThreads
	0.74s	0	0.74s	99.00%	
	0.7s	0	0.7s	99.00%	
	0.7s	0	0.71s	99.00%	
	0.69s	0	0.7s	99.00%	
epsilon	10.33s	14.59s	24.95s	99.00%	LiteThreads
	9.51s	13.62s	23.14s	99.00%	
	8.99s	13.92s	22.92s	99.00%	
	11.58s	12.94s	24.53s	99.00%	
	9.35s	13.83s	23.19s	99.00%	
epsilon	5.04s	15.38s	20.43s	99.00%	PosixThreads
	5.26s	15.8s	21.09s	99.00%	
	5.37	15.52s	20.93s	99.00%	
	5.41s	15.08s	20.51s	99.00%	
	5.71s	15.1s	20.83s	99.00%	
epsilon	0.57s	0	0.57s	99.00%	QuickThreads
	0.59s	0	0.59s	99.00%	
	0.59s	0	0.59s	99.00%	
	0.57s	0	0.57s	99.00%	
	0.57s	0	0.58s	99.00%	

LiteThreads library has identical system time as when the loop iterations are (100, 200), since the spin time only affects the user-level time. As the loop iteration in the mutex_unlock increases (from 0 to 200 and 2000), the user time of the simulation increments monotonically (in Table 3, 4, 7), while the spin time in the mutex_lock has no effect on the simulation time.

As we are using the sequential simulator, these phenomena are easily explainable. When the current thread is running in the program, no other thread can enter or exist the critical section during the spin lock of this thread. After releasing the lock, any thread could enter the critical section if no one else is executing. In such a case, the spin time in the mutex_unlock is wasted and the lock is always available when some thread wants to grab it. Thus, the user time in the simulation is linear to the spin time in the mutex_unlock and unrelated to the loop iteration in the mutex_lock.

Table 2: Simulation Results of Producer-Consumer Model on mu and xi (LiteThreads loops=100, 200)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	2.43s	5.17s	7.61s	99.00%	LiteThreads
	2.44s	5.1s	7.55s	99.00%	
	2.49s	5.06s	7.55s	99.00%	
	2.51s	5.09s	7.6s	99.00%	
	2.55s	5.07s	7.63s	99.00%	
mu	1.6s	4.9s	6.51s	99.00%	PosixThreads
	1.51s	4.98s	6.5s	99.00%	
	1.58s	4.93s	6.52s	99.00%	
	1.59s	4.92s	6.52s	99.00%	
	1.64s	4.85s	6.5s	99.00%	
mu	0.34s	0	0.34s	99.00%	QuickThreads
	0.33s	0	0.34s	99.00%	
	0.33s	0	0.33s	99.00%	
	0.33s	0	0.34s	99.00%	
	0.33s	0	0.34s	99.00%	
xi	2.61s	3.65s	6.28s	99.00%	LiteThreads
	2.71s	3.63s	6.36s	99.00%	
	2.65s	3.8s	6.48s	99.00%	
	2.56s	3.68s	6.27s	99.00%	
	2.63s	3.93s	6.59s	99.00%	
xi	1.33s	4.75s	6.1s	99.00%	PosixThreads
	1.27s	4.99s	6.28s	99.00%	
	1.47s	4.96s	6.45s	99.00%	
	1.25s	4.86s	6.13s	99.00%	
	1.36s	4.89s	6.27s	99.00%	
xi	0.55s	0	0.55s	99.00%	QuickThreads
	0.53s	0	0.54s	99.00%	
	0.53s	0	0.54s	99.00%	
	0.53s	0	0.54s	99.00%	
	0.53s	0	0.53s	99.00%	

In the most optimized case (loop iterations=0,0), the LiteThreads library spends no time in spinlock and the smaller user time of LiteThreads leads to better performance than PosixThreads.

3 Performance and Optimizations on Thread Creation and Deletion

Another difference in LiteThreads is that it makes use of clone system call, instead of fork in the thread creation. Compared with fork, clone system call has more options to control the sharing between the parent and child thread, and would be more efficient when new child threads are created. Except that, the mechanism of the two thread libaries in thread creation is similar.

Table 3: Simulation Results of Producer-Consumer Model (LiteThreads loops=0, 0)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
alpha	4.31s	13.38s	17.73s	99.00%	LiteThreads
	4.03s	13.76s	17.83s	99.00%	
	4.24s	13.36s	17.66s	99.00%	
	4.25s	13.2s	17.49s	99.00%	
	4.24s	13.23s	17.51s	99.00%	
epsilon	4.39s	13.13s	17.55s	99.00%	LiteThreads
	4.11s	13.91s	18.05s	99.00%	
	4.26s	13.31s	17.58s	99.00%	
	3.93s	13.06s	17s	99.00%	
	4.17s	12.94s	17.12s	99.00%	
mu	1.13s	5.07s	6.2s	99.00%	LiteThreads
	1.05s	5.19s	6.25s	99.00%	
	1.13s	5.07s	6.21s	99.00%	
	1.13s	5.09s	6.22s	99.00%	
	1.16s	5.06s	6.23s	99.00%	
xi	0.97s	3.82s	4.8s	99.00%	LiteThreads
	1s	3.88s	4.9s	99.00%	
	0.95s	3.64s	4.61s	99.00%	
	0.98s	3.85s	4.85s	99.00%	
	1.02s	3.87s	4.9s	99.00%	

3.1 Stack Space Allocation

Before invoking the system call (clone or fork), the thread library needs to allocate a chunk of stack space for the new thread. In both thread libraries, this process is achieved by the *malloc()* function which is quite complex and time-consuming. By finding a feasible space wheneveer a new thread is created, the *malloc()* limits the performance of both thread libraries. In order to achieve a higher efficiency in LiteThreads, we need to find another way to allocate the stack space.

As each *malloc()* function call needs to switch between the user level and system level, and also spends lots of time finding a big enough chunk of free space, the *malloc()* function consumes much simulation time whenever it is called. One way to reduce the complexity is to allocate a whole chunk of stack space at the beginning of the simulation. As the stack space for each thread is fixed, we can use one *malloc()* function call to allocate the stack space for all the threads created in the program. Later when a new thread is created, the simulator only needs to pick the first available stack space from the whole chunk of memory.

In the LiteThreads library, we utilize this mechanism to optimize thread creation. An integer array *FreeStacks* is used as the data structure to record which stack space is available. The integer variable *FreeStackTop* hold the top index in the *FreeStacks*. To make thread creation faster, the array *FreeStacks* works as a stack and the allocation of a piece of stack space only involves pulling the top item from the array *FreeStacks*. This operation takes constant time and there is no time spent in searching. Our specific implementation is shown as Listing 1 and Listing 2.

Table 4: Simulation Results of Producer-Consumer Model (LiteThreads loops=0, 200)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
alpha	8.4s	13.82s	22.27s	99.00%	LiteThreads
	9.85s	12.58s	22.49s	99.00%	
	10.22s	12.31s	22.57s	99.00%	
	9.54s	12.87s	22.57s	99.00%	
	10.3s	12.27s	22.62s	99.00%	
epsilon	10.76s	13.1s	23.93s	99.00%	LiteThreads
	9.16s	14.19s	23.37s	99.00%	
	9.46s	13.6s	23.1s	99.00%	
	8.88s	14.37s	23.26s	99.00%	
	10.32s	13.51s	23.91s	99.00%	
mu	2.45s	5.08s	7.54s	99.00%	LiteThreads
	2.28s	5.28s	7.56s	99.00%	
	2.47s	5.09s	7.57s	99.00%	
	2.49s	5.02s	7.52s	99.00%	
	2.29s	5.2s	7.5s	99.00%	
xi	2.62s	3.8s	6.44s	99.00%	LiteThreads
	2.51s	3.64s	6.18s	99.00%	
	2.51s	3.64s	6.17s	99.00%	
	2.62s	3.73s	6.38s	99.00%	
	2.49s	3.7s	6.21s	99.00%	

Listing 1: Thread Initialization with Constant-time Stack Space Allocation

```
static int FreeStacks[THREAD_STACK_NUM];
static int FreeStackTop;
static void *Global_StackTop;
static void *Global_StackStart;

void litethread_init(void)
{
  int i;

  if (!(Global_StackStart = malloc((SIM_THREAD_STACK_SIZE
                         + sizeof(litethread_arg)) * THREAD_STACK_NUM)))
  {
    return;
  }
  Global_StackTop = (char*)Global_StackStart + SIM_THREAD_STACK_SIZE
                    * THREAD_STACK_NUM + sizeof(litethread_arg)
                    * (THREAD_STACK_NUM - 1);
  for (i = 0; i < THREAD_STACK_NUM; i++)
    FreeStacks[i] = i;
  FreeStackTop = i - 1;
}
```

Table 5: Simulation Results of Producer-Consumer Model (LiteThreads loops=100, 0)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
alpha	4.16s	12.36s	16.57s	99.00%	LiteThreads
	4.42s	12.1s	16.55s	99.00%	
	4.19s	12.32s	16.56s	99.00%	
	4.33s	12.08s	16.45s	99.00%	
	4.18s	12.3s	16.51s	99.00%	
epsilon	4.55s	12.55s	17.15s	99.00%	LiteThreads
	4.78s	11.98s	16.77s	99.00%	
	4.02s	12.84s	16.87s	99.00%	
	4.07s	12.8s	16.88s	99.00%	
	4.06s	12.87s	16.95s	99.00%	
mu	1.11s	5.15s	6.26s	99.00%	LiteThreads
	1.22s	5.05s	6.28s	99.00%	
	1.16s	5.09s	6.26s	99.00%	
	1.12s	5.07s	6.2s	99.00%	
	1.13s	5.08s	6.22s	99.00%	
xi	1.04s	3.79s	4.85s	99.00%	LiteThreads
	1.04s	3.71s	4.77s	99.00%	
	1.03s	3.9s	4.94s	99.00%	
	0.92s	3.73s	4.67s	99.00%	
	1.02s	3.84s	4.88s	99.00%	

Listing 2: Thread Creation with Constant-time Stack Space Allocation

```
int litethread_create(int (*fn)(void*), void *args)
{
  void            *stacktop;
  void            *stack;
  litethread_arg  *ltarg;
  int             Result;

  assert(fn);

  if (FreeStackTop >= 0)
  { stack = (char*)Global_StackStart + (SIM_THREAD_STACK_SIZE
          + sizeof(litethread_arg)) * FreeStacks[FreeStackTop];
    Result = FreeStacks[FreeStackTop];
    FreeStackTop--;
    }
  else
  {
    errno = ENOMEM;
    return -1;
    }

  stacktop = (char*)stack + SIM_THREAD_STACK_SIZE;
  ltarg    = (litethread_arg*) stacktop;
```

Table 6: Simulation Results of Producer-Consumer Model (LiteThreads loops=1000, 0)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
alpha	4.25s	12.3s	16.6s	99.00%	LiteThreads
	3.75s	12.7s	16.49s	99.00%	
	4.19s	12.25s	16.48s	99.00%	
	4.31s	12.24s	16.59s	99.00%	
	4.25s	12.15s	16.44s	99.00%	
epsilon	4.04s	14.45s	18.51s	99.00%	LiteThreads
	4.08s	13.51s	17.71s	99.00%	
	4.36s	13.08s	17.48s	99.00%	
	4.54s	13.87s	18.42s	99.00%	
	4.01s	14.73s	18.76s	99.00%	
mu	1.1s	5.11s	6.22s	99.00%	LiteThreads
	1.14s	5.06s	6.21s	99.00%	
	1.11s	5.09s	6.2s	99.00%	
	1.07s	5.13s	6.21s	99.00%	
	1.08s	5.13s	6.22s	99.00%	
xi	1.07s	3.78s	4.86s	99.00%	LiteThreads
	1.04s	3.87s	4.93s	99.00%	
	0.99s	3.67s	4.68s	99.00%	
	1.05s	3.8s	4.88s	99.00%	
	0.98s	3.84s	4.84s	99.00%	

```
    ltarg->fn = fn;
    ltarg->arg = args;
    ltarg->PrivateData = NULL;
#ifdef STACK_TOP_CHECK
    ltarg->guard1[0] = ltarg->guard1[1] = ltarg->guard1[2] = 0xDEADBEEF;
    ltarg->guard2[0] = ltarg->guard2[1] = ltarg->guard2[2] = 0xDEADBEEF;
#endif

    ThreadID[Result] = clone( litethread_start_stop, stacktop,
#ifdef CLONE_IO
                              CLONE_IO |
#endif
                              CLONE_FS | CLONE_FILES | CLONE_SIGHAND
                              | CLONE_VM | CLONE_THREAD |
#ifdef CLONE_DETACHED
                              CLONE_DETACHED |
#endif
                              CLONE_CHILD_SETTID | CLONE_CHILD_CLEARTID,
                              ltarg, NULL, NULL, &ctid[Result]);

    if (ThreadID[Result] != -1)
      return Result;
    else
    return -1;
}
```

Table 7: Simulation Results of Producer-Consumer Model (LiteThreads loops=1000, 2000)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
alpha	61.07s	12.88s	74.14s	99.00%	LiteThreads
	61.31s	12.34s	73.81s	99.00%	
	62.21s	11.59s	73.96s	99.00%	
	61.97s	11.82s	73.95s	99.00%	
	62.13s	11.65s	73.94s	99.00%	
epsilon	63.54s	14.11s	77.67s	99.00%	LiteThreads
	63.36s	13.81s	77.19s	99.00%	
	63.85s	13.54s	77.42s	99.00%	
	63.55s	13.64s	77.22s	99.00%	
	63.72s	13.52s	77.27s	99.00%	
mu	14.4s	4.74s	19.15s	99.00%	LiteThreads
	12.3s	6.83s	19.13s	99.00%	
	11.81s	7.27s	19.09s	99.00%	
	13.35s	5.79s	19.14s	99.00%	
	13.62s	5.48s	19.11s	99.00%	
xi	14.94s	4.3s	19.3s	99.00%	LiteThreads
	15.79s	3.58s	19.43s	99.00%	
	14.29s	5.09s	19.44s	99.00%	
	15.75s	3.77s	19.58s	99.00%	
	14.27s	5.03s	19.36s	99.00%	

3.2 Thread Creation and Deletion Benchmark

In order to measure the performance of the optimized LiteThreads library, a benchmark having intensive thread creation/deletion (named TFMUL, Threads with pure Float Multiplication) is utilized. Also, all the tests are running on the four 32-bit Linux machines. The initial LiteThreads, the allocation-optimized LiteThreads, PosixThreads, and QuickThreads are used in the sequential simulator. To avoid the same stack space is reused from time to time in the simulation, a random pattern of stack space allocation is added into the testbench. The details of the testbench are shown in Listing 3.

Listing 3: Intensive Thread Creation/Deletion Benchmark

```
#include <stdio.h>
#include <stdlib.h>
#include <sim.sh>

// number of multiplications per unit
#define MAXLOOP 1000

// number of loops
#ifndef MAXTHREAD
```

```
#define MAXTHREAD 10000
#endif

typedef double float_t;

behavior Fmul
{
  int i = 0;
  float_t f = 1.2;

  void main()
  {
    while(i < MAXLOOP)
    {
      f *= 1.1;
      i ++;
    }
  }
};

behavior Main
{
  Fmul  fmul0, fmul1, fmul2, fmul3, fmul4,
        fmul5, fmul6, fmul7, fmul8, fmul9;

  int main(void) {
    int i;
    char *ptr47, *ptr53, *ptr73, *ptr89;
    printf("Fmul[%d,%d] starting... \n", MAXTHREAD, MAXLOOP);
    for(i = 0; i < MAXTHREAD; i++)
    {
      par { fmul0; }
      ptr47 = (char*)malloc(47);
      par { fmul0; fmul1; fmul2; fmul3; fmul4; fmul5; fmul6; }
      ptr73 = (char*)malloc(73);
      free(ptr47);
      par { fmul0; fmul1; fmul2; fmul3; fmul4; fmul5; fmul6; fmul7; }
      ptr73 = (char*)malloc(73);
      free(ptr89);
      par {fmul0; fmul1; fmul2; fmul3; }
      ptr47 = (char*)malloc(47);
      free(ptr73);
      par { fmul0; fmul1; }
      ptr89 = (char*)malloc(89);
      free(ptr47);
      par { fmul0; fmul1; fmul2; fmul3; fmul4; }
      ptr53 = (char*)malloc(53);
      free(ptr89);
      par { fmul0; fmul1; fmul2; fmul3; fmul4; fmul5; }
      ptr73 = (char*)malloc(73);
      free(ptr53);
      free(ptr73);
```

```
    }
    printf("Done!\n");
    return(0);
    }
};
```

3.3 Experiments and Results

Table 8: Simulation Results of TFMUL on alpha

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
alpha	10.31s	52.51s	67.95s	92.00%	Initial LiteThreads
	10.31s	57.44s	73s	92.00%	
	9.99s	57.34s	72.68s	92.00%	
	10.48s	56.93s	72.54s	92.00%	
	10.17s	57.59s	72.84s	93.00%	
alpha	9.23s	46.09s	60.2s	91.00%	Optimized LiteThreads
	9.17s	46.06s	60.21s	91.00%	
	9.17s	45.25s	59.4s	91.00%	
	9.15s	46.56s	60.81s	91.00%	
	9.03s	45.84s	59.86s	91.00%	
alpha	13.17s	47.31s	65.81s	91.00%	Posixthread
	13.33s	47.35s	65.95s	92.00%	
	13.33s	47.92s	66.62s	91.00%	
	13.25s	47.62s	66.08s	92.00%	
	13.32s	47.46s	65.96s	92.00%	
alpha	1.71s	3.74s	5.51s	99.00%	Quickthread
	1.64s	3.69s	5.39s	99.00%	
	1.71s	3.64s	5.4s	99.00%	
	1.69s	4.01s	5.76s	98.00%	
	1.76s	3.85s	5.66s	99.00%	

The simulation results of TFMUL are shown in Table 8, 9, 10 and 11. For both the intial LiteThreads and the optimized LiteThreads in these tables, they have already implemented the optimization in context switching (loop iterations in mutex_lock and mutex_unlock are 0). Thus, both of them should have smaller user time than PosixThreads, which is indicated in the second column in Table 8, 9, 10 and 11.

On the other hand, as each thread in the benchmark has only pure computation, most of the simulation time is spent on thread creation and deletion (1,000,000 threads in total), which results in much larger system time than user time.

From the tables it is easily seen that, on all the four servers the QuickThreads library still has the smallest simulation time in both user time and system time. For the initial LiteThreads, it has a little larger system time than PosixThreads in Table 8. The same phenomena are shown in the other three tables.

Table 9: Simulation Results of TFMUL on epsilon

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
epsilon	8.36s	50.81s	64.56s	91.00%	Initial LiteThreads
	8.06s	49.13s	62.36s	91.00%	
	7.93s	49.42s	62.53s	91.00%	
	8.17s	49.01s	62.35s	91.00%	
	8.01s	49.27s	62.46s	91.00%	
epsilon	7.65s	47.27s	60.5s	90.00%	Optimized LiteThreads
	7.65s	47.36s	60.71s	90.00%	
	7.16s	44.21s	56.67s	90.00%	
	7.33s	45.41s	58.17s	90.00%	
	7.66s	47.71s	60.66s	91.00%	
epsilon	10.56s	46.4s	62.41s	91.00%	Posixthread
	10.15s	46.07s	61.54s	91.00%	
	10.71s	45.64s	61.64s	91.00%	
	10.36s	46.28s	61.9s	91.00%	
	10.49s	46.66s	62.45s	91.00%	
epsilon	1.57s	4.02s	5.66s	98.00%	Quickthread
	1.47s	3.69s	5.2s	99.00%	
	1.52s	3.69s	5.26s	99.00%	
	1.57s	3.93s	5.6s	98.00%	
	1.44s	3.92s	5.4s	99.00%	

After implementing the allocation optimization in thread creation, the system time of LiteThreads on all servers is reduced obviously. For an instance, the system time of the LiteThreads in Table 11 decreases from about 17 seconds to 14 seconds. However, despite of the obvious improvement in system level overhead, sometimes on epsilon and mu the system time of LiteThreads is still a little larger than PosixThreads.

Combining the optimizations on context switching and thread creation, the LiteThreads library has achieved better performance in elapsed time than the PosixThreads library on all four machines.

4 Performance on 64-bit Architectures

All the previous experiments are running on the 32-bit Linux hosts. On a 64-bit host, the LiteThreads, PosixThreads and QuickThreads have similar performance as before.

Next, we execute the previous two benchmarks on one 64-bit machine. The xi server in our previous experiments has the 64-bit CPU and 64-bit Fedora 12 Linux operating system installed. When enabling these two features, this machine can run the 64-bit benchmarks.

In our new experiment, the two original benchmarks are both running on this machine and the LiteThreads has included both the optimizations of the context switching and the stack space allocation. Besides, all the LiteThreads, PosixThreads and QuickThreads libraries are running in 64-bit mode.

Table 10: Simulation Results of TFMUL on mu

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	3.01s	21.78s	26.99s	91.00%	Initial LiteThreads
	2.93s	21.79s	26.9s	91.00%	
	3.08s	21.83s	27.1s	91.00%	
	3.02s	21.79s	26.99s	91.00%	
	3.12s	21.75s	27.06s	91.00%	
mu	2.51s	19.05s	23.74s	90.00%	Optimized LiteThreads
	2.41s	19.11s	23.71s	90.00%	
	2.53s	19.15s	23.85s	90.00%	
	2.55s	18.95s	23.65s	90.00%	
	2.36s	19s	23.53s	90.00%	
mu	3.82s	17.39s	23.5s	90.00%	Posixthread
	3.74s	17.78s	23.79s	90.00%	
	3.79s	17.56s	23.64s	90.00%	
	3.82s	17.32s	23.42s	90.00%	
	3.72s	17.8s	23.83s	90.00%	
mu	0.58s	1.81s	2.39s	99.00%	Quickthread
	0.58s	1.78s	2.37s	99.00%	
	0.57s	1.72s	2.29s	99.00%	
	0.59s	1.71s	2.31s	99.00%	
	0.57s	1.82s	2.4s	99.00%	

From the simulation results in Table 12 and 13, we can easily find that the simulation performance is similar as before. In both the two benchmarks, the QuickThreads library has the best performance, while the optimized LiteThreads are obviously better than PosixThreads. When using the sequential simulator, QuickThreads accordingly has the smallest user and system time, leading to the much smaller elapsed time in simulation.

In the benchmark of the Producer-Consumer model, the user time of LiteThreads is similar to that of PosixThreads, but the smaller system time makes LiteThreads more efficient. In the benchmark with intensive thread creation/deletion, both of the user time and system time in the LiteThreads library are smaller than those of the PosixThreads library, as a result of the optimization in the mutex lock/unlock and stack space allocation.

Based on these statistics, we can conclude that LiteThreads has better performance than PosixThreads on the 64-bit machine, just as on the 32-bit hosts.

5 Conclusion and Future Work

In this report, we made use of two different benchmarks to evaluate three thread libraries used in the SpecC sequential simulator. From the simulation results, we can draw the conclusion that the QuickThreads library is most efficient in the sequential simulator, while the PosixThreads library is worse than the optimized LiteThreads library. The optimizations on the context switching and thread creation/deletion reduce the user-level and system-level overhead of LiteThreads, respectively. These

Table 11: Simulation Results of TFMUL on xi

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	2.56s	17.19s	21.75s	90.00%	Initial LiteThreads
	2.42s	17.1s	21.54s	90.00%	
	2.48s	16.85s	21.32s	90.00%	
	2.48s	16.9s	21.37s	90.00%	
	2.5s	17.25s	21.77s	90.00%	
xi	2.11s	14.67s	18.69s	89.00%	Optimized LiteThreads
	2.03s	14.71s	18.65s	89.00%	
	2s	14.49s	18.41s	89.00%	
	1.97s	14.44s	18.27s	89.00%	
	1.99s	14.5s	18.4s	89.00%	
xi	3.29s	16.41s	21.82s	90.00%	Posixthread
	3.32s	16.28s	21.68s	90.00%	
	3.17s	16.41s	21.7s	90.00%	
	3.3s	16.55s	21.93s	90.00%	
	3.34s	16.43s	21.9s	90.00%	
xi	0.64s	1.33s	2.02s	97.00%	Quickthread
	0.64s	1.24	1.94s	97.00%	
	0.65s	1.38s	2.09s	97.00%	
	0.6s	1.26s	1.91s	97.00%	
	0.59s	1.23s	1.87s	97.00%	

Table 12: Simulation Results of Producer-Consumer Model on 64-bit Architectures

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	10.46s	27.26s	37.83s	99.00%	LiteThreads
	10.42s	27.48s	38.01s	99.00%	
	9.98s	26.8s	36.89s	99.00%	
	10.61s	27.66s	38.39s	99.00%	
	10.24s	27.45s	37.8s	99.00%	
xi	10.55s	33s	43.68s	99.00%	Posixthread
	10.43s	33.11s	43.67s	99.00%	
	10.51s	32.75s	43.38s	99.00%	
	10.96s	33.65s	44.74s	99.00%	
	10.34s	33.61s	44.07s	99.00%	
xi	4.35s	0	4.36s	99.00%	Quickthread
	4.11s	0	4.12s	99.00%	
	4.31s	0	4.32s	99.00%	
	4.25s	0	4.26s	99.00%	
	4.3s	0	4.32s	99.00%	

conclusions are true on both 32-bit and 64-bit Linux servers.

One open question we noticed in the evaluation is the relationship between the simulation performance and the CPU affinity. In our experiments with intensive context switches, we get some

Table 13: Simulation Results of TFMUL on 64-bit Architectures

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	2.3s	14.67s	18.95s	89.00%	LiteThreads
	2.26s	14.57s	18.78s	89.00%	
	2.29s	14.44s	18.67s	89.00%	
	2.36s	14.52s	18.83s	89.00%	
	2.27s	14.38s	18.59s	89.00%	
xi	3.92s	20.26s	26.37s	91.00%	Posixthread
	4.05s	19.69s	25.9s	91.00%	
	3.85s	20.08s	26.12s	91.00%	
	4s	19.99s	26.19	91.00%	
	3.94s	19.97s	26.07s	91.00%	
xi	0.8s	3.38s	4.2s	99.00%	Quickthread
	0.83s	3.08s	3.93s	99.00%	
	0.82s	3.22s	4.06s	99.00%	
	0.82s	3.2s	4.04s	99.00%	
	0.76s	2.99s	3.76s	99.00%	

interesting simulation results when using the LiteThreads and setting the CPU affinity. Table 14, 15, 16, 17 and 18, 19, 20, 21 list the simulation performance of LiteThreads in two different situations.

Table 14: Simulation Results of Producer-Consumer Model (cores=0 1, loops=0 0)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
epsilon	6.27s	15s	19.93s	106.00%	LiteThreads
	6.38s	14.56s	18.54s	112.00%	
	5.02s	16.18s	19.31s	109.00%	
	7.13s	14.54s	20.16s	107.00%	
	5.84s	16.78s	18.69s	121.00%	
mu	0.58s	5.19s	7.76s	74.00%	LiteThreads
	0.49s	5.2s	7.52s	75.00%	
	0.44s	5.25s	7.29s	78.00%	
	0.55s	5.14s	7.1s	80.00%	
	0.45s	5.1s	7s	79.00%	
xi	2.91s	11.06s	14.54s	96.00%	LiteThreads
	2.59s	11.21s	14.54s	94.00%	
	1.97s	11.84s	14.55s	94.00%	
	1.84s	12.04s	14.52s	95.00%	
	3.54s	10.2s	14.45s	95.00%	

In the first experiment, the Producer-Consumer benchmark is running on servers epsilon, mu and xi, while the Producer thread and the Consumer thread in the program are set on logical core 0 and 1 respectively. The second experiment is only carried out on server xi, with the two threads (Producer thread and Consumer thread) on the same physical core but two different logical cores (logical cores 0&12, 3&15 and 10&22).

Table 15: Simulation Results of Producer-Consumer Model (cores=0 1, loops=100 0)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
epsilon	7.03s	14.43s	19.64s	109.00%	LiteThreads
	6.12s	17.74s	24.53s	97.00%	
	5.79s	15.64s	20.23s	105.00%	
	7.1s	13.7s	18.12s	114.00%	
	6.13s	14.85s	18.2s	115.00%	
mu	0.2s	5.67s	7.22s	81.00%	LiteThreads
	0.48s	5.25s	6.66s	86.00%	
	0.55s	5.06s	6.58s	85.00%	
	0.47s	5.18s	6.77s	83.00%	
	0.52s	5.19s	7.11s	80.00%	
xi	5.09s	8.91s	15.05s	93.00%	LiteThreads
	3.65s	10.28s	15.08s	92.00%	
	5.04s	8.73s	14.62s	94.00%	
	3.06s	10.79s	14.58s	95.00%	
	5.54s	8.24s	14.43s	95.00%	

Table 16: Simulation Results of Producer-Consumer Model (cores=0 1, loops=0 200)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
epsilon	10.46s	14.63s	24.22s	103.00%	LiteThreads
	10.49s	13.9s	20.53s	118.00%	
	10.54s	13.86s	19.43s	125.00%	
	10.57s	13.85s	19.24s	126.00%	
	10.29s	14.21s	19.34s	126.00%	
mu	2.06s	4.76s	7.02s	97.00%	LiteThreads
	1.63s	4.93s	6.68s	98.00%	
	1.8s	4.72s	6.36s	102.00%	
	1.74s	4.86s	6.65s	99.00%	
	1.96s	4.74s	7.12s	94.00%	
xi	6.37s	10.51s	14.67s	115.00%	LiteThreads
	6.6s	10.3s	14.79s	114.00%	
	6.46s	10.44s	14.66s	115.00%	
	6.46s	10.3s	14.76s	113.00%	
	7.07s	10.14s	14.85s	115.00%	

From Figure 1, 2, 3 and 4, we can find that the communication overhead grows in these two experiments. However, in both experiments, we could still get the same patterns of the user time and system time as those in the case where all the threads in the program are forced to run on one logical core. The system time in different cases stays similar, while the user time decreases monotonically with the loop iteration in the mutex unlock, and is not affected by the variation of the loop iteration in the mutex lock.

However, in the first experiment (Table 14, 15, 16, 17), we can notice that the user time, system

Table 17: Simulation Results of Producer-Consumer Model (cores=0 1, loops=100 200)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
epsilon	10.57s	14.44s	20.82s	120.00%	LiteThreads
	10.7s	15.05s	24.15s	106.00%	
	10.38s	14.95s	21.4s	118.00%	
	11.08s	15.58s	25.42s	104.00%	
	10.14s	15.27s	20.24s	125.00%	
mu	1.54s	5.04s	6.43s	102.00%	LiteThreads
	1.4s	5.23s	6.72s	98.00%	
	1.84s	4.85s	7.06s	94.00%	
	1.78s	4.86s	6.66s	99.00%	
	1.79s	5.13s	7.19s	96.00%	
xi	4.24s	12.38s	15.08s	110.00%	LiteThreads
	4.98s	11.79s	15.09s	111.00%	
	6.01s	10.86s	15.22s	110.00%	
	4.89s	11.88s	15.03s	111.00%	
	6.13s	10.52s	14.64s	113.00%	

Table 18: Simulation Results of Producer-Consumer Model on xi(same physical core, different logical cores, loops=0 0)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Core
xi	3.34s	6.53s	10.65s	92.00%	(0 12)
	3.2s	6.84s	10.88s	92.00%	
	3.54s	5.29s	9.59s	92.00%	
	3.45s	6.12s	10.35s	92.00%	
	2.71s	5.66s	8.66s	96.00%	
xi	3.07s	5.44s	8.46s	100.00%	(3 15)
	3.31s	5.51s	8.79s	100.00%	
	3.49s	5.81s	9.65s	96.00%	
	3.12s	5.48s	8.92s	96.00%	
	3.12s	5.77s	9.38s	94.00%	
xi	2.91s	5.53s	8.3s	101.00%	(10 22)
	3.05s	5.76s	8.77s	100.00%	
	3.15s	5.68s	8.68s	101.00%	
	3.25s	5.93s	9.14s	100.00%	
	3.31s	6.29s	9.81s	97.00%	

time and elapsed time are not consistent in each case, which leads to the fact that the elapsed time on servers mu and xi remains identical no matter how we change the loop iterations in the LiteThreads library.

In the second experiment, the elapsed time of LiteThreads is even smaller when we increase the spin time in mutex_unlock, contradictory to our conclusion in the previous parts. Also sometimes, the CPU load surpasses 100%, even though we only use the sequential simulator in the tests.

Table 19: Simulation Results of Producer-Consumer Model on xi(same physical core, different logical cores, loops=100 0)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Core
	3.38s	6.4s	10.53s	92.00%	
	3.3s	5.73s	9.81s	92.00%	
xi	3s	6.53s	10.21s	93.00%	(0 12)
	3.36s	6.23s	10.38s	92.00%	
	3.68s	5.3s	9.68s	92.00%	
	3.29s	5.81s	9.64s	94.00%	
	3.49s	5.15s	9.17s	94.00%	
xi	3.59s	5.08s	9.15s	94.00%	(3 15)
	3.27s	5.63s	9.44s	94.00%	
	3.27s	5.87s	9.74s	93.00%	
	2.87s	5.46s	8.35s	99.00%	
	2.95s	5.82s	8.69s	100.00%	
xi	2.6s	4.37s	6.78s	102.00%	(10 22)
	3.29s	6.25s	9.51s	100.00%	
	3.14s	6.03s	9.13s	100.00%	

Table 20: Simulation Results of Producer-Consumer Model on xi(same physical core, different logical cores, loops=0 200)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Core
	4.29s	4s	7.58s	109.00%	
	5.13s	3.7s	7.86s	112.00%	
xi	4.22s	4.22s	7.61s	110.00%	(0 12)
	4.17s	4.39s	7.79s	109.00%	
	4.39s	4.1s	7.64s	111.00%	
	4.78s	3.56s	7.07s	118.00%	
	4.15s	3.93s	6.76s	119.00%	
xi	4.33s	4.09s	6.99s	120.00%	(3 15)
	4.57s	3.66s	7.01s	117.00%	
	4.54s	4.29s	7.67s	115.00%	
	4.83s	3.63s	6.95s	121.00%	
	4.05s	4.15s	6.76s	121.00%	
xi	4.31s	4.25s	6.98s	122.00%	(10 22)
	3.8s	4.39s	6.78s	120.00%	
	4.24s	4.22s	6.97s	121.00%	

These phenonema are explicit in the two experiments, and extremely obvious on servers epsilon and xi. As both epsilon and xi have the feature of hyperthreading, while mu does not, we believe these phenomena are related behind the hyperthreading feature.

Based on what we have found, we plan to design more specific experiments to find the reasons of these open questions in the future.

Table 21: Simulation Results of Producer-Consumer Model on xi(same physical core, different logical cores, loops=100 200)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Core
xi	4.38s	4.01s	7.63s	110.00%	(0 12)
	4.46s	4.13s	7.77s	110.00%	
	4.7s	3.91s	7.72s	111.00%	
	4.46s	4.14s	7.66s	112.00%	
	4.63s	3.98s	7.77s	110.00%	
xi	4.79s	3.5s	7.56s	109.00%	(3 15)
	4.29s	4.18s	7.27s	116.00%	
	4.95s	3.53s	7.55s	112.00%	
	4.94s	3.47s	7.47s	112.00%	
	4.95s	3.59s	7.52s	113.00%	
xi	3.84s	4.21s	6.64s	121.00%	(10 22)
	4.32s	4.11s	6.96s	121.00%	
	3.95s	4.09s	6.64s	121.00%	
	3.68s	3.87s	5.95s	126.00%	
	3.79s	3.93s	6.08s	126.00%	

Acknowledgment

The authors would like to thanks Jeff White and Tony Mathew for their initial exploration on LiteThreads as an alternative to PosixThreads.

This work has been supported in part by funding from the National Science Foundation (NSF) under research grant NSF Award #0747523. The authors thank the NSF for the 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 National Science Foundation.

References

[1] Tony Mathew and Rainer Dömer. A custom thread library built on native linux threads for faster embedded system simulation. Technical Report CECS-TR-11-10, Center for Embedded Computer Systems, University of California, Irvine, December 2011.