Center for Embedded Computer Systems
University of California, Irvine

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.

Contents

List of Figures

List of Tables

List of Listings

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

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

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 **mu**) and Intel(R) Xeon(R) architecture X5650 2.66 GHz CPU (named **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].

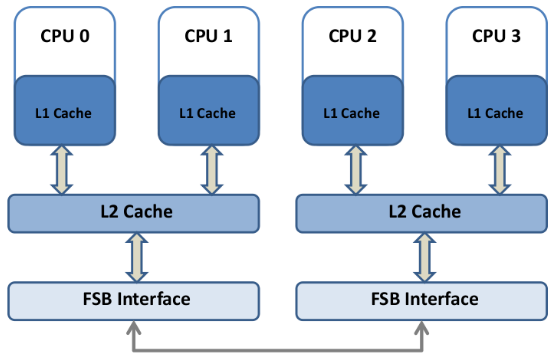

Figure 1: Intel Core 2 Quad architecture, Q9650 (mu) [3]

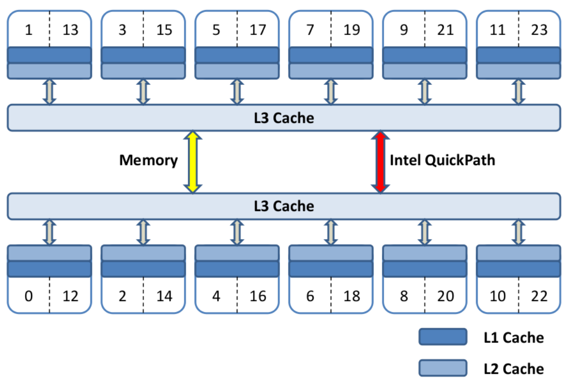

Figure 2: Intel Xeon architecture, X5650 (xi) [3]

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 **mu** and **xi** are shown in Figure 3 to 6 [1]. Table 1 and 2 list the data used in these figures.

[1]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

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	26.85s	0	26.87s	99.00%	QuickThreads
	34.11s	14.22s	48.35s	99.00%	ContextThreads
	84.8s	189.48s	274.38s	99.00%	PosixThreads
xi	22.08s	0	22.14s	99.00%	QuickThreads
	28.75s	9.79s	38.65s	99.00%	ContextThreads
	63.6s	231.25s	295.66s	99.00%	PosixThreads

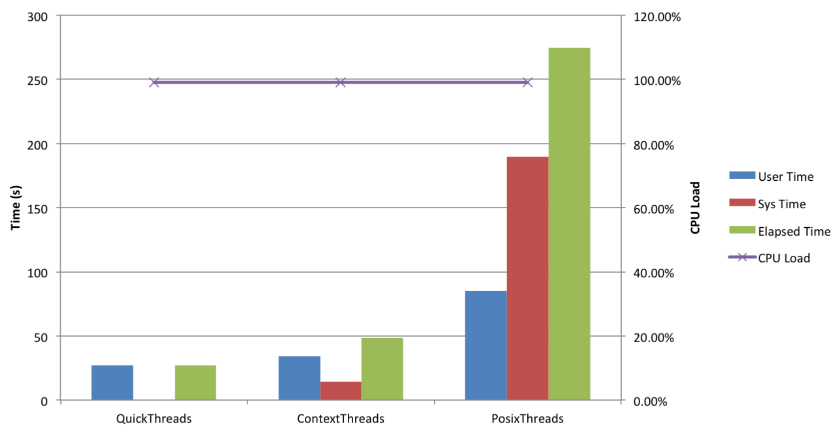

Figure 3: Simulation Results for Producer-Consumer Model on mu

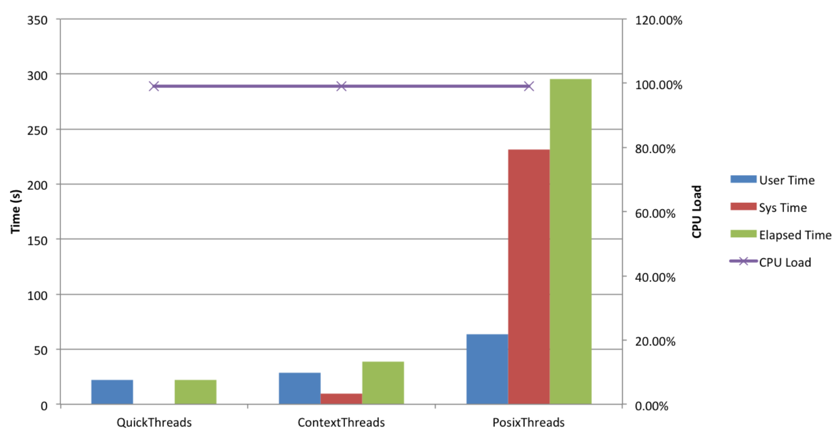

Figure 4: Simulation Results for Producer-Consumer Model on xi

Table 2: Simulation Results for TFMUL Model

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

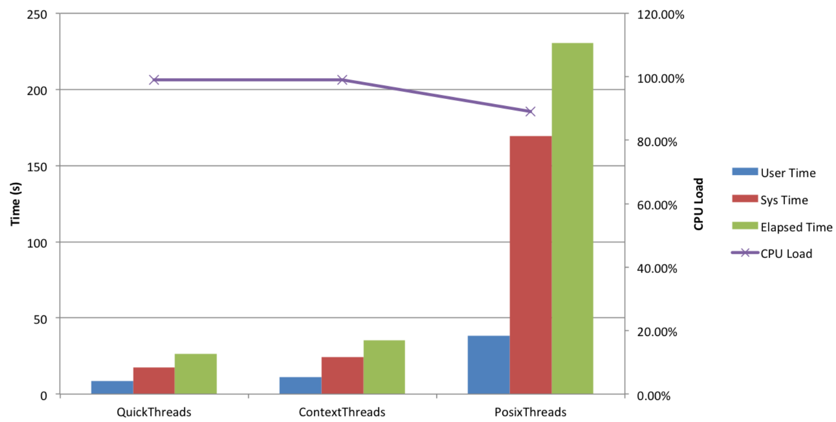

Figure 5: Simulation Results for TFMUL Model on mu

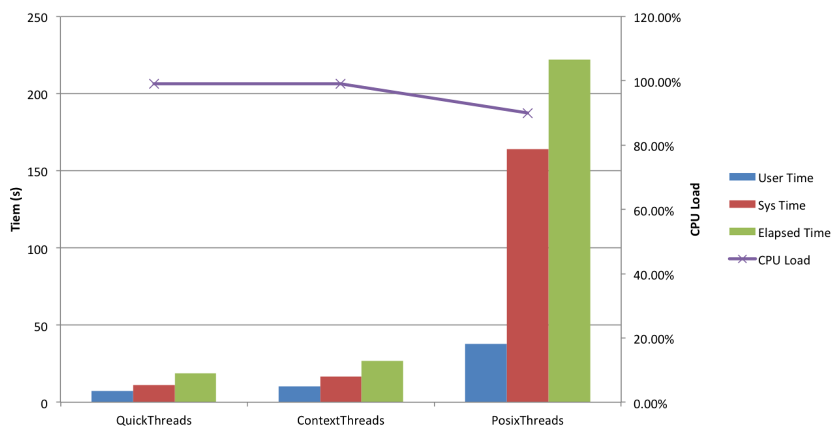

Figure 6: Simulation Results for TFMUL Model on xi

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

[1] GNU. *Linux Programmer's Manual*, March 2009.

[2] David Keppel. Tools and Techniques for Building Fast Portable Threads Packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.

[3] Guantao Liu and Rainer Dömer. Performance Evaluation and Optimization of A Custom Native Linux Threads Library. Technical Report CECS-12-11, Center for Embedded Computer Systems, University of California, Irvine, October 2012.

A Benchmark Examples

A.1 Producer-Consumer Model

Listing 1: Producer-Consumer Model

```
// prodcons.sc: simple producer-consumer example
// author:       Rainer Doemer, Guantao Liu
// 02/14/13 GL modified to test HybridThreads library
// 01/09/12 RD modified to test ooo-simulator run-ahead

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

#include <c_typed_double_handshake.sh>

#define DATA 42

#define ITERATIONS 5000000

#define EXIT_ON_HANDSHAKE // to exit on hand-shake

#define printf nop     // eliminate printing messages

#ifdef USE_FLOATING_POINT
#define FDATA 42.5e0
#endif

#ifndef MAXTHREAD
#define MAXTHREAD 1
#endif

DEFINE_I_TYPED_SENDER(char, char) // interface i_char_sender
DEFINE_I_TYPED_RECEIVER(char, char) // interface i_char_receiver
DEFINE_I_TYPED_TRANCEIVER(char, char) // interface i_char_tranceiver
DEFINE_C_TYPED_DOUBLE_HANDSHAKE(char, char)// channel c_char_double_handshake

DEFINE_I_TYPED_SENDER(short, short) // interface i_short_sender
DEFINE_I_TYPED_RECEIVER(short, short) // interface i_short_receiver
DEFINE_I_TYPED_TRANCEIVER(short, short) // interface i_short_tranceiver
DEFINE_C_TYPED_DOUBLE_HANDSHAKE(short, short)// channel c_short_double_handshake

DEFINE_I_TYPED_SENDER(int, int) // interface i_int_sender
DEFINE_I_TYPED_RECEIVER(int, int) // interface i_int_receiver
DEFINE_I_TYPED_TRANCEIVER(int, int) // interface i_int_tranceiver
DEFINE_C_TYPED_DOUBLE_HANDSHAKE(int, int)// channel c_int_double_handshake

DEFINE_I_TYPED_SENDER(llong, long long) // interface i_llong_sender
DEFINE_I_TYPED_RECEIVER(llong, long long)// interface i_llong_receiver
DEFINE_I_TYPED_TRANCEIVER(llong, long long)// interface i_llong_tranceiver
DEFINE_C_TYPED_DOUBLE_HANDSHAKE(llong, long long)// channel c_llong_double_handshake

#ifdef USE_FLOATING_POINT
DEFINE_I_TYPED_SENDER(float, float)          // interface i_float_sender
DEFINE_I_TYPED_RECEIVER(float, float)        // interface i_float_receiver
DEFINE_I_TYPED_TRANCEIVER(float, float)      // interface i_float_tranceiver
```

```
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_TRANCEIVER(double, double)      // interface i_double_tranceiver
DEFINE_C_TYPED_DOUBLE_HANDSHAKE(double, double)// channel c_double_double_handshake

DEFINE_I_TYPED_SENDER(ldouble, long double) // interface i_ldouble_sender
DEFINE_I_TYPED_RECEIVER(ldouble, long double) // interface i_ldouble_receiver
DEFINE_I_TYPED_TRANCEIVER(ldouble, long double) // interface i_ldouble_tranceiver
DEFINE_C_TYPED_DOUBLE_HANDSHAKE(ldouble, long double)
// channel c_ldouble_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_ldouble_sender pL,
#endif
  i_char_sender pc,
  i_short_sender ps,
  i_int_sender pi,
  i_llong_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);
```

```
    pc.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, l);
    pl.send(l);
    l++;
#ifdef 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
    }
    print_time();
    printf("Producer: done.\n");
  }
};

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;

    print_time();
    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);
      print_time();
      printf("Consumer: received char  c = %d (0x%02x)\n", c, c);
      ps.receive(&s);
      print_time();
      printf("Consumer: received short s = %d (0x%04x)\n", s, s);
      pi.receive(&i);
      print_time();
      printf("Consumer: received int   i = %d (0x%08x)\n", i, i);
      pl.receive(&l);
      print_time();
      printf("Consumer: received llong l = %lld (0x%016llx)\n", l, l);
#ifdef USE_FLOATING_POINT
      pF.receive(&F);
      print_time();
      printf("Consumer: received float F = %g\n", F);
      pD.receive(&D);
      print_time();
      printf("Consumer: received double D = %g\n", D);
      pL.receive(&L);
      print_time();
      printf("Consumer: received ldouble L = %Lg\n", L);
#endif
    }
    print_time();
    printf("Consumer: done.\n");
#ifdef EXIT_ON_HANDSHAKE
    pdone.send();
#endif
  }
};

behavior monitor(
#ifdef EXIT_ON_HANDSHAKE
  i_receive pdone
#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_llong_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(
#ifdef USE_FLOATING_POINT
        cF, cD, cL,
#endif
        cc, cs, ci, cl);
    consumer     cons(
#ifdef USE_FLOATING_POINT
        cF, cD, cL,
#endif
        cc, cs, ci, cl
#ifdef EXIT_ON_HANDSHAKE
        , cend
#endif
        );
    monitor    mon
#ifdef EXIT_ON_HANDSHAKE
        (cend)
#endif
      ;

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

behavior Main
{
```

```
  DUT top;

  int main(void)
  {
    print_time();
    printf("Main: starting...\n");
    top.main();
    print_time();
    printf("Main: done.\n");
    return 0;
  }
};

// EOF prodcons.sc
```

A.2 TFMUL Model

Listing 2: TFMUL Model

```
// 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>

// number of multiplications per unit
#define MAXLOOP 1000

// number of threads
#ifndef MAXTHREAD
#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,
```

```
            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; fmul8; }
    ptr53 = (char*)malloc(53);
    free(ptr73);
    par {fmul0; fmul1; fmul2; }
    ptr89 = (char*)malloc(89);
    free(ptr53);
    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);
    par { fmul0; fmul1; fmul2; fmul3; fmul4; fmul5; fmul6; fmul7; fmul8; fmul9; }
    ptr89 = (char*)malloc(89);
    free(ptr73);
    par { fmul0; }
    ptr47 = (char*)malloc(47);
    free(ptr89);
    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(ptr53);
    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);
    }
};

// EOF TFMUL.sc
```

B Measured Simulation Times for All Benchmarks and Applications

B.1 Simulation Time for Producer-Consumer Model

Table 3: Producer-Consumer Model on mu

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
	26.78s	0	26.79s	99.00%	
	26.97s	0	26.99s	99.00%	
mu	26.91s	0	26.92s	99.00%	QuickThreads
	26.85s	0	26.87s	99.00%	
	26.83s	0	26.84s	99.00%	
	34.27s	14.04s	48.34s	99.00%	
	34.24s	14.06s	48.32s	99.00%	
mu	34.54s	14.23s	48.79s	99.00%	ContextThreads
	34.41s	14.1s	48.53s	99.00%	
	34.11s	14.22s	48.35s	99.00%	
	84.8s	191.57s	276.46s	99.00%	
	84.49s	189.62s	274.21s	99.00%	
mu	84.8s	189.48s	274.38s	99.00%	PosixThreads
	84.22s	188.86s	273.18s	99.00%	
	84.16s	191.44s	275.69s	99.00%	

Table 4: Producer-Consumer Model on xi

Hostname	**Usr Time**	**Sys Time**	**Elapsed Time**	**CPU Load**	**Thread Library**
xi	22.11s	0	22.17s	99.00%	QuickThreads
	22.08s	0	22.15s	99.00%	
	22.08s	0	22.14s	99.00%	
	22.04s	0	22.11s	99.00%	
	21.8s	0	21.86s	99.00%	
xi	28.44s	10.04s	38.6s	99.00%	ContextThreads
	28.57s	10.05s	38.74s	99.00%	
	28.82	10.28s	39.22s	99.00%	
	28.75s	9.79s	38.65s	99.00%	
	28.1s	10.16s	38.37s	99.00%	
xi	63.86s	233.22s	297.9s	99.00%	PosixThreads
	63.6s	231.25s	295.66s	99.00%	
	65.28s	228.73s	294.82s	99.00%	
	63.05s	229.41s	293.27s	99.00%	
	64.88s	234.53s	300.24s	99.00%	

B.2 Simulation Time for TFMUL Model

Table 5: TFMUL Model on mu

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	8.42s	17.97s	26.41s	99.00%	QuickThreads
	8.61s	17.67s	26.29s	99.00%	
	8.28s	17.56s	25.86s	99.00%	
	8.41s	17.73s	26.15s	99.00%	
	8.61s	18.31s	26.94s	99.00%	
mu	11.21s	24.08s	35.31s	99.00%	ContextThreads
	11.1s	24.36s	35.48s	99.00%	
	11.25s	23.68s	34.95s	99.00%	
	11.17s	24.32s	35.51s	99.00%	
	11.73s	24.89s	36.64s	99.00%	
mu	37.6s	170.21s	231.26s	89.00%	PosixThreads
	37.61s	170.56s	231.48s	89.00%	
	38.22s	169.36s	230.67s	89.00%	
	37.38s	169.43s	229.93s	89.00%	
	37.46s	168.77s	229.25s	89.00%	

Table 6: TFMUL Model on xi

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	7.38s	11.21s	18.69s	99.00%	QuickThreads
	7.06s	11.34s	18.48s	99.00%	
	7.31s	11.85s	19.27s	99.00%	
	7.11s	12.08s	19.3s	99.00%	
	7.08s	11.37s	18.55s	99.00%	
xi	10.48s	18.51s	29.12s	99.00%	ContextThreads
	10.28s	17.29s	27.7s	99.00%	
	10.1s	16.72s	26.93s	99.00%	
	9.86s	16.36s	26.34s	99.00%	
	9.63s	16.63s	26.39s	99.00%	
xi	39.46s	165.52s	225.7s	90.00%	PosixThreads
	37.84s	163.84s	222.02s	90.00%	
	37.71s	163.21s	221.67s	90.00%	
	37.46s	164.67s	222.81s	90.00%	
	36.42s	162.83s	219.74s	90.00%	