A User-level Thread Library Built on Linux
Context Functions for Efficient ESL Simulation

Guantao Liu and Rainer Dömer

Technical Report CECS-13-07
June 6, 2013

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/
A User-level Thread Library Built on Linux Context Functions for Efficient ESL Simulation

Guantao Liu and Rainer Dömer

Technical Report CECS-13-07
June 6, 2013

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

Currently QuickThreads library is widely used in the multi-threaded programs such as Electronic System Level (ESL) simulation. As a user-level thread library, QuickThreads is very efficient in thread manipulation as it operates solely at user level and introduces no operating system overhead. While QuickThreads library utilizes a portable interface to wrap machine-dependent code that performs thread initialization and context switching, it only works on a certain number of specific platforms and architectures. In this report, we propose a new user-level thread library that offers the same features as QuickThreads, but makes use of Linux library functions and therefore is portable to all 32-bit Linux platforms.
List of Figures

1 Intel Core 2 Quad architecture, Q9650 (mu) [3] ................................. 3
2 Intel Xeon architecture, X5650 (xi) [3] ........................................ 3
3 Simulation Results for Producer-Consumer Model on mu ...................... 4
4 Simulation Results for Producer-Consumer Model on xi ...................... 4
5 Simulation Results for TFMUL Model on mu .................................. 5
6 Simulation Results for TFMUL Model on xi .................................. 5
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Simulation Results for Producer-Consumer Model</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Simulation Results for TFMUL Model</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Producer-Consumer Model on ( \mu )</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>Producer-Consumer Model on ( \xi )</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>TFMUL Model on ( \mu )</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>TFMUL Model on ( \xi )</td>
<td>17</td>
</tr>
</tbody>
</table>
List of Listings

1  Producer-Consumer Model .................................................. 8
2  TFMUL Model ................................................................. 13
Abstract

Currently QuickThreads library is widely used in the multi-threaded programs such as Electronic System Level (ESL) simulation. As a user-level thread library, QuickThreads is very efficient in thread manipulation as it operates solely at user level and introduces no operating system overhead. While QuickThreads library utilizes a portable interface to wrap machine-dependent code that performs thread initialization and context switching, it only works on a certain number of specific platforms and architectures. In this report, we propose a new user-level thread library that offers the same features as QuickThreads, but makes use of Linux library functions and therefore is portable to all 32-bit Linux platforms.

1 Introduction

Before the popular use of multiprocessor machines, user-level thread library is widely adopted in multithreading programs to handle multiple concurrent tasks in a time-slice manner. On the single processor, multiple tasks are executed in a time-division multiplexing mode and the context switching between these tasks generally happens so frequently that the users perceive the threads or tasks as running at the same time. Thus, the essential part of the user-level thread library is how to efficiently manage context switching between different user threads in the same process. A well-known example of the user-level threads is QuickThreads [2], which owns great performance in sequential multithreading programs. However, QuickThreads only works right on a limited number of platforms, which restricts the usage of QuickThreads. In the remainder of this technical report, a new
user-level thread library built on Linux context library functions is proposed for the SpecC sequential simulator. Preliminary simulation results indicate that the new thread library has very similar performance to QuickThreads and it is portable to all 32-bit Linux platforms.

2 Basic Ideas of the ContextThreads Library

As a popular user-level thread library, QuickThreads provides a somewhat portable interface to machine-dependent code that performs thread initialization and context switching [2]. To offer more simplicity and flexibility to the thread package, QuickThreads separate the notion (starting and stopping threads) from the thread allocation and scheduling of different queues. In fact, QuickThreads does not manipulate any allocation and run queues. Instead, it only provide a simple mechanism that performs a context switch, and then invokes a client function on behalf of the halted thread. During such a context switching, QuickThreads library will first save the register values of the old thread on to its stack, adjusting the stack pointer as needed, and then jump to the functions of the new thread by loading its stack.

Although QuickThreads library is superior in the thread initialization and context switching, it is only portable to a certain number of platforms and architectures (80386, DEC, VAX family and so on). On other platforms, modern Linux operating systems offer some library functions to achieve the same functionalities. Some examples of these functions are `getcontext`, `setcontext`, `swapcontext` and `makecontext` declared in `ucontext.h` [1]. Specifically, `getcontext` would save the current execution context (register values, program counter and etc.) to a data structure typed `ucontext_t`, `setcontext` would load a `ucontext_t` struct and switch to the specified execution context, `swapcontext` will save the current context and switch to another one and `makecontext` will create a new execution context by defining the thread functions and arguments. To create a new thread, we can first call `getcontext` to retrieve the current context and modify the context by specifying the function and arguments in `makecontext`. To context switch to a new thread, we could just use the `swapcontext` function to stop the current thread and continue executing another. Integrating these functions, a new user-level ContextThreads library is created. Similar to what QuickThreads do, ContextThreads separate the thread execution and scheduling. Changing scheduling policies in the ContextThreads library would be as easy as changing a function pointer in `makecontext`. By utilizing the same ideas in QuickThreads library and the Linux context functions, the new user-level thread library offers high performance in computation as well as a wide portability to all Linux platforms.

3 Performance Evaluation of the ContextThreads Library

3.1 Platform Architectures and Benchmark Examples

To evaluate the performance of ContextThreads, we utilize two SpecC benchmarks to test two different aspects of a thread library: a Producer-Consumer example (Prod-Cons) to evaluate the context switching performance and a parallel benchmark which has intensive thread creation/deletion operations to test the feature of thread initialization (named TFMUL, Threads with pure Floating-point MULtiplication). Both of the benchmarks are running on two 32-bit Linux machines, which have
Intel(R) Core(TM) 2 Quad architecture Q9650 3.0 GHz CPU (named \textit{mu}) and Intel(R) Xeon(R) architecture X5650 2.66 GHz CPU (named \textit{xi}) respectively. Figure 1 and 2 illustrate the architectures of the two processors. The dashed line in the middle of the processor means that the CPU has the hyperthreading feature enabled [3].

![Figure 1: Intel Core 2 Quad architecture, Q9650 (mu) [3]](image1)

![Figure 2: Intel Xeon architecture, X5650 (xi) [3]](image2)

To simulate these two examples, we adopt three SpecC sequential simulators which are based on QuickThreads, ContextThreads and PosixThreads. The simulation times for the two benchmarks on \textit{mu} and \textit{xi} are shown in Figure 3 to 6. Table 1 and 2 list the data used in these figures.

\footnote{The simulation results in all these figures are picked up from the tables in Appendix B, and they always choose the example which has medium elapsed time.}
### Table 1: Simulation Results for Producer-Consumer Model

<table>
<thead>
<tr>
<th>Hostname</th>
<th>User Time (s)</th>
<th>System Time (s)</th>
<th>Elapsed Time (s)</th>
<th>CPU Load</th>
<th>Thread Library</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mu</strong></td>
<td>26.85</td>
<td>0</td>
<td>26.87</td>
<td>99.00%</td>
<td>QuickThreads</td>
</tr>
<tr>
<td></td>
<td>34.11</td>
<td>14.22</td>
<td>48.35</td>
<td>99.00%</td>
<td>ContextThreads</td>
</tr>
<tr>
<td></td>
<td>84.8</td>
<td>189.48</td>
<td>274.38</td>
<td>99.00%</td>
<td>PosixThreads</td>
</tr>
<tr>
<td><strong>xi</strong></td>
<td>22.08</td>
<td>0</td>
<td>22.14</td>
<td>99.00%</td>
<td>QuickThreads</td>
</tr>
<tr>
<td></td>
<td>28.75</td>
<td>9.79</td>
<td>38.65</td>
<td>99.00%</td>
<td>ContextThreads</td>
</tr>
<tr>
<td></td>
<td>63.6</td>
<td>231.25</td>
<td>295.66</td>
<td>99.00%</td>
<td>PosixThreads</td>
</tr>
</tbody>
</table>

![Figure 3: Simulation Results for Producer-Consumer Model on mu](image1)

![Figure 4: Simulation Results for Producer-Consumer Model on xi](image2)
Table 2: Simulation Results for TFMUL Model

<table>
<thead>
<tr>
<th>Hostname</th>
<th>Usr Time</th>
<th>Sys Time</th>
<th>Elapsed Time</th>
<th>CPU Load</th>
<th>Thread Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>mu</td>
<td>8.61s</td>
<td>17.67s</td>
<td>26.29s</td>
<td>99.00%</td>
<td>QuickThreads</td>
</tr>
<tr>
<td></td>
<td>11.1s</td>
<td>24.36s</td>
<td>35.48s</td>
<td>99.00%</td>
<td>ContextThreads</td>
</tr>
<tr>
<td></td>
<td>38.22s</td>
<td>169.36s</td>
<td>230.67s</td>
<td>89.00%</td>
<td>PosixThreads</td>
</tr>
<tr>
<td>xi</td>
<td>7.38s</td>
<td>11.21s</td>
<td>18.69s</td>
<td>99.00%</td>
<td>QuickThreads</td>
</tr>
<tr>
<td></td>
<td>10.1s</td>
<td>16.72s</td>
<td>26.93s</td>
<td>99.00%</td>
<td>ContextThreads</td>
</tr>
<tr>
<td></td>
<td>37.84s</td>
<td>163.84s</td>
<td>222.02s</td>
<td>90.00%</td>
<td>PosixThreads</td>
</tr>
</tbody>
</table>

Figure 5: Simulation Results for TFMUL Model on mu

Figure 6: Simulation Results for TFMUL Model on xi

3.2 Producer-Consumer Model

The first parallel benchmark, Producer-Consumer model, is a simple example which features intensive context switching. During the whole simulation, the program will create three threads: a
Producer, a Consumer and a Monitor. The Producer instance will repeatedly send data to the Consumer through a double-handshake channel. This communication is wrapped up in a huge loop and the monitor will terminate the whole program when all the communication is done. Hence, this example has a limited amount of parallelism but a heavyweight of thread synchronization. The exact code of the Producer-Consumer model is listed in Appendix A.1.

From Figure 3 and 4, it is easily seen that QuickThreads library has the best performance on both two platforms while the sequential PosixThreads simulator owns the worst performance. ContextThreads library is slightly inferior to QuickThreads and is much better than PosixThreads. The almost zero system time in the QuickThreads simulator indicates that QuickThreads have very low kernel-level overhead and is quite efficient in context switching. ContextThreads library has a small amount of system-level time as the signal mask is saved and restored using the system call sigprocmask, introducing some kernel-level overhead to the ContextThreads library. Even so, ContextThreads is still more than 5 times faster than PosixThreads for the Producer-Consumer example. When the benchmark has intensive context switching, PosixThreads spend lots of time in the kernel-level scheduling and synchronization of different threads. From Figure 3 and 4, we can see that the system-level overhead of PosixThreads is more than 10 times larger than that of QuickThreads and ContextThreads. Generally speaking, for the Producer-Consumer Model, the sequential QuickThreads simulator has a speedup of 10 over the sequential PosixThreads simulator and the ContextThreads simulator has a speedup of more than 6 over the PosixThreads simulator.

3.3 Threads with Pure Floating-point Multiplication (TFMUL)

TFMUL model is a highly parallel example that stresses thread creation/deletion. In each thread of TFMUL, it is doing floating-point multiplication and there is no inter-thread communication. Thus, all the child threads in the benchmark can be executing at the same time without any data dependencies. A total of 10,000,000 threads are created in the whole process and it brings a heavy load in thread initialization. A.2 shows the source code of the TFMUL example.

For the TFMUL benchmark, we can draw the same conclusion as the Producer-Consumer example. The heavy load of kernel threads creation and kernel structs manipulation will burden the performance of PosixThreads library. On both mu and xi, the PosixThreads library is more than 6 times slower than the other two user-level thread libraries. For QuickThreads and ContextThreads, they both have a high efficiency in thread initialization (as indicated by the small system time on Figure 5 and 6) and QuickThreads library still owns the best performance.

4 Conclusion

According to the simulation results for the two benchmarks, we can conclude that ContextThreads library is slightly inferior to QuickThreads as a user-level thread library, but is portable to all 32-bit Linux platforms as it does not depend on any machine-dependent code (ContextThreads fail on LP64-architectures as the function makecontext requires additional parameters to be type int, but the function call passes pointers. On LP64 architectures, the size of pointer is larger than that of an integer). Therefore, ContextThreads library would be a good substitute of the QuickThreads library...
on platforms where QuickThreads are not configurable.

Acknowledgment

The authors thank Professor Demsky, EECS Department, UC Irvine for the initial idea of replacing QuickThreads with ContextThreads.

References


A Benchmark Examples

A.1 Producer-Consumer Model

Listing 1: Producer-Consumer Model

1 // prodcons.sc: simple producer–consumer example
2 // author: Rainer Doemer, Guantao Liu
3 // 02/14/13 GL modified to test HybridThreads library
4 // 01/09/12 RD modified to test ooo-simulator run-ahead
5
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <assert.h>
9 #include <csim.sh>
10 #include <sched.h>
11
12 #include <c_typed_double_handshake.sh>
13
14 #define DATA 42
15
16 #define ITERATIONS 5000000
17
18 #define EXIT_ON_HANDSHAKE // to exit on hand–shake
19
20 #define printf nop // eliminate printing messages
21
22 #ifdef USE_FLOATING_POINT
23 #define FDATA 42.5e0
24 #endif
25
26 #ifdef MAXTHREAD
27 #define MAXTHREAD 1
28 #endif
29
30 DEFINE_I_TYPED_SENDER(char, char) // interface i_char_sender
31 DEFINE_I_TYPED_RECEIVER(char, char) // interface i_char_receiver
32 DEFINE_I_TYPED_TRANSCIVER(char, char) // interface i_char_tranciever
33 DEFINE_C_TYPED_DOUBLE_HANDSHAKE(char, char) // channel c_char_double_handshake
34
35 DEFINE_I_TYPED_SENDER(short, short) // interface i_short_sender
36 DEFINE_I_TYPED_RECEIVER(short, short) // interface i_short_receiver
37 DEFINE_I_TYPED_TRANSCIVER(short, short) // interface i_short_tranciever
38 DEFINE_C_TYPED_DOUBLE_HANDSHAKE(short, short) // channel c_short_double_handshake
39
40 DEFINE_I_TYPED_SENDER(int, int) // interface i_int_sender
41 DEFINE_I_TYPED_RECEIVER(int, int) // interface i_int_receiver
42 DEFINE_I_TYPED_TRANSCIVER(int, int) // interface i_int_tranciever
43 DEFINE_C_TYPED_DOUBLE_HANDSHAKE(int, int) // channel c_int_double_handshake
44
45 DEFINE_I_TYPED_SENDER(llong, llong) // interface i_llong_sender
46 DEFINE_I_TYPED_RECEIVER(llong, llong) // interface i_llong_receiver
47 DEFINE_I_TYPED_TRANSCIVER(llong, llong) // interface i_llong_tranciever
48 DEFINE_C_TYPED_DOUBLE_HANDSHAKE(llong, llong) // channel c_llong_double_handshake
49
50 #ifdef USE_FLOATING_POINT
51 DEFINE_I_TYPED_SENDER(float, float) // interface i_float_sender
52 DEFINE_I_TYPED_RECEIVER(float, float) // interface i_float_receiver
53 DEFINE_I_TYPED_TRANSCIVER(float, float) // interface i_float_tranciever
54
DEFINE_C_TYPED_DOUBLE_HANDSHAKE(float, float) // channel c_float_double_handshake

DEFINE_I_TYPED_SENDER(double, double)    // interface i_double_sender
DEFINE_I_TYPED_RECEIVER(double, double)  // interface i_double_receiver
DEFINE_I_TYPED_TRANSCIEVER(double, double) // interface i_double_transceiver
DEFINE_C_TYPED_DOUBLE_HANDSHAKE(double, double) // channel c_double_double_handshake

DEFINE_I_TYPED_SENDER(long double, long double) // interface i_long_double_sender
DEFINE_I_TYPED_RECEIVER(long double, long double) // interface i_long_double_receiver
DEFINE_I_TYPED_TRANSCIEVER(long double, long double) // interface i_long_double_transceiver
DEFINE_C_TYPED_DOUBLE_HANDSHAKE(long double, long double) // channel c_long_double_double_handshake

#endif

void nop(const char*, ...) {
/* do nothing */
}

import "c_handshake";

behavior producer(
#ifdef USE_FLOATING_POINT
  i_float_sender pF,
  i_double_sender pD,
  i_long_double_sender pl,
#endif
  i_char_sender pc,
  i_short_sender ps,
  i_int_sender pi,
  i_long_sender pl)
{
  void main(void)
  {
    char c = DATA;
    short s = DATA;
    int i = DATA;
    long long l = DATA;
#ifdef USE_FLOATING_POINT
    float F = FDATA;
    double D = FDATA;
    long double L = FDATA;
#endif
    int n;

    print_time();
    if (((char*)s)[0] == DATA) {
      printf("Producer: appears to be LITTLE endian\n");
    }
    else if (((char*)s)[1] == DATA) {
      printf("Producer: appears to be BIG endian\n");
    }
    else {
      printf("Producer: appears to be UNKNOWN endian\n");
    }
    for(n=0; n<ITERATIONS; n++)
    {
      waitfor(10);
      print_time();
      printf("Producer: sending char c = %d (0x%02x)\n", c, c);
    }
PCs send (c);
c++;
waitfor (10);
print_time();
printf("Producer: sending short s = %d (0x%04x)\n", s, s);
ps.send(s);
s++;
waitfor (10);
print_time();
printf("Producer: sending int i = %d (0x%08x)\n", i, i);
pi.send(i);
i++;
waitfor (10);
print_time();
printf("Producer: sending llong l = %lld (0x%016llx)\n", l, 1);
pl.send(l);
l++;  

#define USE_FLOATING_POINT
waitfor (10);
print_time();
printf("Producer: sending float F = %g\n", F);
pF.send(F);
F += .5;
waitfor (10);
print_time();
printf("Producer: sending double D = %g\n", D);
pD.send(D);
D += .5;
waitfor (10);
print_time();
printf("Producer: sending ldouble L = %Lg\n", L);
pl.send(L);
L += .5;
#endif

behavior consumer(
#ifdef USE_FLOATING_POINT
i_float_receiver pF,
i_double_receiver pD,
i_ldouble_receiver pL,
#endif
i_char_receiver pc,
i_short_receiver ps,
i_int_receiver pi,
i_llong_receiver pl
#ifdef EXIT_ON_HANDSHAKE
i_send pdone
#endif
)

void main(void)
{
    char c;
    short s = DATA;
    int i;
long long l;

#ifdef USE_FLOATING_POINT
    float F = FDATA;
    double D = FDATA;
    long double L = FDATA;
#endif

int n;

if (((char*) & s)[0] == DATA) {
    printf("Consumer: appears to be LITTLE endian\n");
} else if (((char*) & s)[1] == DATA) {
    printf("Consumer: appears to be BIG endian\n");
} else {
    printf("Consumer: appears to be UNKNOWN endian\n");
}

s = 0;
for (n=0; n<ITERATIONS; n++) {
    pc.receive(& c);
    printf("Consumer: received char c = %d (0x%02x)\n", c, c);
    ps.receive(& s);
    printf("Consumer: received short s = %d (0x%04x)\n", s, s);
    pi.receive(& i);
    printf("Consumer: received int i = %d (0x%08x)\n", i, i);
    pl.receive(& l);
    printf("Consumer: received long l = %lld (0x%016llx)\n", l, l);
    
    if define USE_FLOATING_POINT
        pF.receive(& F);
        printf("Consumer: received float F = %g\n", F);
        pD.receive(& D);
        printf("Consumer: received double D = %g\n", D);
        pl.receive(& L);
        printf("Consumer: received long double L = %Lg\n", L);
    endif
#endif

    printf("Consumer: done.\n");
#endif
#endif
#endif

behavior monitor(
#ifdef EXIT_ON_HANDSHAKE
    pdone.send();
#endif
#endif
)

void main(void)
#ifdef EXIT_ON_TIME
    waitfor(EXIT_ON_TIME);
#endif

#ifdef EXIT_ON_HANDSHAKE
    pdone.receive();
#endif

    print_time();
    printf("Monitor: Done, exiting...\n");
    exit(0);
}

behavior DUT
{
    c_char_double_handshake cc;
    c_short_double_handshake cs;
    c_int_double_handshake ci;
    c_long_double_handshake cl;
#ifdef USE_FLOATING_POINT
    c_float_double_handshake cF;
    c_double_double_handshake cD;
    c_ldouble_double_handshake cL;
#endif
#ifdef EXIT_ON_HANDSHAKE
    c_handshake cend;
#endif
    producer prod(cc, cs, ci, cl);
#ifdef USE_FLOATING_POINT
    cF, cD, cL,
#endif
    consumer cons(cc, cs, ci, cl);
#ifdef USE_FLOATING_POINT
    cF, cD, cL,
#endif
    monitor mon(cend);
#ifdef EXIT_ON_HANDSHAKE
    cend
#endif
}

behavior Main
{
    void main(void)
    {
        int i;
        for (i = 0; i < MAXTHREAD; i++)
            par { prod.main();
                cons.main();
                mon.main();
            }
    }
}

behavior Main
{
A.2 TFMUL Model

Listing 2: TFMUL Model

```c
// TFMUL.sc: parallel floating-point benchmark
// author: Weiwei Chen, Rainer Doemer, Guantao Liu
// 02/15/13 GL modified to test HybridThreads library
// 11/13/11 RD modified to create more parallel threads
// 09/02/11 WC created to test parallel simulators

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

#define MAXLOOP 1000

// number of multiplications per unit
// number of threads
#endif
#define MAXTHREAD 100000
#endif

// type of floating-point numbers
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,
```


```c
    fmul5, fmul6, fmul7, fmul8, fmul9;

int main(void)
{
    int i;
    char *ptr47, *ptr53, *ptr73, *ptr89;
    printf("Fmul%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; }
        ptr53 = (char*)malloc(53);
        free(ptr73);
        par { fmul0; fmul1; fmul2; }
        ptr73 = (char*)malloc(73);
        free(ptr53);
        par { fmul0; fmul1; fmul2; fmul3; fmul4; }
        ptr47 = (char*)malloc(47);
        free(ptr73);
        par { fmul0; fmul1; }
        ptr89 = (char*)malloc(89);
        free(ptr47);
        par { fmul0; fmul1; fmul2; fmul3; fmul4; fmul5; fmul6; fmul7; }
        ptr73 = (char*)malloc(73);
        free(ptr89);
        par { fmul0; fmul1; fmul2; fmul3; fmul4; fmul5; fmul6; fmul7; fmul8; }
        ptr53 = (char*)malloc(53);
        free(ptr73);
        par { fmul0; }
        ptr47 = (char*)malloc(47);
        free(ptr47);
        par { fmul0; fmul1; fmul2; fmul3; fmul4; fmul5; fmul6; }
        ptr73 = (char*)malloc(73);
        free(ptr47);
        par { fmul0; fmul1; fmul2; fmul3; fmul4; fmul5; fmul6; fmul7; fmul8; }
        ptr53 = (char*)malloc(53);
        free(ptr73);
        par { fmul0; fmul1; fmul2; }
        ptr89 = (char*)malloc(89);
        free(ptr47);
        par { fmul0; fmul1; fmul2; fmul3; }
        ptr73 = (char*)malloc(73);
        free(ptr53);
        par { fmul0; fmul1; }
        ptr89 = (char*)malloc(89);
        free(ptr47);
        par { fmul0; fmul1; fmul2; fmul3; fmul4; }
    }
```
B Measured Simulation Times for All Benchmarks and Applications

B.1 Simulation Time for Producer-Consumer Model

Table 3: Producer-Consumer Model on mu

<table>
<thead>
<tr>
<th>Hostname</th>
<th>Usr Time</th>
<th>Sys Time</th>
<th>Elapsed Time</th>
<th>CPU Load</th>
<th>Thread Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>mu</td>
<td>26.78s</td>
<td>0</td>
<td>26.79s</td>
<td>99.00%</td>
<td>QuickThreads</td>
</tr>
<tr>
<td></td>
<td>26.97s</td>
<td>0</td>
<td>26.99s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>26.91s</td>
<td>0</td>
<td>26.92s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>26.85s</td>
<td>0</td>
<td>26.87s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>26.83s</td>
<td>0</td>
<td>26.84s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>34.27s</td>
<td>14.04s</td>
<td>48.34s</td>
<td>99.00%</td>
<td>ContextThreads</td>
</tr>
<tr>
<td></td>
<td>34.24s</td>
<td>14.06s</td>
<td>48.32s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>34.54s</td>
<td>14.23s</td>
<td>48.79s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>34.41s</td>
<td>14.1s</td>
<td>48.53s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>34.11s</td>
<td>14.22s</td>
<td>48.35s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td>mu</td>
<td>84.8s</td>
<td>191.57s</td>
<td>276.46s</td>
<td>99.00%</td>
<td>PosixThreads</td>
</tr>
<tr>
<td></td>
<td>84.49s</td>
<td>189.62s</td>
<td>274.21s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>84.8s</td>
<td>189.48s</td>
<td>274.36s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>84.22s</td>
<td>188.86s</td>
<td>273.18s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>84.16s</td>
<td>191.44s</td>
<td>275.69s</td>
<td>99.00%</td>
<td></td>
</tr>
</tbody>
</table>
### Table 4: Producer-Consumer Model on xi

<table>
<thead>
<tr>
<th>Hostname</th>
<th>Usr Time</th>
<th>Sys Time</th>
<th>Elapsed Time</th>
<th>CPU Load</th>
<th>Thread Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>xi</td>
<td>22.11s</td>
<td>0</td>
<td>22.17s</td>
<td>99.00%</td>
<td>QuickThreads</td>
</tr>
<tr>
<td></td>
<td>22.08s</td>
<td>0</td>
<td>22.15s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.08s</td>
<td>0</td>
<td>22.14s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.04s</td>
<td>0</td>
<td>22.11s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>21.8s</td>
<td>0</td>
<td>21.86s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td>xi</td>
<td>28.44s</td>
<td>10.04s</td>
<td>38.6s</td>
<td>99.00%</td>
<td>ContextThreads</td>
</tr>
<tr>
<td></td>
<td>28.57s</td>
<td>10.05s</td>
<td>38.74s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28.82</td>
<td>10.28s</td>
<td>39.22s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28.75s</td>
<td>9.79s</td>
<td>38.65s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28.1s</td>
<td>10.16s</td>
<td>38.37s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td>xi</td>
<td>63.86s</td>
<td>233.22s</td>
<td>297.9s</td>
<td>99.00%</td>
<td>PosixThreads</td>
</tr>
<tr>
<td></td>
<td>63.6s</td>
<td>231.25s</td>
<td>295.66s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>65.28s</td>
<td>228.73s</td>
<td>294.82s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>63.05s</td>
<td>229.41s</td>
<td>293.27s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64.88s</td>
<td>234.53s</td>
<td>300.24s</td>
<td>99.00%</td>
<td></td>
</tr>
</tbody>
</table>
### B.2 Simulation Time for TFMUL Model

Table 5: TFMUL Model on mu

<table>
<thead>
<tr>
<th>Hostname</th>
<th>Usr Time</th>
<th>Sys Time</th>
<th>Elapsed Time</th>
<th>CPU Load</th>
<th>Thread Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>mu</td>
<td>8.42s</td>
<td>17.97s</td>
<td>26.41s</td>
<td>99.00%</td>
<td>QuickThreads</td>
</tr>
<tr>
<td></td>
<td>8.61s</td>
<td>17.67s</td>
<td>26.29s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8.28s</td>
<td>17.56s</td>
<td>25.86s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8.41s</td>
<td>17.73s</td>
<td>26.15s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8.61s</td>
<td>18.31s</td>
<td>26.94s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td>mu</td>
<td>11.21s</td>
<td>24.08s</td>
<td>35.31s</td>
<td>99.00%</td>
<td>ContextThreads</td>
</tr>
<tr>
<td></td>
<td>11.1s</td>
<td>24.36s</td>
<td>35.48s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11.25s</td>
<td>23.68s</td>
<td>34.95s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11.73s</td>
<td>24.32s</td>
<td>35.51s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14.73s</td>
<td>39.64s</td>
<td>54.37s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td>mu</td>
<td>37.6s</td>
<td>170.21s</td>
<td>231.26s</td>
<td>99.00%</td>
<td>PosixThreads</td>
</tr>
<tr>
<td></td>
<td>37.61s</td>
<td>170.56s</td>
<td>231.48s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>37.22s</td>
<td>169.36s</td>
<td>217.62s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>37.38s</td>
<td>169.43s</td>
<td>229.93s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>37.46s</td>
<td>168.77s</td>
<td>229.25s</td>
<td>99.00%</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: TFMUL Model on xi

<table>
<thead>
<tr>
<th>Hostname</th>
<th>Usr Time</th>
<th>Sys Time</th>
<th>Elapsed Time</th>
<th>CPU Load</th>
<th>Thread Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>xi</td>
<td>7.38s</td>
<td>11.21s</td>
<td>18.69s</td>
<td>99.00%</td>
<td>QuickThreads</td>
</tr>
<tr>
<td></td>
<td>7.06s</td>
<td>11.34s</td>
<td>18.48s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.31s</td>
<td>11.85s</td>
<td>19.27s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.11s</td>
<td>12.08s</td>
<td>19.3s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.08s</td>
<td>11.37s</td>
<td>18.55s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td>xi</td>
<td>10.48s</td>
<td>18.51s</td>
<td>29.12s</td>
<td>99.00%</td>
<td>ContextThreads</td>
</tr>
<tr>
<td></td>
<td>10.28s</td>
<td>17.29s</td>
<td>27.7s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10.1s</td>
<td>16.72s</td>
<td>26.93s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9.86s</td>
<td>16.36s</td>
<td>26.34s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9.63s</td>
<td>16.63s</td>
<td>26.93s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td>xi</td>
<td>39.46s</td>
<td>165.52s</td>
<td>225.7s</td>
<td>99.00%</td>
<td>PosixThreads</td>
</tr>
<tr>
<td></td>
<td>37.84s</td>
<td>163.84s</td>
<td>222.02s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>37.71s</td>
<td>163.21s</td>
<td>221.67s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>37.46s</td>
<td>164.67s</td>
<td>222.81s</td>
<td>99.00%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>36.42s</td>
<td>162.83s</td>
<td>219.74s</td>
<td>90.00%</td>
<td></td>
</tr>
</tbody>
</table>