Center for Embedded Computer Systems
University of California, Irvine

A Hybrid Thread Library for Efficient Electronic System Level Simulation

Guantao Liu and Rainer Dömer

Technical Report CECS-13-11
October 8, 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 Hybrid Thread Library for Efficient Electronic System Level Simulation

Guantao Liu and Rainer Dömer

Technical Report CECS-13-11
October 8, 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

In recent years, fast simulation of Electronic System Level (ESL) models has gained significant attraction due to the explosive growth of system size and complexity. As a System-Level Description Language (SLDL), SpecC language has explicit advantages in specifying parallelism and hierarchy in ESL models, which creates potential for efficient parallel simulation. Currently, the SpecC simulator utilizes QuickThreads for its sequential simulator, and PosixThreads for the parallel one. In this thesis, we will propose the design and implementation of a hybrid thread library for efficient system-level simulation in SpecC. Our proposed thread library is based on QuickThreads and PosixThreads and combines the advantages of both kernel-level and user-level thread libraries. Systematic performance evaluation indicates that the new thread library achieves a significant improvement in simulation time for parallel benchmarks and real-world embedded applications.

Contents

List of Figures

List of Tables

List of Listings

A Hybrid Thread Library for Efficient Electronic System Level 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

In recent years, fast simulation of Electronic System Level (ESL) models has gained significant attraction due to the explosive growth of system size and complexity. As a System-Level Description Language (SLDL), SpecC language has explicit advantages in specifying parallelism and hierarchy in ESL models, which creates potential for efficient parallel simulation. Currently, the SpecC simulator utilizes QuickThreads for its sequential simulator, and PosixThreads for the parallel one. In this thesis, we will propose the design and implementation of a hybrid thread library for efficient system-level simulation in SpecC. Our proposed thread library is based on QuickThreads and PosixThreads and combines the advantages of both kernel-level and user-level thread libraries. Systematic performance evaluation indicates that the new thread library achieves a significant improvement in simulation time for parallel benchmarks and real-world embedded applications.

1 Introduction

Nowadays, the fast development of modern embedded systems imposes enormous challenges to the fast simulation of Electronic System Level (ESL) models. The traditional user-level and kernel-level thread libraries have both advantages and disadvantages in the implementation of the parallel system-level simulator. User-level thread library such as QuickThreads owns a high computation performance but it does not support parallel execution; kernel-level thread library (PosixThreads) can work in parallel on multiprocessor host, but the heavy load of thread initialization and synchronization will slow down the simulation. A modification in threading model is necessary to achieve better performance of the parallel ESL simulators.

1.1 Background on Multithreading

A thread is a sequential piece of code that is executing in the operating system. It is the smallest unit of programmed instructions that can be scheduled independently by OS. In contrast to a process in the operating system, a thread owns no memory or resources of its own except for a stack, a copy of the registers and the program counter. All the threads in the same process share the address space. Therefore, context switching between threads in the same process is much more efficient than context switching between processes. Also, the creation and deletion of a thread is much simpler than a process. Sometimes, a thread is called a lightweight process (LWP).

Generally speaking, there are two ways to implement threads in an operating system. The major difference between these two kinds of threads is the relationship between the thread and the operating system scheduler. One kind of threads is created and scheduled by the kernel, always called kernel-level threads. For these threads, all the thread manipulation is done by the in-kernel scheduler. By mapping one userspace thread to one kernel thread, it allows multiple threads to run in parallel on multiprocessors. Also, kernel-level threads offer synchronization primitives such as mutexes and conditional variables against concurrent access to the shared variables. The well-known PosixThreads library is implemented in this way and its threading model is as Figure 1.

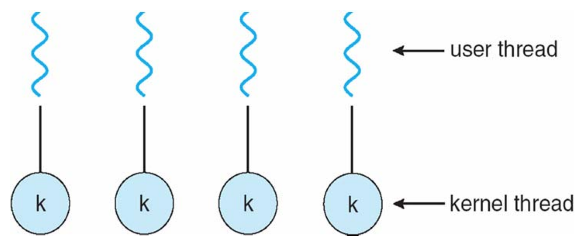


Figure 1: Kernel-level Threading Model [19]

On the other hand, threads can also be implemented in userspace libraries. In such a case, the operating system kernel is not aware of the threads, and all the thread management is done by the runtime thread library. Such kinds of threads are called user-level threads and a good example is QuickThreads. QuickThreads library is a basic cooperative userspace threads package. As the thread operations require no kernel intervention, QuickThreads have extremely low overheads and can achieve a high computation performance. However, because the threads in the Quickthreads library are invisible to the operating system, the user-level threads are only allowed to gain access to one core in the CPU. Thus, simultaneous access to multiple processors is not possible and user-level thread libraries are always uniprocessors threads package. The thread mapping model of a user-level thread library such as QuickThreads is given in Figure 2.

As user-level threads require no kernel intervention, switching between user-level threads is much faster than that of kernel-level threads. This often matters to concurrent systems that use a

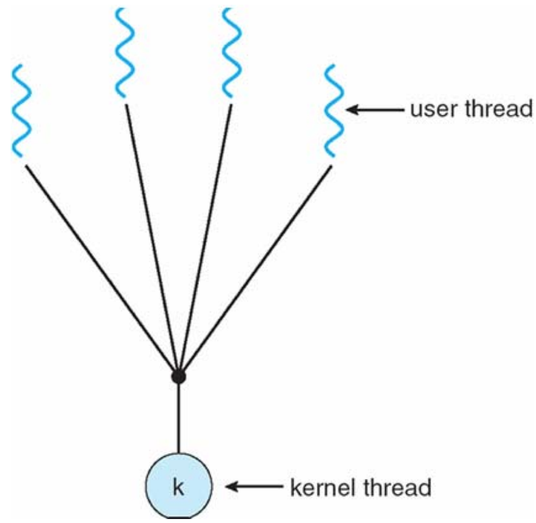


Figure 2: User-level Threading Model [19]

large number of very short-lived threads and consist of a large amount of context switching. Another advantage of user-level threads is that it does not require switching to kernel-level scheduler and again switch back to the userspace program during context switching. Both of these features make user-level threads very computation efficient on the uniprocessor models. Also, as user-level threads do not need to manage kernel structures, the creation and deletion of a user-level thread is often faster than a kernel-level thread. The only drawback of the user-level thread library is that kernel-level threads are able to run simultaneously on multiprocessors, while purely user-level threads cannot achieve this feature. In order to obtain a better performance of the application program (say ESL simulators), it requires improvement on the models and implementation of the thread library.

1.2 Problem Description

1.2.1 SLDL Execution Semantics

System Level Description Language (SLDL) is widely used today to describe Electronic System Level (ESL) models. In contrast to the traditional C/C++ languages which are sequential and flat, SLDL explicitly specifies the behavioral and structural hierarchy in the design model. The key ESL concepts in the system model provide possibilities for multithreading and parallelism to enhance the simulation performance. In order to guarantee fast and accurate model validations, the current SLDL simulation is based on the traditional Discrete Event (DE) simulation.

The traditional Discrete Event (DE) Simulation is driven by events and simulation time advances. Usually, it is comprised of multiple delta and timed cycles. During each of these cycles, threads are moved among different queues in the scheduler. Basically, there are five queues (READY, RUN, WAIT, WAITFOR, COMPLETE) in the scheduler to define different states of a thread [6] [9]. When a thread is assigned to one core in the CPU and starts running, it is moved from the READY queue to the RUN queue. If the thread is waiting for some events, it is put into the WAIT queue. Similarly, when the thread is waiting for time advances, it will be suspended and moved to the WAITFOR queue. After a thread finishes its job, it will end in the COMPLETE queue. Within the entire simulation cycles, the scheduler is called to update queues and move the simulation forward whenever events are delivered or time increases [6]. At any time the RUN queue is not full, the scheduler will pick up the next thread to run from the READY queue randomly. When the READY queue becomes empty, it would be filled again by waking up threads in the WAIT queue who have received events they were waiting for. In this case, threads are moving from WAIT queue to READY queue and a delta-cycle is advanced. If the READY queue is still empty after waking up all the available threads in the WAIT queue, a new timed-cycle begins and some threads with the earliest timestamp in the WAITFOR queue are migrated to the READY queue. If no more threads are available in the WAITFOR queue and the READY queue is still empty, the simulation program terminates. Figure 3 shows a complete flow chart of Discrete Event (DE) simulation with delta and timed cycles specified.

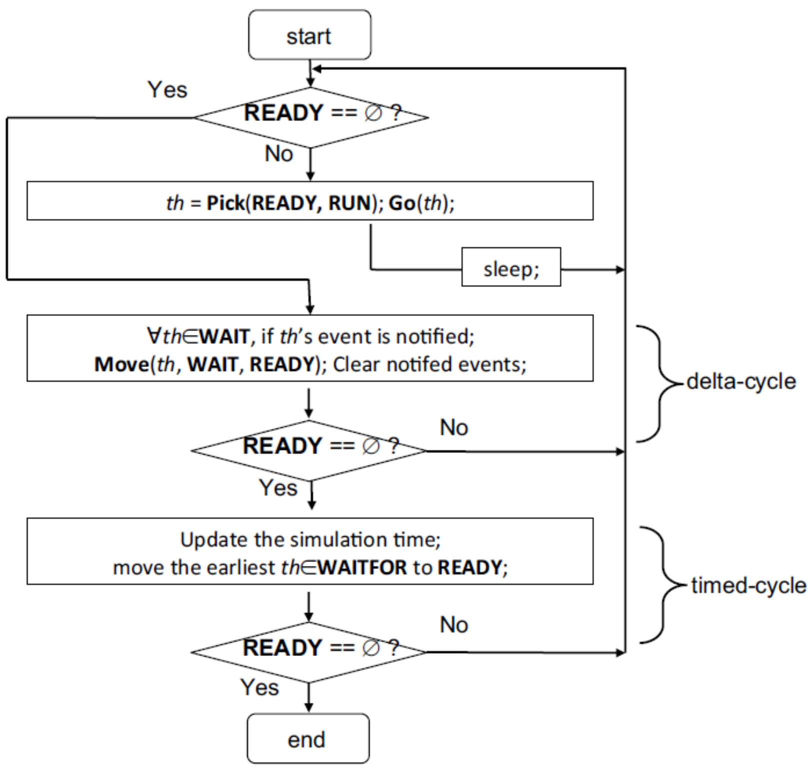


Figure 3: Traditional Discrete Event Simulation Scheduler [9]

Parallel Discrete Event Simulation (PDES) is an enhancement of the traditional Discrete Event (DE) simulation, which has the potential to efficiently map the explicit parallelism in the system model onto the parallel cores available on the simulation host [10]. Compared with the Discrete Event Simulation, the scheduler will issue as many threads from the READY queue as CPU cores are available in each cycle. Even though several threads are busy running on different CPUs, there is only one centralized scheduler for the whole simulation program to maintain the simulation semantics. An extended control flow of the Parallel Discrete Event Simulation is shown in Figure 4.

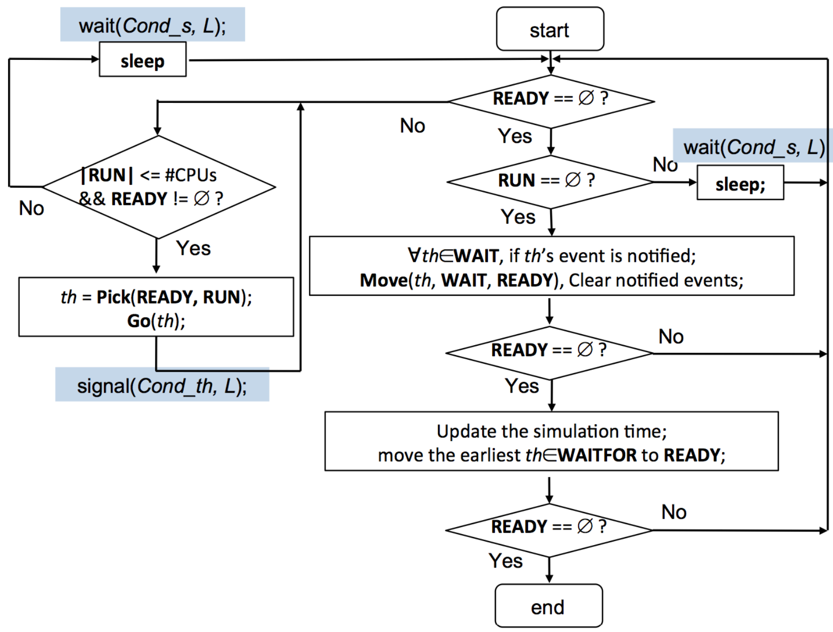


Figure 4: Parallel Discrete Event Simulation Scheduler [9]

In order to protect the shared scheduling resources in the simulation engine (including the thread queues and event lists, and shared variables in communication channels of the application model [6]), one central lock (mutex) is necessary to ensure the mutually exclusive access by the concurrent threads. As the length of READY queue (the number of threads blocked at the same time) varies with time due to thread creation or migration among different queues, synchronization primitives such as barriers are not suitable in this case. Thus, as described in Figure 5, a thread in the parallel simulation program will grab the centralized lock whenever it needs to update the shared scheduling resources. Also as soon as the thread finishes the scheduling task (*Go(schedule)* in Figure 5) and begins to perform any functions in the application model, it will release the centralized mutex and run in parallel with other threads.

1.2.2 Software Stack of SpecC Simulators

SystemC and SpecC are two examples of the System Level Description Languages (SLDL). Both of them use the Discrete Event approach to implement simulations. However, as the threading model in

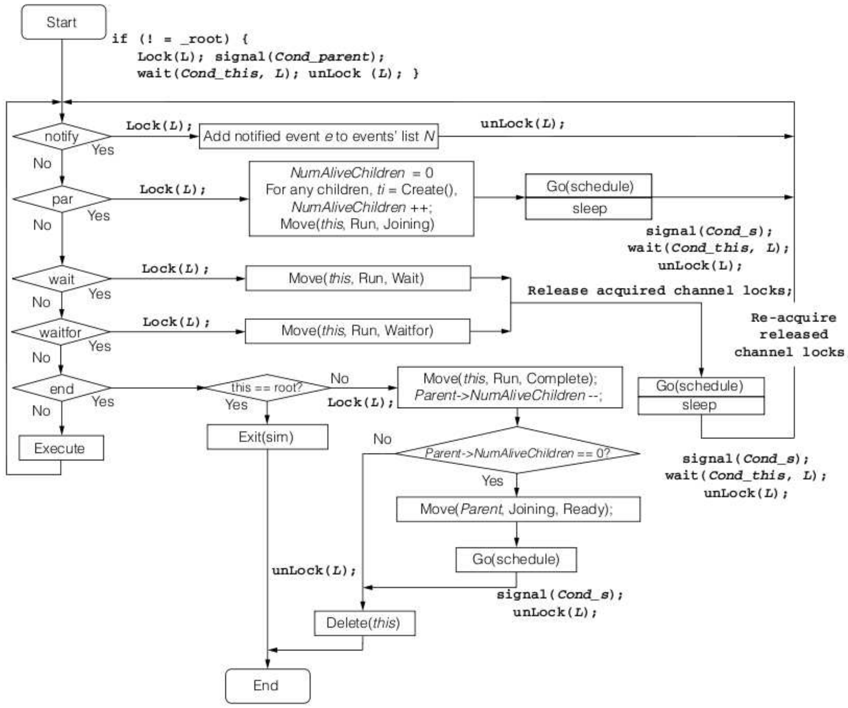


Figure 5: Lifecycle of a Thread in Parallel Discrete Event Simulation [6]

SpecC is explicitly specified as preemptive, it carries the promise of utilizing the available hardware resources on a multi-core host to increase the simulation performance. Therefore, Parallel Discrete Event Simulation (PDES) is also utilized in SpecC to implement parallel simulators. Currently there are two major synchronization paradigms in the SpecC parallel simulators, synchronous and out-of-order (asynchronous) [7]. Synchronous PDES ensures in-order event execution while out-of-order model will proceed as long as every event is safe. Out-of-order PDES can be faster than the synchronous one as it breaks the temporal barriers which prevent effective parallel execution. However, the synchronous paradigm is more stable and predictive at the current stage. So in this technical report we will only aim at the thread library used in regular synchronous Parallel Discrete Event Simulation (PDES).

The software stack of the SpecC Discrete Event (DE) simulator is shown on Figure 6. The application is transformed from SpecC programs, by replacing the structural and behavioral blocks with corresponding structs in the simulation library. The simulation library provides these structural and behavioral structs and is responsible for the scheduling of the simulation program. The thread library is invoked by the simulation library to achieve thread manipulations like creation and context switching. The thread library in turn calls Linux OS APIs to fulfill the specific functionalities. At the bottom, created threads will run on the hardware units (CPUs) to perform their work.

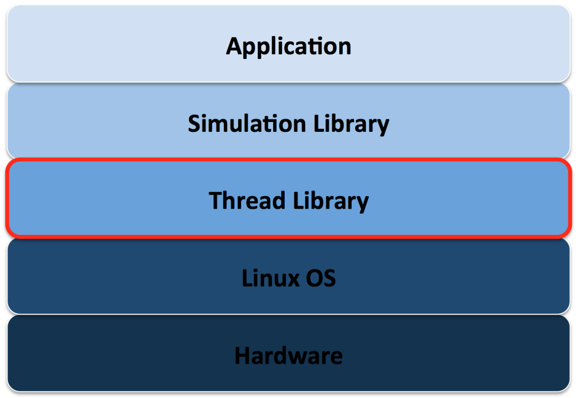


Figure 6: Software Stack of SpecC Simulators

1.2.3 Problem Definition

In this technical report, we will focus on the thread library in the software stack of the SpecC parallel simulators, as highlighted in the red box in Figure 6. In other words, we will utilize the original Discrete Event (DE) simulation engine in the SpecC development environment but design and implement a compatible thread library to achieve higher efficiency in thread manipulation. The threading model will not modify any scheduling mechanism in the original simulator but be designed to reduce the overheads in thread creation/deletion and context switching. In order to utilize the full hardware resources on the multiprocessor hosts, the underlying operating system needs to be aware of the multiple simulation threads running in parallel and a kernel-level thread library is necessary.

1.3 Goals

To implement a thread library for efficient ESL simulators in the SpecC development environment, a number of goals are identified as follows:

- Efficiency: The new thread library should combine the benefits of PosixThreads and QuickThreads, as shown in Figure 7. As such, the hybrid approach should, on multi-core machines, reach a minimum performance similar to PosixThreads, and on single-core machines, similar performance as QuickThreads.
- Parallelism: In order to utilize the power of available multiple processors, the new thread library must truly support multi-core hosts and parallel simulation of the design model.
- Compatibility: As the threading library is part of the SpecC Discrete Event (DE) simulators, we could not make the implementation completely free of restrictions but had to follow a certain interface (function interface to simulation library). Also, the new thread library must be replaceable with the original PosixThreads library and follow the simulation semantics specified in the SpecC parallel simulators.

- Portability: The new thread library should limit the machine-dependent code and be portable to a number of platforms.

- Accuracy: The simulation should always be correct regardless of the scale of the model.

The relationship among the new thread library, PosixThreads and QuickThreads is shown in Figure 7.

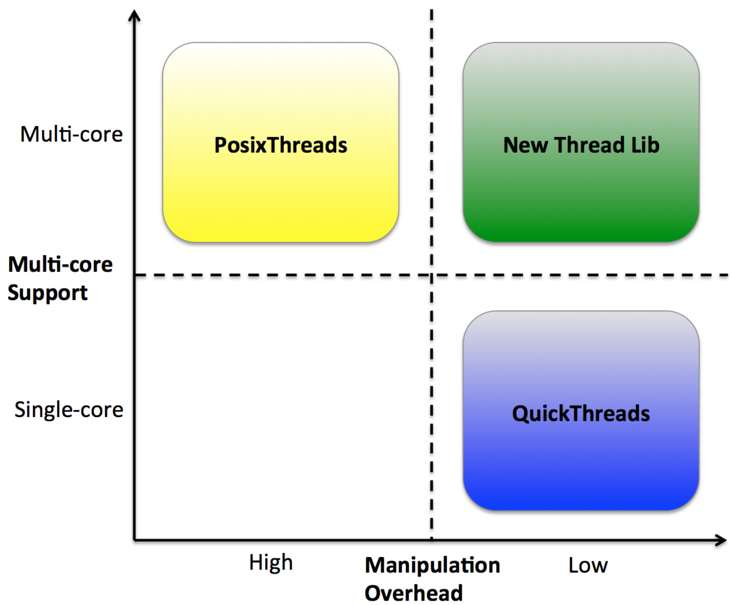


Figure 7: Relationship Among Three Thread Libraries

1.4 Related Work

How to accelerate the simulation speed of Electronic System Level models has been a well-studied subject for the past few years. While the single threaded simulation kernel inherent to System-Level Description Languages (SLDL) [11] prevents it from utilizing the parallel computation resources available in today's common multi-core CPUs, [6] [7] [8] [11] [18] extend the simulation kernels in the System-Level Description Languages to speed up system simulation on multi-core machines. Specifically, [6] and [11] attempt to parallelize simulation engine by executing several threads concurrently in the evaluation phase of Discrete Event Simulation (DES) and utilizing as many cores as possible. In [8], the simulation schedulers are distributed to every processing node and run a subset of the application modules. To guarantee the simulation semantics, all local schedulers need to synchronize both channel and time at the end of each delta cycle. However, the partitioning of the application must be done by the designer manually. Compared with [8], [18] proposes a centralized master thread for scheduling and a group of worker threads to utilize the parallel computing power on multicore CPUs. While the centralized scheduler is responsible for synchronization and communication, the execution of SystemC processes can be partially or in total offloaded onto the pool of worker threads. Based on these researches, [11] breaks the simulation-cycle barriers and let data-independent threads run asynchronously and in parallel.

Even though all these approaches can speed up ESL simulation, they make improvements on the simulation models. In contrast, [17] presents a DE simulation modeling strategy with a corresponding thread model. The key idea is to break the bottleneck of a centralized scheduler and a global simulation time in SystemC. Then each group of threads has its distributed time and all the synchronization and communication is accomplished with timed messages. The partitioning of the threads is explicitly controlled by the design and the same group of threads are executing on the same physical core. Above each physical CPU, there is an associated Posix thread for local scheduling, and a group of Quick threads executing simulation tasks. The software architecture of this SystemC distributed simulation engines is as shown in Figure 8.

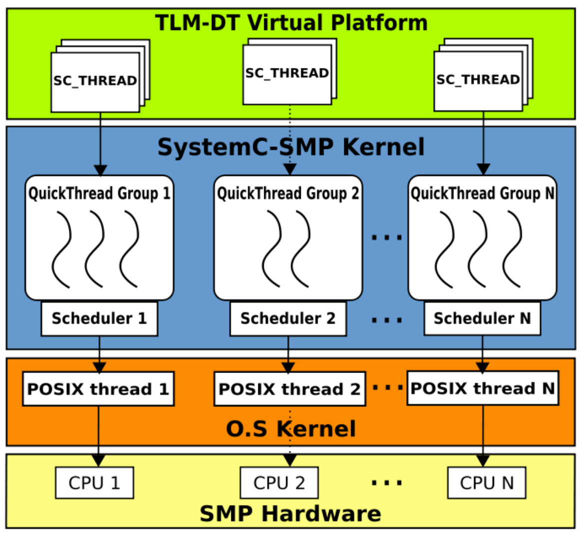


Figure 8: Software Architecture of the SystemC Distributed Simulator [17]

The technical report [14] and [16] talked about the implementation and evaluation of a native Linux-based thread library for fast embedded system simulation. This new custom thread library named LiteThreads takes advantages of native components in Linux operating system, achieving low overhead of thread initialization and manipulation. Specifically, the *clone* system call is used to create new threads and Linux futex (Fast User Space Mutex) provides the necessary synchronization primitives in the thread library. As the features of Posix threads are cut down in *clone* and context switching could be sometimes completed in the userspace, the LiteThreads library owns a better performance than the regular PosixThreads. This conclusion is validated by the simulation results in [14]. The software stack of the SpecC simulator using LiteThreads is shown in Figure 9.

Even though LiteThreads has superior performance to PosixThreads, it does not support parallel simulation. As lots of features of PosixThreads are eliminated in the LiteThreads package, several

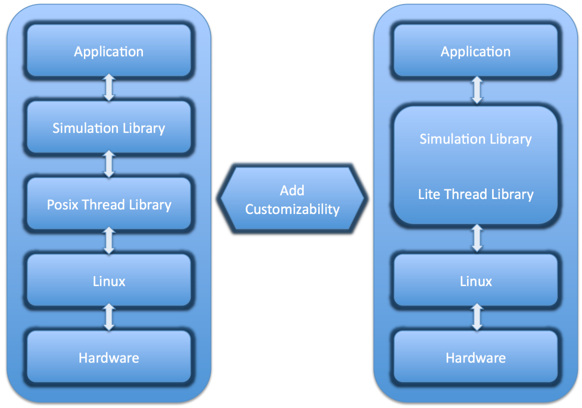


Figure 9: Software Stack of SpecC Parallel Simulator using LiteThreads [16]

Linux functions (such as *malloc* and *printf*) which depend on the kernel structs in Posix threads will fail in the parallel simulator using LiteThreads. So currently LiteThreads only supports the SpecC sequential simulator.

The remainder of this technical report is organized in the following manner. First, we will propose a new hybrid mode of parallel thread library to integrate the advantages of both kernel threads and user threads. Then, to demonstrate the mechanisms of the new thread library, a parallel program is used to illustrate the simulation process. Finally, we utilize several parallel benchmarks and embedded applications to evaluate the performance of the hybrid thread library.

2 Basic Principles of HybridThreads

2.1 Motivation for the HybridThreads Library

Based on the introduction and evaluation in [15], we can conclude the features of QuickThreads, ContextThreads and PosixThreads as shown in Figure 10.

	Context Switching	Thread Create/ Delete	Portability	Multi-core support	User vs. Kernel
PosixThreads	1	1	5	yes	Kernel
ContextThreads	4	4	3	no	User
QuickThreads	5	5	2	no	User

(1: worst 2: bad 3: medium 4: good 5: best)

Figure 10: Comparison of Popular Thread Libraries

As indicated in Figure 10, kernel-level thread library such as PosixThreads can guarantee to work right on multiprocessor machines but performs poorly due to the load on the system. QuickThreads and ContextThreads are implemented on top of kernel threads, which perform well but have deficiency in parallel execution. With the adventure of the multiprocessor machines, a natural extension to utilize the power of multiple processors as well as guarantee the computation performance is to integrate the PosixThreads (kernel-level) library and QuickThreads (user-level) library, which is the basic idea of the new HybridThreads library [13]. The thread mapping model of the new thread library is shown as Figure 11.

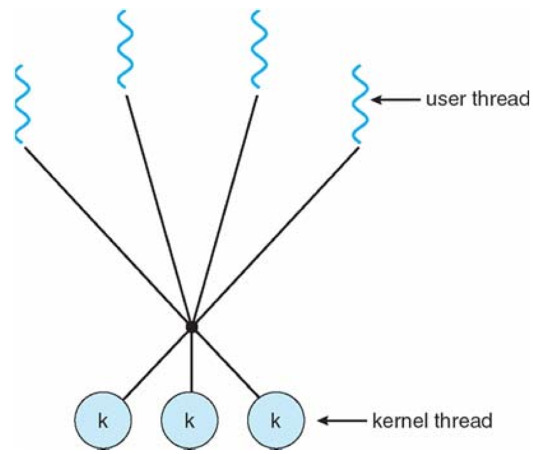


Figure 11: HybridThreads Threading Model [19]

2.2 Mechanisms of the HybridThreads Library

2.2.1 Basic Ideas of the HybridThreads Library

One basic requirement of the SpecC Parallel Discrete-Event Simulation scheduler is to utilize all the hardware resources on the multi-core hosts. Hence, kernel-level threads library is necessary for the SpecC parallel simulation engine. However, as outlined in [15] and Section 2.1, the kernel-level thread library is too heavyweight and the high cost of kernel-level overhead will burden the whole simulators. Based on the simulation results in [15], it is obvious that PosixThreads library is 10 times slower than QuickThreads. In order to utilize the full power of multiprocessor machines and alleviate the overhead in thread initialization and synchronization, a straightforward idea is to combine the kernel-level thread library and user-level thread library. The user threads that are completely managed by the userspace cannot run in parallel on different CPUs, although they will achieve a great performance in computation on uniprocessor system. On the other hand, due to the usage of kernel scheduler, kernel-level threads can run in different cores but the synchronization and sharing resources among threads are more expensive than user-level threads. When we build the user-level threads (Quick threads) above kernel-level threads (Posix threads), the new thread library will combine the advantages of both thread libraries. This is just what we implemented as the HybridThreads library.

In HybridThreads, the thread library will create no more kernel-level (Posix) threads than the number of available CPU cores. These kernel-level threads are affiliated to the CPU cores in a one-to-one mode. They are used to manage all the kernel data structs and synchronization among different processors. The Linux OS would be only aware of these kernel-level threads and they exist from the beginning to the end of the simulation. For all the userspace threads created by the parallel simulator, they are treated as user-level (Quick) threads and manipulated through the thread's stack pointer. All of these user-level threads are mapped to the CPU cores and running sequentially above each kernel-level thread. Thus, a ”full” user-level thread library is running on each CPU core, and just working as what QuickThreads usually does. In the thread initialization, only the thread's stack is updated with the new execution context. During a context switch, a helper function saves the register values and program counter of the old thread on to its stack, and then switches to the stack of the new thread and invokes the client function on behalf of the new thread. Only when it needs to communicate or synchronize among different processors, the kernel-level threads are used to guarantee the safety and integrity of the concurrent execution. Therefore, as long as there are enough threads running on one core and they only communicate with each other, the new HybridThreads library has a similar performance as QuickThreads on each core. The synchronization and sharing resources would incur some overheads in user and kernel level, but it would be compensated by the performance enhancement brought by parallel execution. The new software hierarchy of SpecC parallel simulators is indicated in Figure 12.

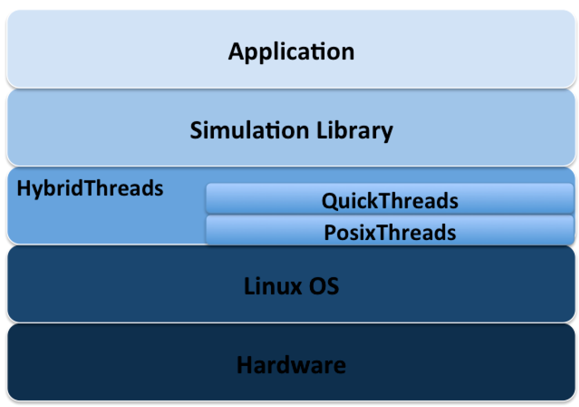


Figure 12: New Software Stack of SpecC Parallel Simulators

2.2.2 Work-stealing vs. Work-sharing

Work-stealing [2] is a good scheduling algorithm to balance the work load on different processors. When a core is idle, it will choose another core and attempt to steal tasks from that core. Generally speaking, a work-stealing scheduler can achieve near-optimal scheduling in an environment which exposes approximately 10 tasks for each core and has poorly-balanced workload over different cores. In contrast, a work-sharing [1] scheduler is preferred in a dedicated environment with a well-balanced workload. Besides, when using work-sharing, the scheduler usually assigns the threads to each core from a thread-pool, in a round-robin fashion to take advantage of code-locality.

As the SpecC PDES scheduler makes use of a centralized READY queue and only issues as many threads as CPU cores are available, the new HybridThreads library adopts work-sharing scheduler to assign tasks to each processor. On average, there is only one thread available on each core, and when one core has no more useful work to perform, the simulator will pick up a new thread from the centralized READY queue and assign it to the idle processor. In this case, work-sharing schedulers work more efficiently than work-stealing ones.

2.3 Simulation Process of a Parallel Program using HybridThreads

In order to better explain the mechanisms of the HybridThreads library, we will first illustrate the data structures used in HybridThreads, and then describe a simulation process of a parallel program which is based on HybridThreads.

Figure 13 illustrates the data structures for the kernel-level threads and user-level threads in HybridThreads, as well as the global variables used to keep HybridThreads working correctly. *PThread_base* holds all the properties of a kernel-level thread in HybridThreads, and *thread_base* is used for the user-level threads. Remarkably, each kernel-level thread is locked to a specific core and the number of *PThread_base* is as many as the available cores in the machine. In the view of the SpecC parallel simulator, a *thread_base* instance refers to a thread in the program and *PThread_base* is invisible to the simulation scheduler. Specifically, *QThreadStart* and *QThreadEnd* in *PThread_base* point to the first and last user-level thread (*thread_base*) above this Posix thread. In *thread_base*, *PThreadPtr* points back to the kernel-level thread and *PrevThreadBase* and *NextThreadBase* are used to build a double linked list to manipulate all the user-level threads running on one kernel-level thread (core). When a user-level thread finishes, *PrevThreadBase* and *NextThreadBase* in its data structure are set to NULL to break from the double linked list.

Among all the global variables, *ActivePThreads* and *ActiveQThreads* are used to keep record of the number of active kernel-level threads and user-level threads. *BusyPThreads* is the number of current busy Posix threads in the program while *MaxPThreads* is the maximum number of active kernel-level threads during the simulation. As discussed in Section 2.2.1, at any time the number of Posix threads should be no more than the number of available CPU cores in the machine. In the five arrays, each item holds the status of the kernel-level threads and user-level threads on each core. The *RootThread* keeps the synchronization primitives of the root thread.

Next, we list the code of a short parallel SpecC program to describe how HybridThreads works. List 1 gives the code and Figure 14 to 20 describe the whole simulation process of the example.

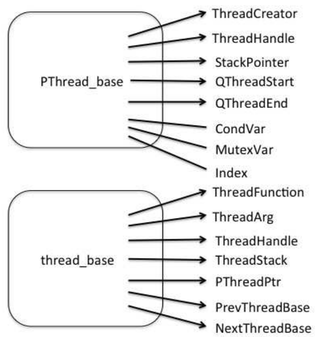
Listing 1: A Parallel SpecC Program

```
// par.sc

#include <stdio.h>
#include <assert.h>

behavior A
{
  void main(void)
  {
    for(int i; i<100; i++);
```

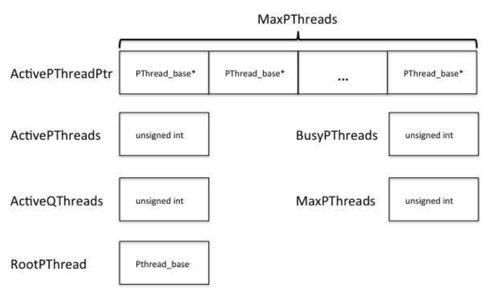
```
  }
};

behavior B
{
  void main(void)
  {
    for(int i; i<200; i++);
  }
};

behavior Main
{
  A A1;
  B B1;
  A A2;
  B B2;
  A A3;
  B B3;

  int main(void)
  {
    puts("par: Starting...");

    par { A1.main();
          B1.main();
          A2.main();
          B2.main();
          A3.main();
          B3.main();
        };

    puts("par: Done.");
    return(0);
  }
};

// EOF
```

In this SpecC example, there will be 6 threads running in parallel. At the beginning of the simulation, HybridThreads will initialize all the global variables in the thread library, and allocate memory space for Arrays *ActivePThreadPtr*, *IdleIndex* and so on (Figure 14.1 and 14.2). To create the root thread, only an instance of *thread_base* is created and points to the global struct *RootThread* (Figure 14.3). When the *par()* structure begins, four kernel-level threads are created along with the user-level threads. At that time, all the user-level threads are floating and the kernel-level threads are idle too. All these kernel-level threads are affiliated to one core each in the machine and are tracked by a corresponding item in *ActivePThreadPtr*. The global variables *ActiveQThreads* and *ActivePThreads* are incremented accordingly when we create a new user-level thread and kernel-level thread. As all the kernel-threads are still idle at this time, *BusyPThreads* remains zero and *IdleIndex* is initialized with the index of the idle Posix threads (*ReverseIndex* holds each idle Posix thread's index in *IdleIndex*). This process is illustrated in Figures 14.4 to 15.10. For the remaining threads A3 and B3, since the host machine (**mu**) has only 4 cores, they are only created as user-level threads (Figure 15.11 and 14.12). According to the control flow illustrated in Figure 4, four child threads will be issued and they will attach to four idle kernel-level threads respectively (Figure

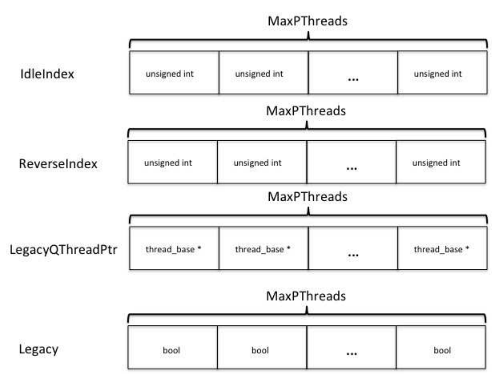
16.13 to 16.17). *BusyPThreads* is incremented as a new kernel-level thread becomes busy. Then these four kernel-level threads will be running simultaneously. In Figures 16.18 to 17.21, when one thread (thread B1 and thread A3) is done, the parallel scheduler will wake up another user-level thread (thread A3 and thread B3) before it dies. As all other Posix threads are busy, the new userspace thread will attach to the current core and continue executing after switching the thread's stack. After all the six threads are issued, the kernel-level thread (thread A2 in Figure 17.22, B3 in Figure 17.24, B2 in Figure 18.27 and A1 in Figure 18.29) will be suspended when it finishes the userspace function in the user-level thread. If we are going to delete the terminated child threads (in Figures 19.31 to 19.36), the thread library needs to copy the execution context of the last user-level thread (thread A1, A2, B2 and B3) above each Posix thread to *LegacyQThreadPtr*. The reason is that all the Posix threads are suspended in the userspace thread's stack. When the simulation terminates (Figure 20.37 to 20.39), all the active Posix threads and remaining execution contexts in *LegacyQThreadPtr* are cleaned up. Meanwhile, all the data and structs in HybridThreads are freed.



(a) Data Structures for kernel-level Threads and user-level Threads



(b) Global Variables in HybridThreads (Part 1)



(c) Global Variables in HybridThreads (Part 2)

Figure 13: Data Structures in HybridThreads

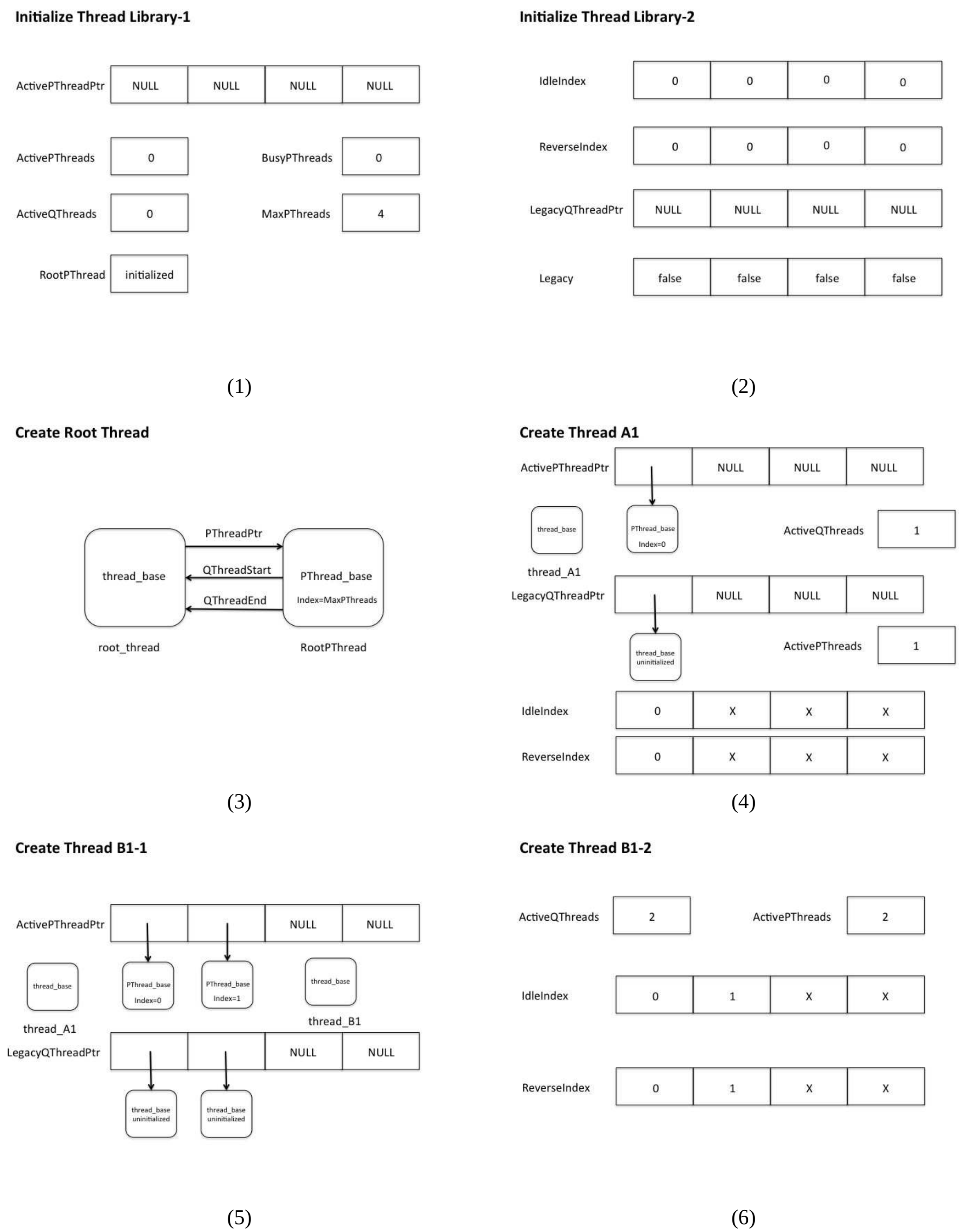


Figure 14: Simulation Process of HybridThreads (Part 1)

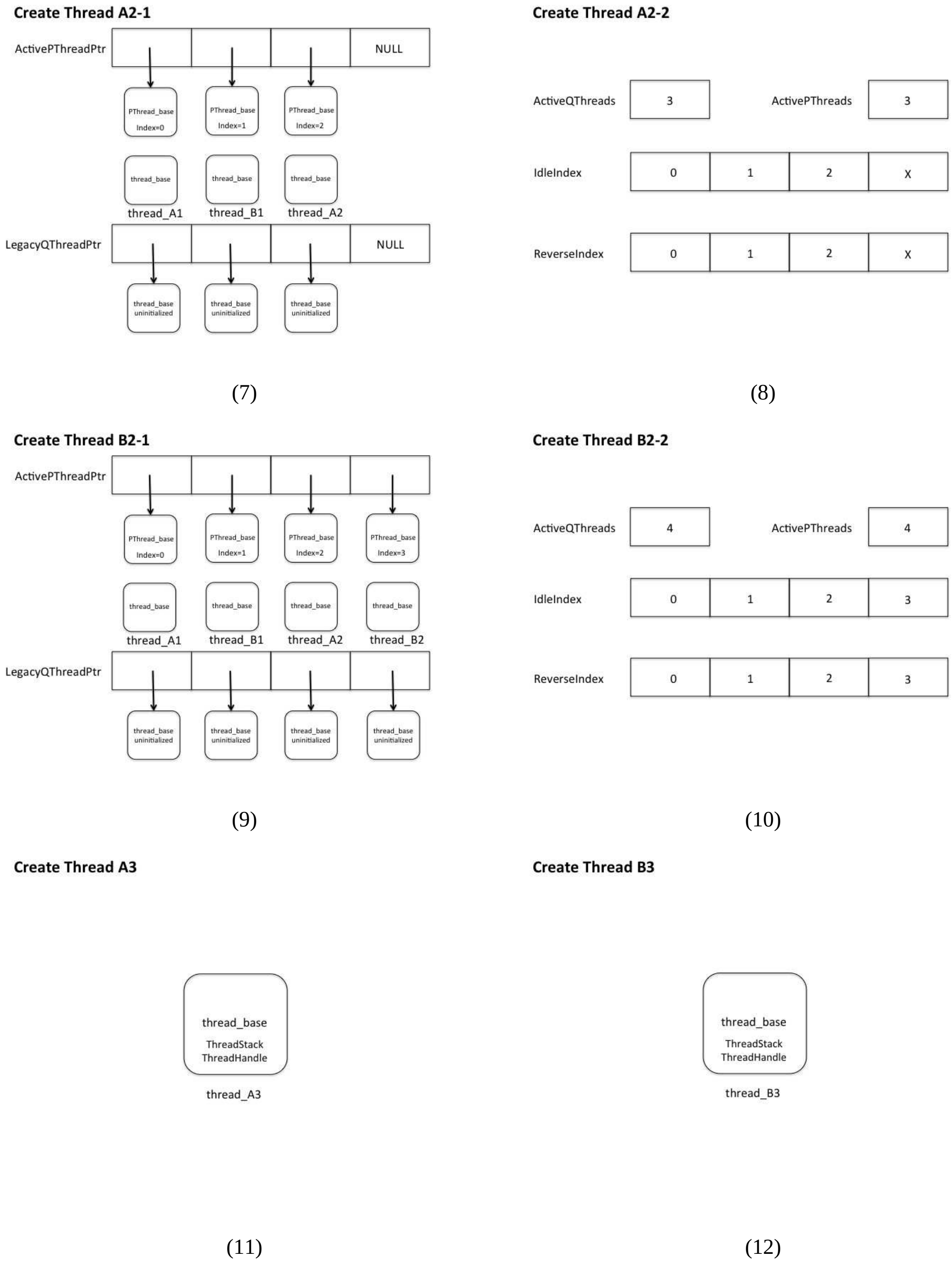
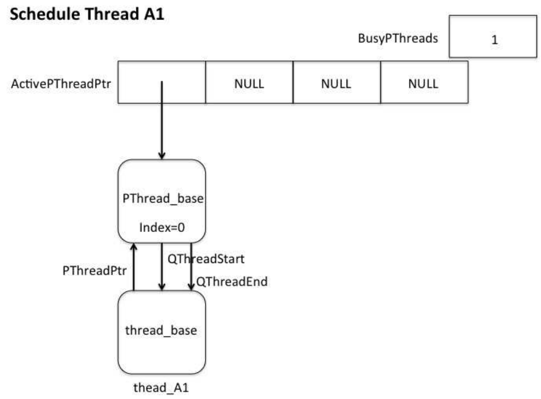
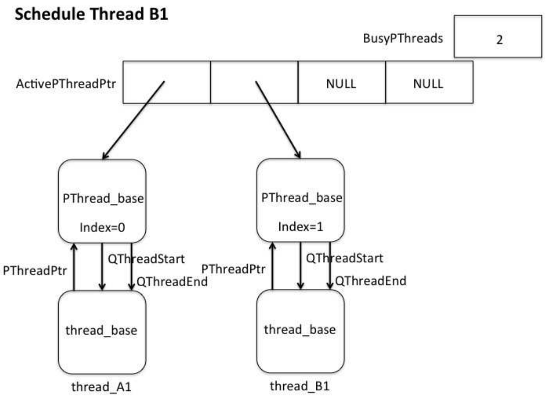


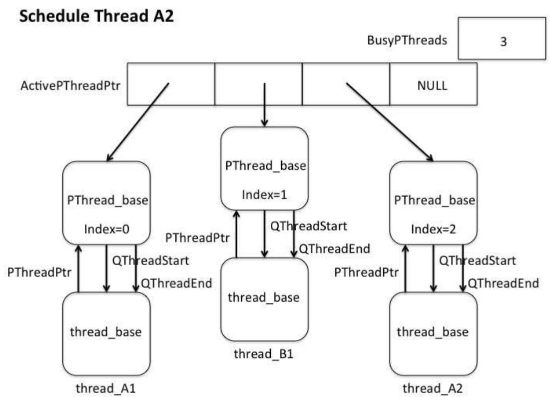
Figure 15: Simulation Process of HybridThreads (Part 2)



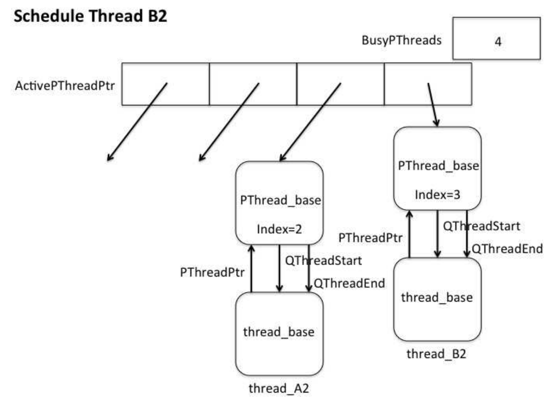
(13)



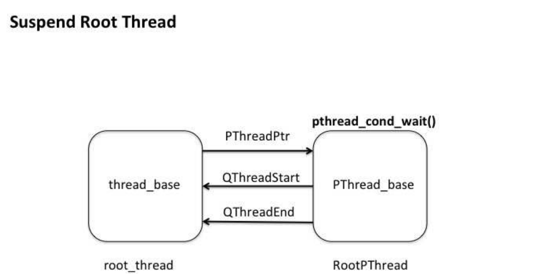
(14)



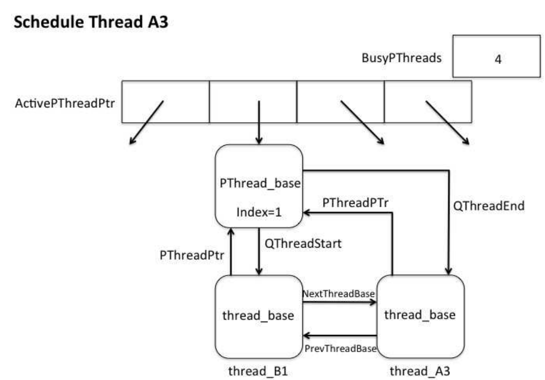
(15)



(16)

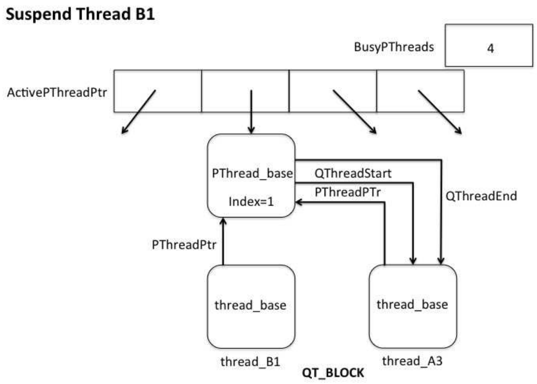


(17)

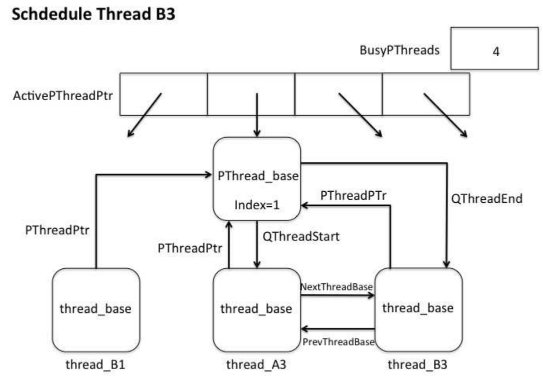


(18)

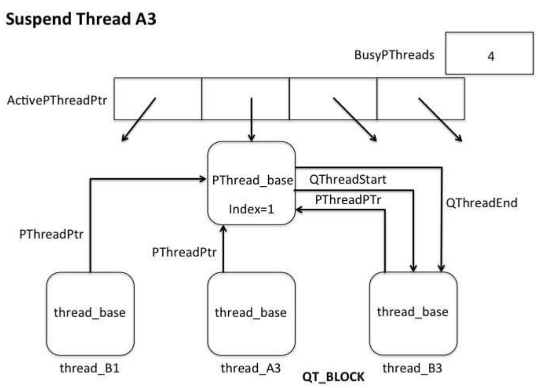
Figure 16: Simulation Process of HybridThreads (Part 3)



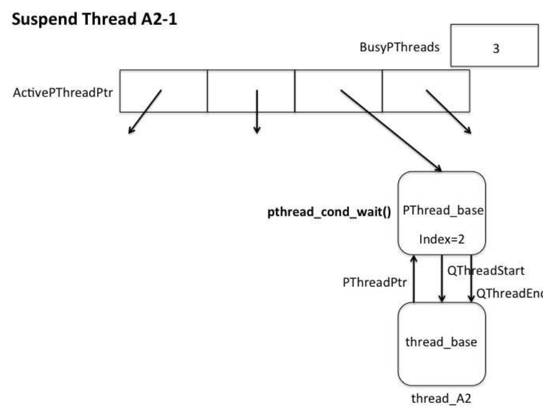
(19)



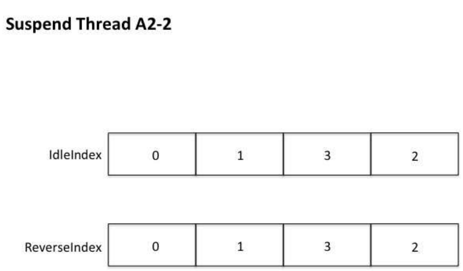
(20)



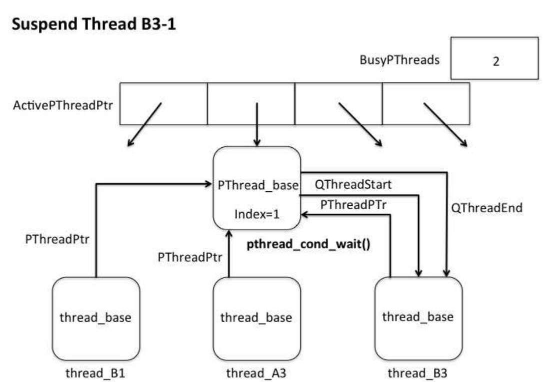
(21)



(22)

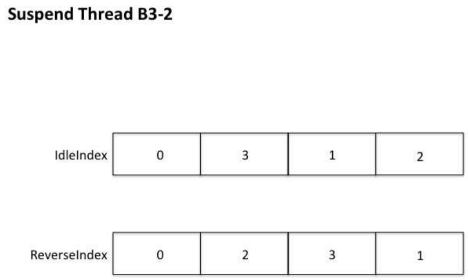


(23)

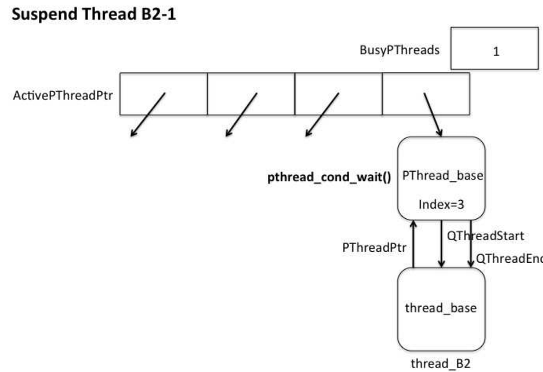


(24)

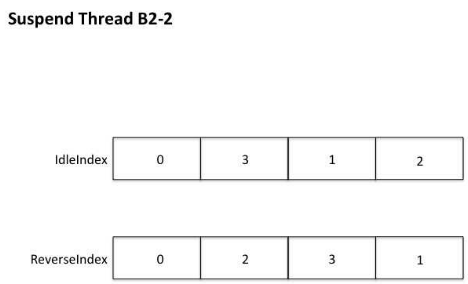
Figure 17: Simulation Process of HybridThreads (Part 4)



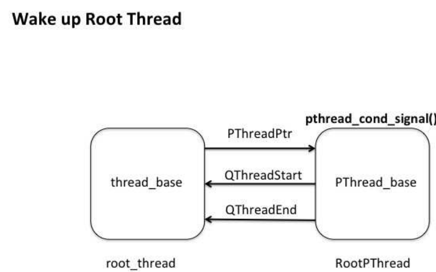
(25)



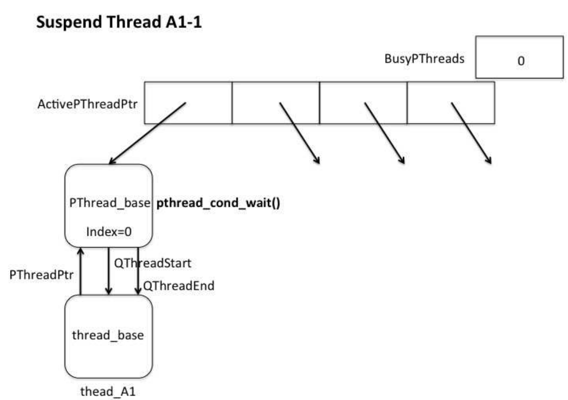
(26)



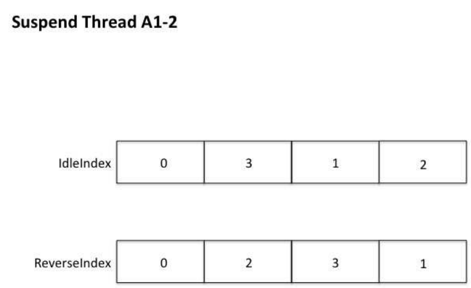
(27)



(28)

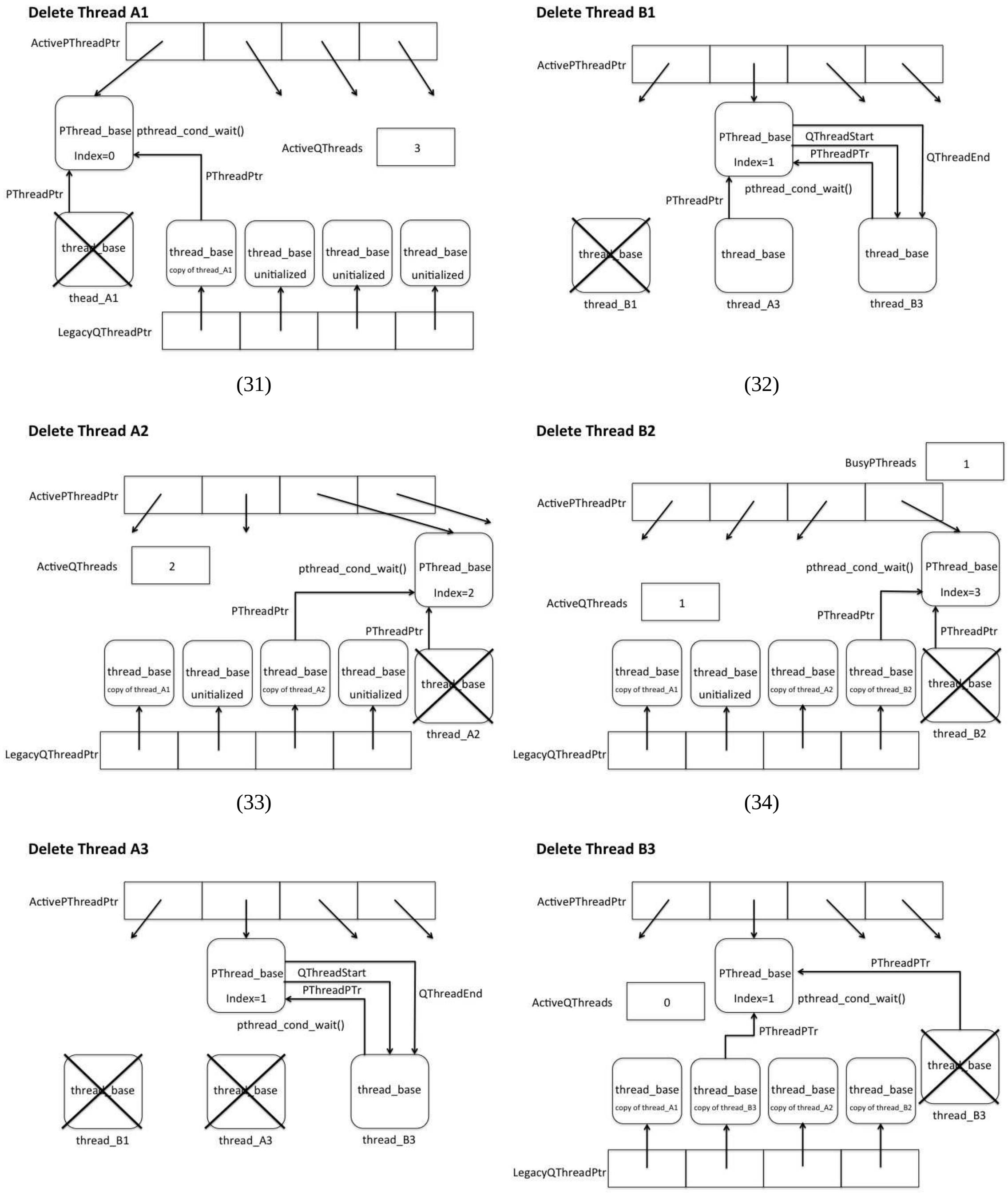


(29)



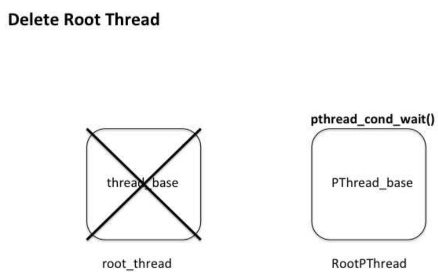
(30)

Figure 18: Simulation Process of HybridThreads (Part 5)

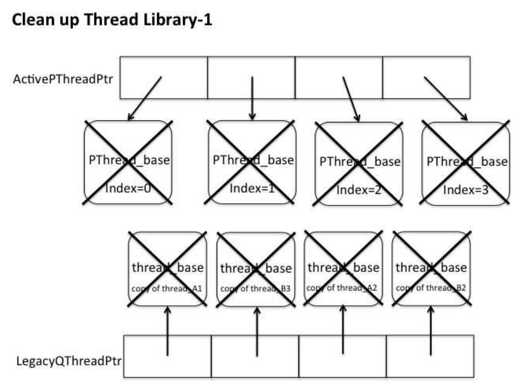


(31) (32) (33) (34) (35) (36)

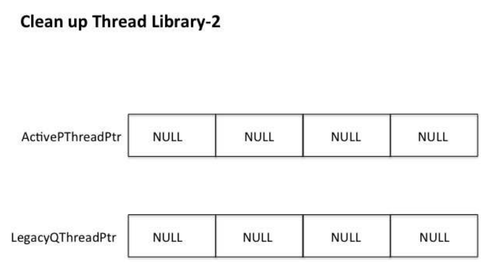
Figure 19: Simulation Process of HybridThreads (Part 6)



(37)



(38)



(39)

Figure 20: Simulation Process of HybridThreads (Part 7)

2.4 Implementations of the Programming Interface in HybridThreads

The whole HybridThreads library is implemented in C++ and assembly language (the machine-dependent code in QuickThreads package). As the HybridThreads library has the same programming interface as the regular parallel PosixThreads library, it can be easily inserted into the existing SpecC simulator. The basic interface in the SpecC parallel simulator consists of functions to create, delete, suspend and wake up threads, which we will describe in detail in the following sections.

2.4.1 Thread Creation

```
void _specc::thread_base::ThreadCreate(thread_fct Function, thread_arg Arg)
```

The HybridThreads library will first allocate a block of memory space for the stack of the new thread. If the current number of kernel-level threads is less than that of available cores, a new Posix thread is created. No matter a Posix thread is created or not, the newly allocated thread's stack is initialized with the new thread function and arguments (*Function* & *Arg*). Figure 21 shows the control flow of the *ThreadCreate* function.

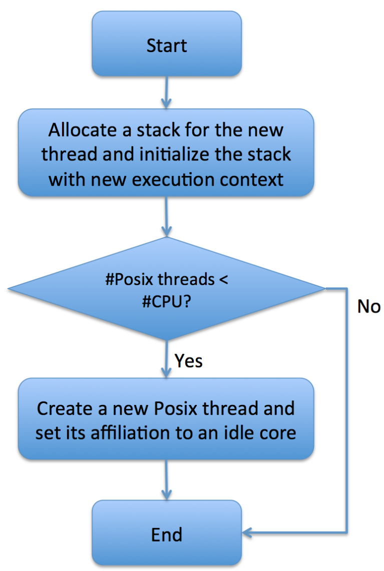


Figure 21: Control Flow of ThreadCreate

2.4.2 Thread Deletion

```
void _specc::thread_base::ThreadDelete(void)
```

In the *ThreadDelete* function, it will only delete the user-level thread. If the underlying kernel-level thread is suspended in the execution context of this user-level thread, we need to save the context onto some global structures and free up that context only after the kernel-level thread is waken up or the whole program is terminated. Otherwise, we would simply deallocate the stack and delete the user-level thread.

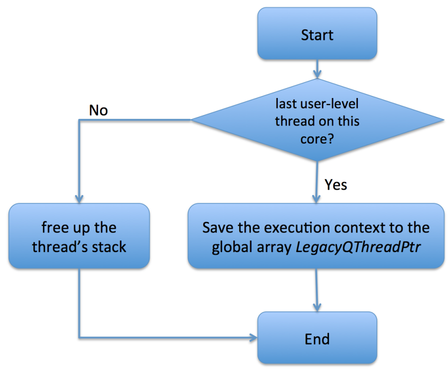


Figure 22: Control Flow of ThreadDelete

2.4.3 Thread Suspension

```
void _specc::thread_base::Wait(void)
```

The *Wait* function in the HybridThreads library is used to switch to a new user-level thread. If no more userspace thread is available on the current Posix thread, the kernel-level thread will be suspended by the conditional variable. In the other case, it will simply save the current context to the stack of the old thread and continue executing another.

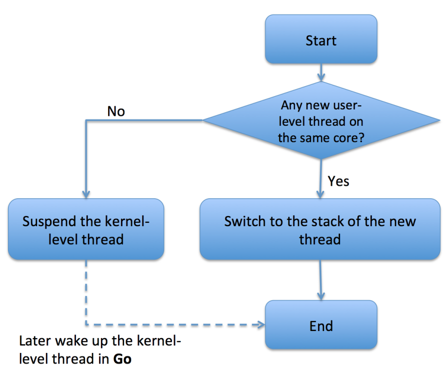


Figure 23: Control Flow of Wait

2.4.4 Thread Wakeup

```
void _specc::thread_base::Go(void)
```

When there are some idle cores or kernel-level threads, HybridThreads library will assign the new user-level thread to these cores. In case that all the kernel-level threads are busy, the new userspace thread will be run on the current core to avoid race conditions and protect consistency of sharing resources.

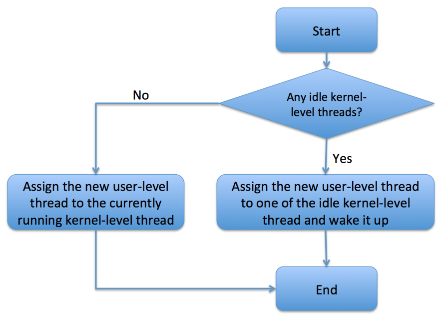


Figure 24: Control Flow of Go

3 Performance Evaluation of HybridThreads

To demonstrate the performance of the new thread library, we will first utilize two parallel benchmarks to test different aspects of the thread library, and then make use of three real-world embedded applications to evaluate the simulation speed of the HybridThreads library. All the benchmarks and applications 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 25 and 26 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 [14].

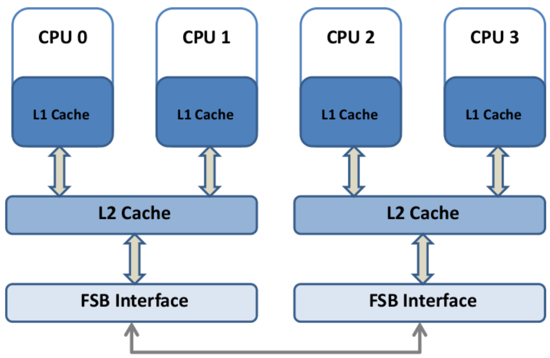


Figure 25: Intel Core 2 Quad architecture, Q9650 (mu) [14]

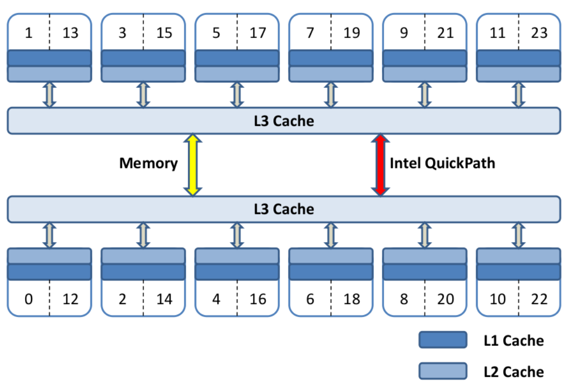


Figure 26: Intel Xeon architecture, X5650 (xi) [14]

3.1 Simulation Results for Parallel Benchmarks

For the HybridThreads library, we use a Producer-Consumer example to measure the performance of context switching and a highly parallel benchmark which calculates Fibonacci number in each thread (named Fibo20, Fibonacci number calculation with a maximum parallelism of 20). The Fibo20 benchmark is designed to test the parallel computation performance of the regular Posix-Threads and new HybridThreads simulators. Hence, Fibo20 has only pure computation but no communication among parallel threads.

3.1.1 Producer-Consumer Model

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
	233.21s	110.85s	413.37s	83.00%	Parallel PosixThreads
	210.56s	75.98s	387.01s	74.00%	HybridThreads
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
	146.75s	282.02s	429.51s	99.00%	Parallel PosixThreads
	262.36s	235.17s	510.12s	97.00%	HybridThreads

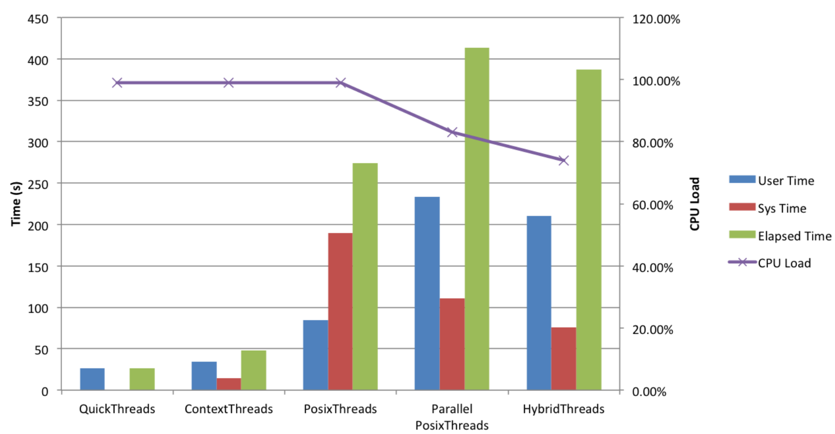


Figure 27: Simulation Results for Producer-Consumer Model on mu

The first parallel benchmark, Producer-Consumer model, is a simple example which features intensive context switching. During the whole simulation, the program will create only three threads:

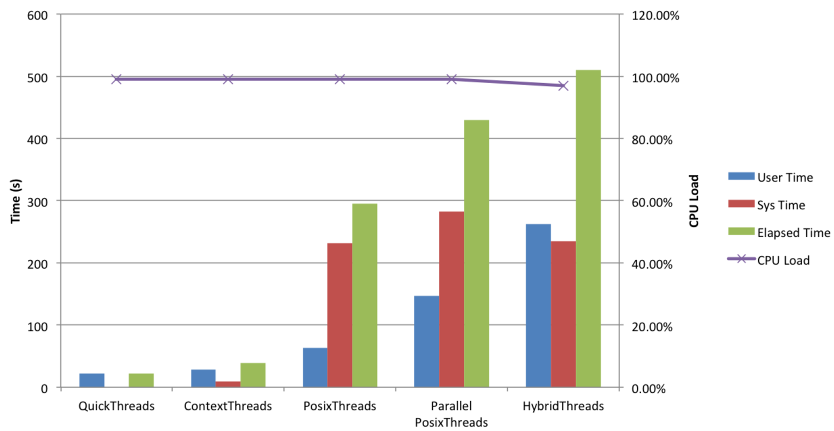


Figure 28: Simulation Results for Producer-Consumer Model on xi

a Producer, a Consumer and a Monitor. The Producer instance is repeatedly sending data to the Consumer through a double-handshake channel. This communication is wrapped up in a large 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 [15].

Figure 27 to 28 list the simulation results for the Producer-Consumer benchmark. As the Producer-Consumer model has a limited amount of parallelism (only one thread running at any time), the overhead of parallel simulation burdens the system and both the parallel thread libraries are slower than the sequential ones. Between the two parallel simulators, they have similar performance on the models which have intensive context switching. Specifically, HybridThreads library is worse than PosixThreads on **xi** as it has a higher user-level overhead.

Table 2: Simulation Results for Prod-Cons Model (_SPECC_NUM_SIMCPUS=1)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	26.85s	0s	26.87s	99.00%	QuickThreads
	34.11s	14.22s	48.35s	99.00%	ContextThreads
	84.8s	189.48s	274.38s	99.00%	PosixThreads
	344.8s	198.43s	736.14s	73.00%	Parallel PosixThreads
	70.26s	8.36s	78.65s	99.00%	HybridThreads
xi	22.08s	0s	22.14s	99.00%	QuickThreads
	28.75s	9.79s	38.65s	99.00%	ContextThreads
	63.6s	231.25s	295.66s	99.00%	PosixThreads
	218.95s	435.84s	725.8s	90.00%	Parallel PosixThreads
	46.5s	7.75s	54.45s	99.00%	HybridThreads

In the SpecC parallel simulator, we can reconfigure the number of simulation cores (_SPECC_

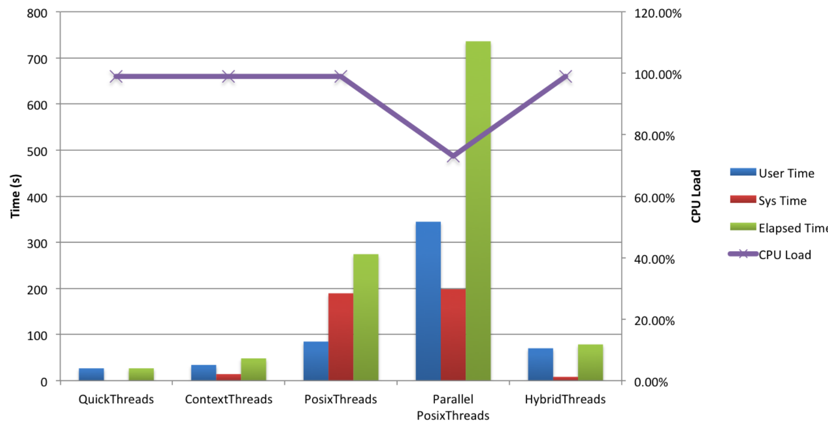


Figure 29: Simulation Results for Prod-Cons Model on mu (_SPECC_NUM_SIMCPUS=1)

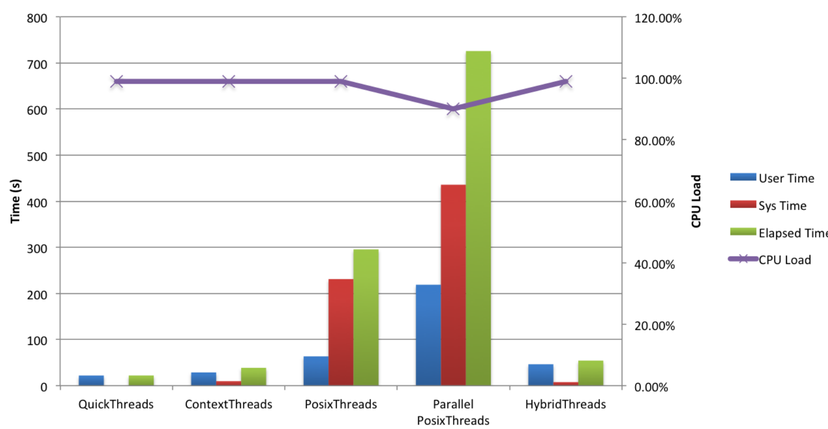


Figure 30: Simulation Results for Prod-Cons Model on xi (_SPECC_NUM_SIMCPUS=1)

NUM_SIMCPUS) to adjust the performance. For the Producer-Consumer Model, when we only use one core to simulate the example, the performance of HybridThreads library improves a lot. On Figure 29 and 30 (also in Table 2), HybridThreads has a close performance to QuickThreads as the context switching is quite similar and only runs in the user level. For the parallel PosixThreads library, it is much worse as it still switches to kernel level in thread scheduling.

3.1.2 Fibo20 Model

Appendix A.1 lists the code of Fibo20. In each thread of Fibo20, it calculates the same Fibonacci series in recursion and there are no data dependencies between different threads. Recall that a Fibonacci number is the sum of the previous two Fibonacci numbers and the recursive calculation

Table 3: Simulation Results for Fibo20 Model

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	234.1s	0.26s	234.45s	99.00%	QuickThreads
	227.19s	0.29s	227.55s	99.00%	ContextThreads
	230.97s	1.81s	233.04s	99.00%	PosixThreads
	266.11s	4.36s	95.93s	281.00%	Parallel PosixThreads
	233.3s	1.09s	60.04s	390.00%	HybridThreads
xi	160.04s	0.17s	160.69s	99.00%	QuickThreads
	160.25s	0.18s	160.92s	99.00%	ContextThreads
	160.59s	1.65s	162.89s	99.00%	PosixThreads
	442.41s	7.54s	58.05s	775.00%	Parallel PosixThreads
	396.05s	2.38s	23.92s	1665.00%	HybridThreads

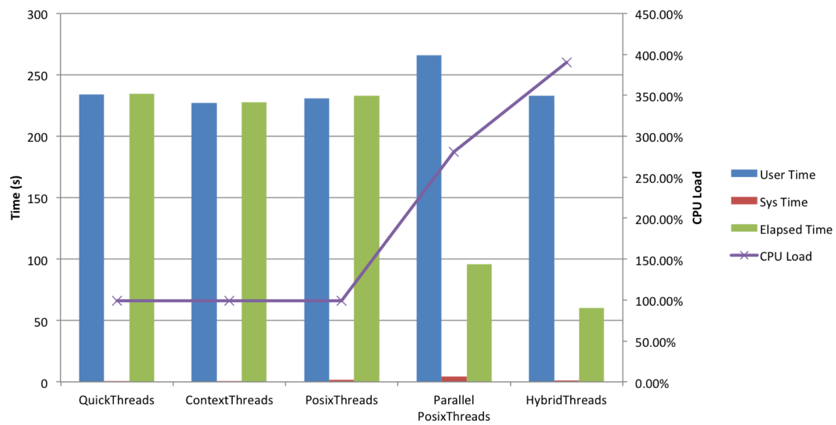


Figure 31: Simulation Results for Fibo20 Model on mu

of a large Fibonacci number is very computation intensive. Plus that the parallel threads have no inter-thread communication, this benchmark extremely favors the parallel simulators. As indicated in the source code, a maximum parallelism of 20 is available in this model.

As expected for the highly parallel Fibo20 benchmark, the performance of the parallel simulators improves tremendously. The HybridThreads library has a speedup of 3.7 on **mu** and 7.3 on **xi** over the sequential QuickThreads library. Compared with the regular PosixThreads library, the new thread library is about 33% faster on **mu** and 61% on **xi** machine.

As the Fibo20 example has a large amount of explicit parallelism, we draw a scalability figure of Parallel PosixThreads and HybridThreads on Figure 33. From the graph, it is easily seen that the new HybridThreads library always has a smaller elapsed time than PosixThreads, with the number of simulation cores varying from 1 to 30. Also, in most cases, the speedups of HybridThreads are higher than PosixThreads. Even though, the new thread library has one drawback as well: there are some bumps in both the elapsed time and relative speedup of HybridThreads, which shows that

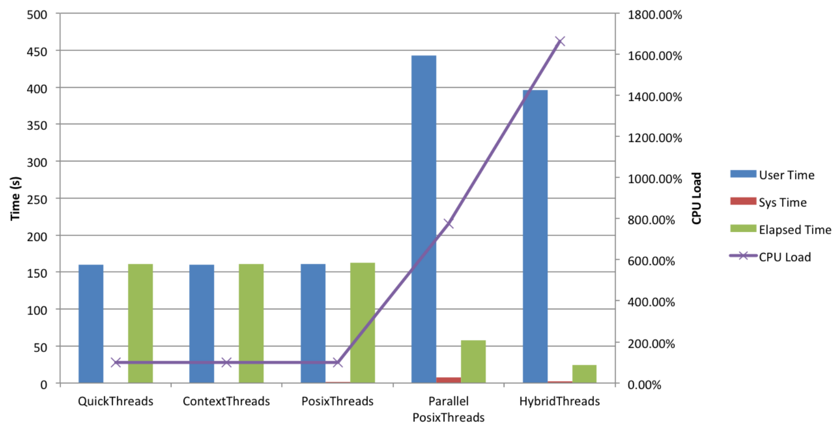


Figure 32: Simulation Results for Fibo20 Model on xi

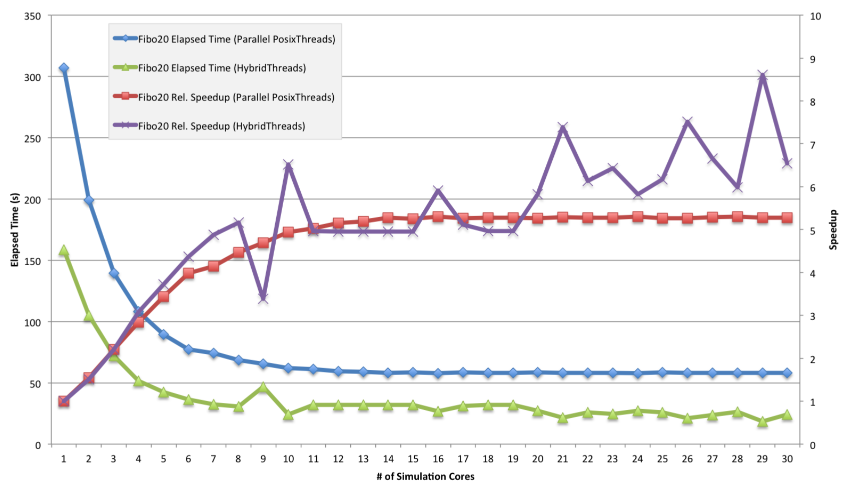


Figure 33: Scalability Figure for Fibo20 Model on xi

HybridThreads library is less consistent than PosixThreads. One possible reason for these ”ups and downs” is that we map threads to cores straightforwardly (one after one in a sequence) in HybridThreads and these threads will be affected by other processes in Linux OS to a larger extent than PosixThreads. Later in the future we will make improvement on this issue and map threads in a more wise and efficient manner. Table 4 and 5 list the specific statistics about the scalability figure of PosixThreads and HybridThreads libraries.

Table 4: Scalability Figure of Parallel PosixThreads for Fibo20 Model on xi

# of Cores	Usr Time	Sys Time	Elapsed Time	CPU Load	Speedup
1	298.7s	5.48s	307.1s	99.00%	1
2	372.54s	6.9s	198.97s	190.00%	1.54
3	362.05s	6.44s	139.3s	264.00%	2.2
4	363.64s	6.73s	107.99s	342.00%	2.84
5	355.43s	6.35s	89.3s	405.00%	3.44
6	346.92s	6.41s	77.18s	457.00%	3.98
7	370.32s	6.62s	74.09s	508.00%	4.14
8	385.92s	6.62s	68.71s	571.00%	4.47
9	396.81s	6.42s	65.45s	616.00%	4.69
10	414.1s	6.59s	62.14s	676.00%	4.94
11	423.53s	6.51s	61.05s	704.00%	5.03
12	428.17s	7.01s	59.54s	730.00%	5.16
13	428.14s	6.7s	59.19s	734.00%	5.19
14	431.96s	7.04s	58.2s	754.00%	5.28
15	435.83s	7.23s	58.45s	758.00%	5.25
16	435.97s	7.36s	57.88s	765.00%	5.31
17	437.28s	7.33s	58.36s	761.00%	5.26
18	436.79s	7.49s	58.18s	763.00%	5.28
19	437.47s	7.7s	58.16s	765.00%	5.28
20	441.46s	6.94s	58.36s	768.00%	5.26
21	441.56s	7.24s	58.05s	773.00%	5.29
22	443.01s	7.18s	58.24s	772.00%	5.27
23	441.21s	7.49s	58.19s	771.00%	5.28
24	440.49s	7.24s	57.89s	773.00%	5.3
25	441.91s	7.18s	58.36s	769.00%	5.26
26	442.07s	7.14s	58.31s	770.00%	5.27
27	441.07s	7.44s	58.01s	773.00%	5.29
28	440.59s	7.24s	57.92s	773.00%	5.3
29	442.16s	7.19s	58.22s	771.00%	5.27
30	441.73s	7.16s	58.21s	771.00%	5.28

Table 5: Scalability Figure of HybridThreads for Fibo20 Model on xi

# of Cores	Usr Time	Sys Time	Elapsed Time	CPU Load	Speedup
1	157.2s	0.72s	158.59s	99.00%	1
2	203.69s	1.06s	105.06s	194.00%	1.51
3	204.54s	1.03s	71.84s	286.00%	2.21
4	195.04s	0.99s	51.639s	379.00%	3.07
5	198.27s	1.05s	42.62s	467.00%	3.72
6	193.5s	1.08s	36.33s	535.00%	4.37
7	199.47s	1.02s	32.47s	617.00%	4.88
8	220.33s	1.46s	30.68s	722.00%	5.17
9	300.18s	1.93s	46.83s	645.00%	3.39
10	221.87s	1.57s	24.32s	918.00%	6.52
11	300.5s	2.14s	31.97s	946.00%	4.96
12	300.85s	2.25s	32.01s	946.00%	4.95
13	321.35s	2.26s	32.04s	1009.00%	4.95
14	341.76s	2.28s	32.05s	1073.00%	4.95
15	362.12s	2.4s	32.05s	1137.00%	4.95
16	327.13s	2.06s	26.84s	1226.00%	5.91
17	395.13s	2.49s	30.98s	1283.00%	5.12
18	422.28s	2.55s	31.94s	1329.00%	4.97
19	442.29s	2.7s	31.95s	1392.00%	4.96
20	453.97s	2.6s	27.22s	1677.00%	5.83
21	351.13s	2.26s	21.47s	1645.00%	7.39
22	430.09s	2.43s	25.83s	1674.00%	6.14
23	410.55s	2.43s	24.66s	1674.00%	6.43
24	457.98s	2.61s	27.24s	1690.00%	5.82
25	426.65s	2.47s	25.7s	1669.00%	6.17
26	345.44s	2.22s	21.12s	1645.00%	7.51
27	399.37s	2.44s	23.83s	1686.00%	6.66
28	441.94s	2.59s	26.51s	1676.00%	5.98
29	304.37s	2.01s	18.42s	1662.00%	8.61
30	403.11s	2.45s	24.22s	1674.00%	6.55

3.2 Simulation Results for Embedded Applications

Next we will demonstrate the simulation performance of the new HybridThreads library for three actual embedded system applications.

3.2.1 JPEG Image Encoder

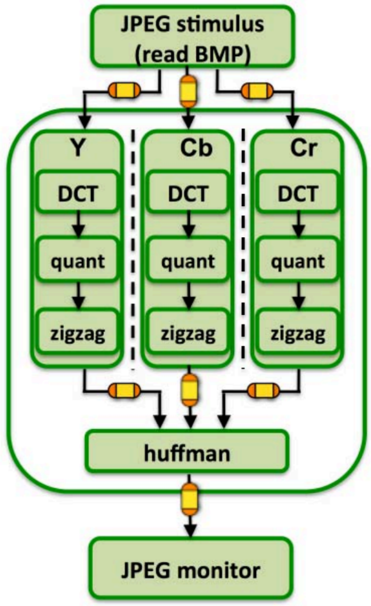


Figure 34: Block Diagram for JPEG Encoder [5]

Table 6: Simulation Results for JPEG Encoder

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	2.23s	0.05s	2.29s	99.00%	QuickThreads
	2.16s	0.1s	2.26s	99.00%	ContextThreads
	2.27s	0.36s	2.64s	99.00%	PosixThreads
	3.78s	0.76s	3.35s	135.00%	Parallel PosixThreads
	3.98s	0.65s	3.4s	136.00%	HybridThreads
xi	1.99s	0.03s	2.03s	99.00%	QuickThreads
	2s	0.06s	2.07s	99.00%	ContextThreads
	2.27s	0.36s	2.65s	99.00%	PosixThreads
	4.5s	1.43s	3.92s	151.00%	Parallel PosixThreads
	4.43s	1.12s	3.8s	146.00%	HybridThreads

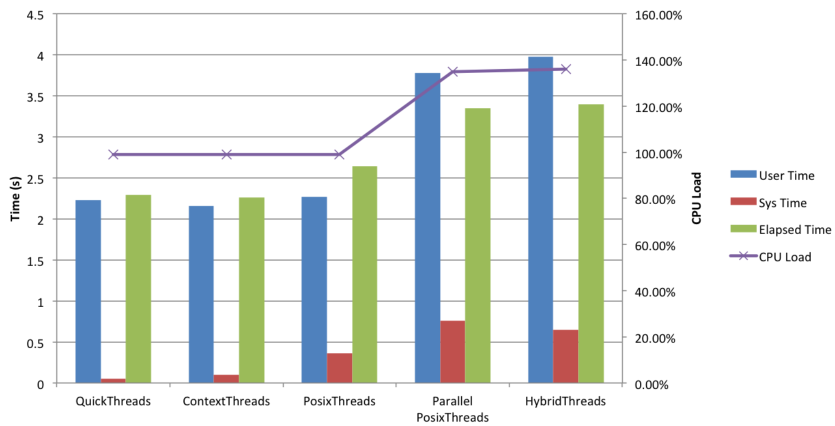


Figure 35: Simulation Results for JPEG Encoder on mu

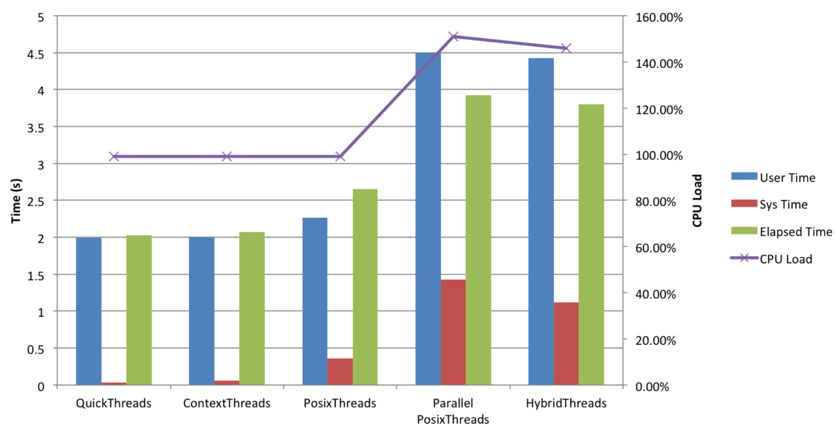


Figure 36: Simulation Results for JPEG Encoder on xi

The JPEG image encoder is a widely used application in embedded systems. After reading a block of image from a BMP file, it will first separate it into three color components. Then the program processes the color components through DCT, Quantization, and Zigzag in parallel. Finally, these three components are encoded in Huffman coding algorithm and combined into a single image [4]. Figure 34 shows the block diagram of the JPEG encoder.

As the available parallelism in JPEG encoder is very low (maximal 3 parallel threads and followed by a significant sequential part), the performance of the two parallel thread libraries is inferior to the sequential ones as shown in Figure 35 and 36. Between the two parallel thread libraries, HybridThreads is slightly better than PosixThreads on **xi** but worse on **mu** for the higher user time. As we can modify the number of simulation cores to reconfigure the parallel thread libraries, Figure 37 and 38 show some more simulation results in the case that the parallel PosixThreads and

HybridThreads simulators are running on one core (or in other words, running in "sequential" mode). Under such circumstances, the HybridThreads library has a similar performance as the QuickThreads while the parallel PosixThreads simulator is burdened by the heavyweight load of system overhead.

Table 7: Simulation Results for JPEG Encoder (_SPECC_NUM_SIMCPUS=1)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	2.23s	0.05s	2.29s	99.00%	QuickThreads
	2.16s	0.1s	2.26s	99.00%	ContextThreads
	2.27s	0.36s	2.64s	99.00%	PosixThreads
	3.78s	0.5s	4.53s	94.00%	Parallel PosixThreads
	2.34s	0.04s	2.38s	99.00%	HybridThreads
xi	1.99s	0.03s	2.03s	99.00%	QuickThreads
	2s	0.06s	2.07s	99.00%	ContextThreads
	2.27s	0.36s	2.65s	99.00%	PosixThreads
	4.43s	1.17s	5.81s	96.00%	Parallel PosixThreads
	2.27s	0.04s	2.32s	99.00%	HybridThreads

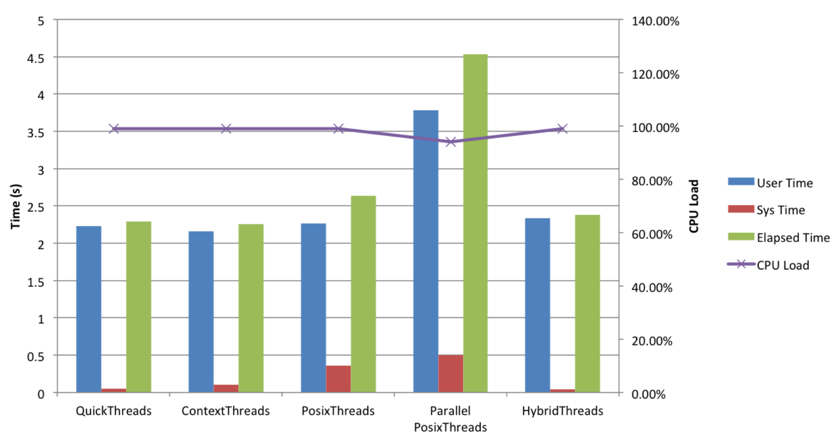


Figure 37: Simulation Results for JPEG Encoder on mu (_SPECC_NUM_SIMCPUS=1)

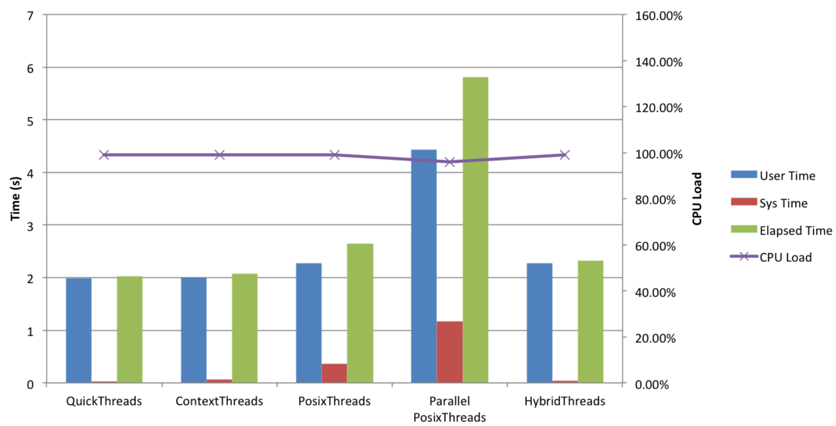


Figure 38: Simulation Results for JPEG Encoder on xi (_SPECC_NUM_SIMCPUS=1)

3.2.2 H.264 AVC Decoder with Parallel Slice Decoding

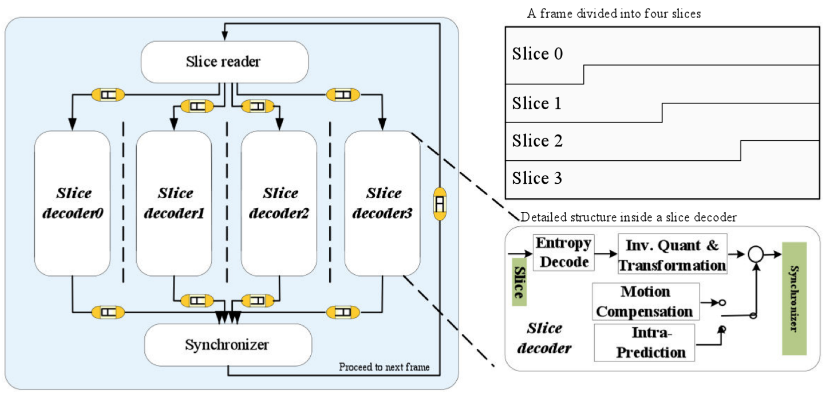


Figure 39: Block Diagram of H.264 AVC Decoder [9]

Figure 39 shows the block diagram of the H.264 Advanced Video Coding (AVC) decoder. The design model begins with reading a new frame from the input stream. The frames are then split into four slices and decoded in parallel [12]. Remarkably, there are no data dependencies between these four slices so that these four slices are fully parallel. After all slices are done, a synchronizer block filters the decoded frame to complete the decoding. Figure 40 and 41 list the simulation results for the H.264 Decoder on two machines.

Table 8: Simulation Results for H.264 Decoder

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	175.7s	2.65s	180.96s	98.00%	QuickThreads
	172.82s	2.75s	176.64s	99.00%	ContextThreads
	174.82s	3.11s	178.94s	99.00%	PosixThreads
	1968s	5.21s	127.54s	157.00%	Parallel PosixThreads
	194.2s	4.62s	124.01s	160.00%	HybridThreads
xi	162.82s	2.18s	167.13s	98.00%	QuickThreads
	163.24s	2.57s	167.23s	99.00%	ContextThreads
	164.58s	3.42s	169.99s	98.00%	PosixThreads
	321.91s	5.8s	209.85s	156.00%	Parallel PosixThreads
	320s	5.52s	207.94s	156.00%	HybridThreads

On **mu** machine, both the parallel simulators have similar performance, and achieve a speedup of 1.45 over the sequential QuickThreads simulator. However, as the **xi** machine has many more cores than mu (24 vs. 4) and the user level overhead is much higher on **xi**, the parallel simulators on **xi** perform poorly with regard to the sequential simulators. But after configuring the number of simulation cores to be one, the new HybridThreads have a performance very close to the sequential

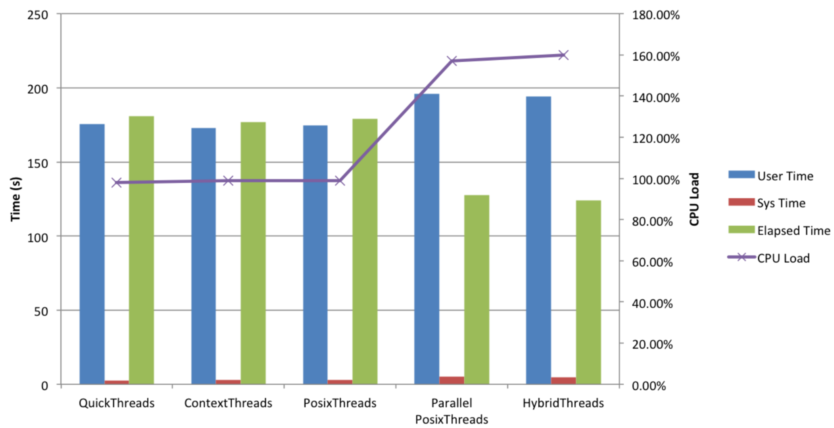


Figure 40: Simulation Results for H.264 Decoder on mu

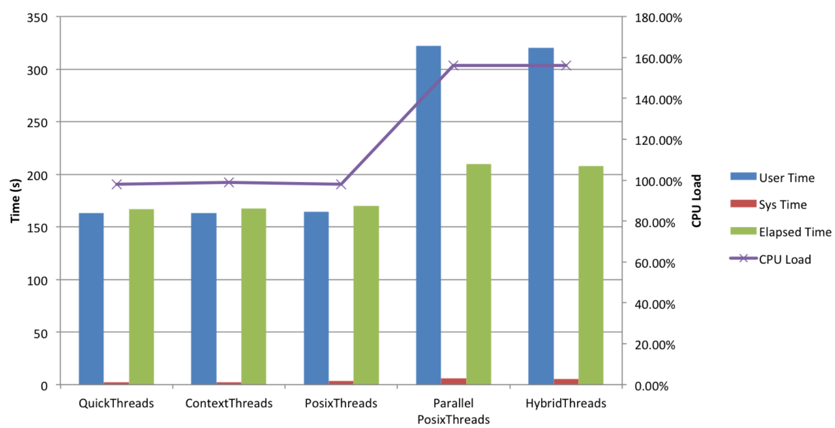


Figure 41: Simulation Results for H.264 Decoder on xi

QuickThreads library, even though that the regular parallel PosixThreads library performs worse than before.

Table 9: Simulation Results for H.264 Decoder (_SPECC_NUM_SIMCPUS=1)

Hostname	**Usr Time**	**Sys Time**	**Elapsed Time**	**CPU Load**	**Thread Library**
xi	162.82s	2.18s	167.13s	98.00%	QuickThreads
	163.24s	2.57s	167.23s	99.00%	ContextThreads
	164.58s	3.42s	169.99s	98.00%	PosixThreads
	311.15s	5.74s	319.61s	99.00%	Parallel PosixThreads
	164.04s	3.02s	169.94s	98.00%	HybridThreads

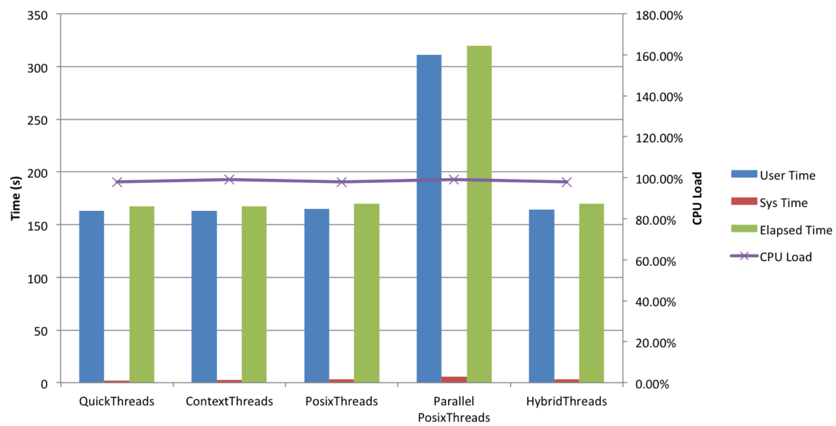


Figure 42: Simulation Results for H.264 Decoder on xi (_SPECC_NUM_SIMCPUS=1)

3.2.3 H.264 AVC Encoder

As a third real-world embedded system application, we have the H.264 AVC Encoder which is converted from the reference C code and has a maximum of 30 frames processed in parallel. The parallel parts happen during the luminance and chrominance pixel residual coding and motion vector search for multiple reference frames [3]. However as there are heavy dependencies among the current block and the left, up and up-left blocks in the image, the available parallelism is quite limited in this application. The simulation results of the five thread libraries are shown in Figure 43 and 44.

Table 10: Simulation Results for H.264 Encoder

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	2368.59s	29.86s	2399.35s	99.00%	QuickThreads
	2356.86s	13.19s	2370.94s	99.00%	ContextThreads
	2364.02s	57.74s	2428.41s	99.00%	PosixThreads
	2966.02s	143.11s	1823.64s	170.00%	Parallel PosixThreads
	2779.07s	51.75s	1546.93s	182.00%	HybridThreads
xi	2268.89s	19.83s	2294.93s	99.00%	QuickThreads
	2251.5s	9.61s	2267.24s	99.00%	ContextThreads
	2261.52s	53.54s	2326.73s	99.00%	PosixThreads
	5820.55s	203.17s	1978.13s	304.00%	Parallel PosixThreads
	6473.3s	131.28s	1812.12s	364.00%	HybridThreads

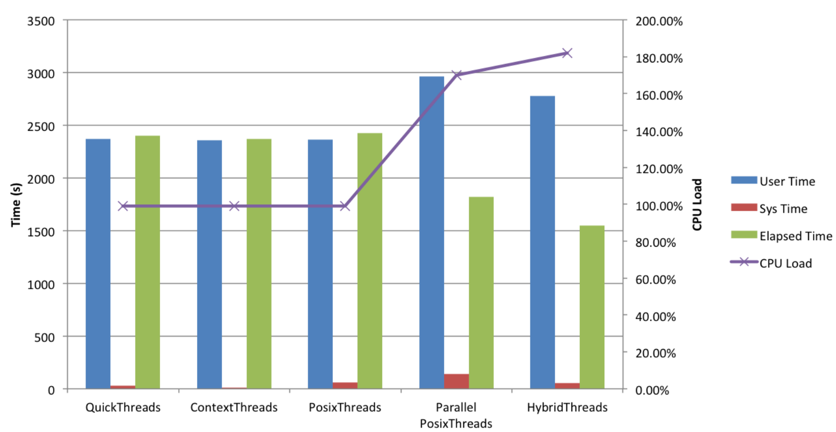


Figure 43: Simulation Results for H.264 Encoder on mu

Analyzing Figure 43 we can conclude that the parallel HybridThreads achieve a performance speedup of 1.31 over the sequential QuickThreads library and 1.24 over the parallel PosixThreads library on **mu** machine. However, on **xi** machine, the performance of the HybridThreads library is

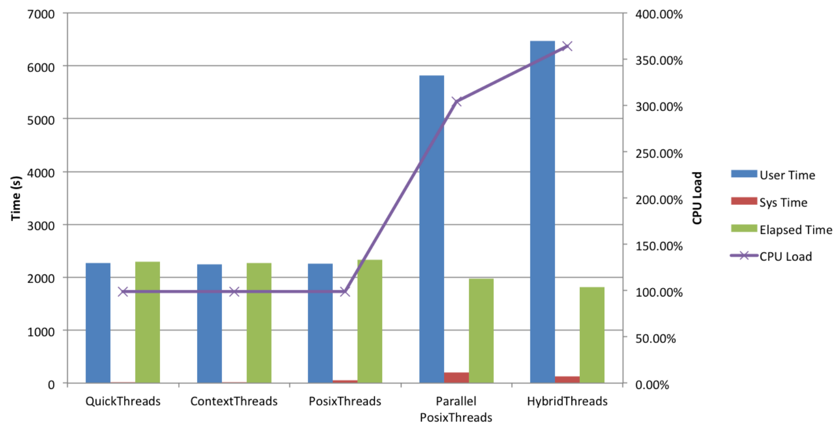


Figure 44: Simulation Results for H.264 Encoder on xi

very ”unstable” as shown on Figure 45. Notice that different runs of the same simulation result in very different measured execution times.

Recall the processor architecture of **xi** as shown in Figure 26, after configuring all the user threads to run on each physical core individually (Figure 46), on one side (Figure 47) and on each physical core of one side individually (Figure 48), we can conclude that the performance inconsistency of the HybridThreads library originates from the asymmetrical communication overhead among different cores and the hyperthreading features. Clearly, it becomes necessary in future work to address this ”unstability” by analyzing the inter-thread communication and mapping them to the CPU cores such that communication and synchronization are minimized. For the ”stable” HybridThreads (_SPECC_NUM_SIMCPUS=6), it has a similar performance as other thread libraries on Figure 49. When the HybridThreads library is running in sequential mode (Figure 50), it has an identical performance as QuickThreads.

Table 11: Simulation Results of HybridThreads on xi (_SPECC_NUM_SIMCPUS=24)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	6473.3s	131.28s	1812.12s	364.00%	HybridThreads
	4351.96s	58.84s	3730.8s	118.00%	
	4352.45s	59.31s	3730.74s	118.00%	
	4352.13s	59.75s	3736.23s	118.00%	
	4352.91s	61.48s	3575.33s	121.00%	

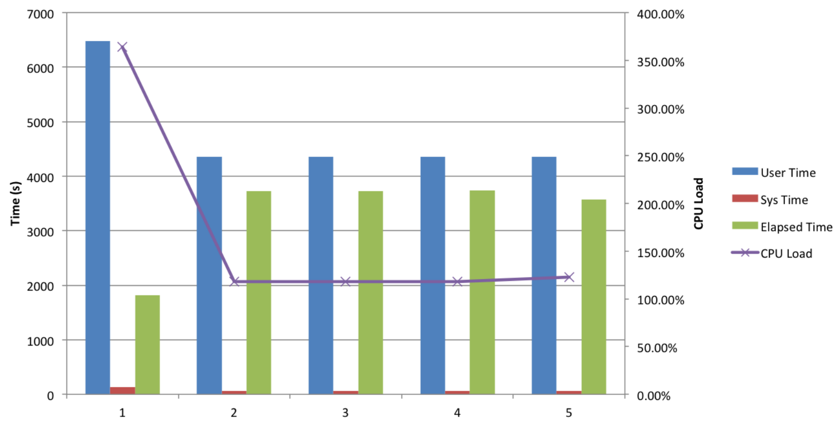


Figure 45: Simulation Results of HybridThreads on xi (_SPECC_NUM_SIMCPUS=24)

Table 12: Simulation Results of HybridThreads on xi (core 0, 1, 2, ..., 11)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	4358.95s	79.74s	1765.1s	251.00%	HybridThreads (core 0,1,2,...,11)
	4353.94s	62.33s	2706.05s	163.00%	
	4349.17s	60.08s	2874.86s	153.00%	
	4356.25s	80.65s	1765.22s	251.00%	
	4355.04s	80.53s	1763.93s	251.00%	

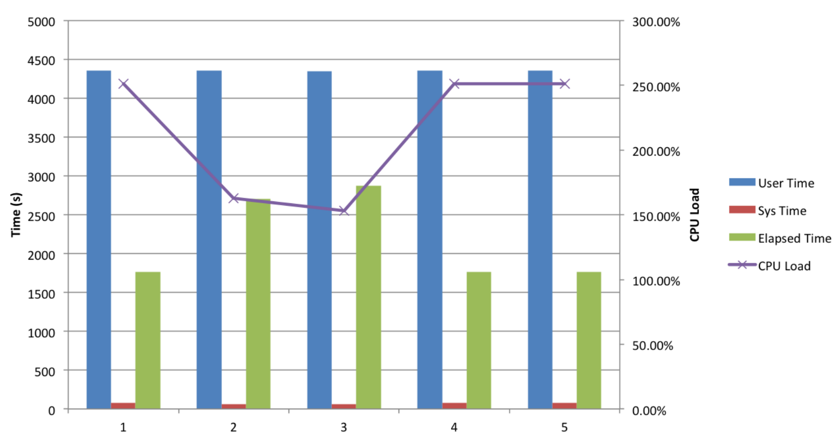


Figure 46: Simulation Results of HybridThreads on xi (core 0, 1, 2, ..., 11)

Table 13: Simulation Results of HybridThreads on xi (core 0, 2, 4, ..., 22)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	6723.55s	82.7s	2012.48s	338.00%	HybridThreads (core 0,2,4,...,22)
	4343.05s	54.83s	2890.15s	152.00%	
	6724.97s	83.21s	2012.87s	338.00%	
	6725.81s	82.26s	2013.54s	338.00%	
	4341.04s	55s	2888.65s	152.00%	

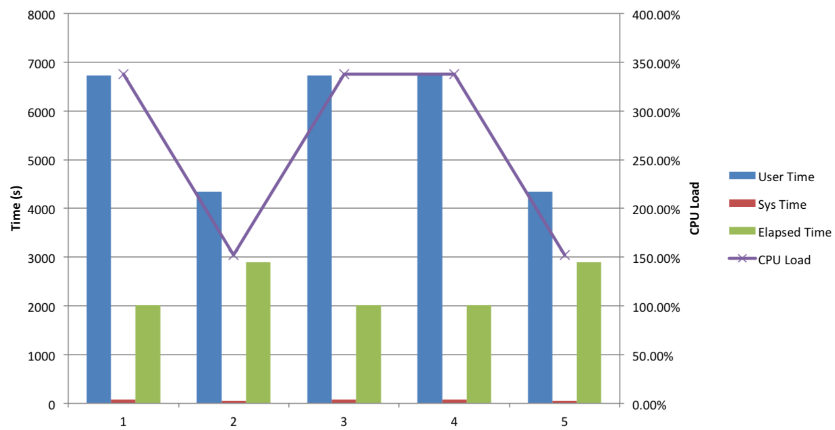


Figure 47: Simulation Results of HybridThreads on xi (core 0, 2, 4, ..., 22)

Table 14: Simulation Results of HybridThreads on xi (core 0, 2, 4, ..., 10)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	4342.36s	54.31s	2360.76s	186.00%	HybridThreads (core 0,2,4,,10)
	4335.22s	54.4s	2357.52s	186.00%	
	4343.75s	55.4s	2361.92s	186.00%	
	4342.16s	55.52s	2361.66s	186.00%	
	4339.63s	54.99s	2359.19s	186.00%	

Table 15: Simulation Results for H.264 Encoder on xi (_SPECC_NUM_SIMCPUS=6)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	2268.89s	19.83s	2294.93s	99.00%	QuickThreads
	2251.5s	9.61s	2267.24s	99.00%	ContextThreads
	2261.52s	53.54s	2326.73s	99.00%	PosixThreads
	4740.68s	183.64s	2238.9s	219.00%	Parallel PosixThreads
	4342.36s	54.31s	2360.76s	186.00%	HybridThreads

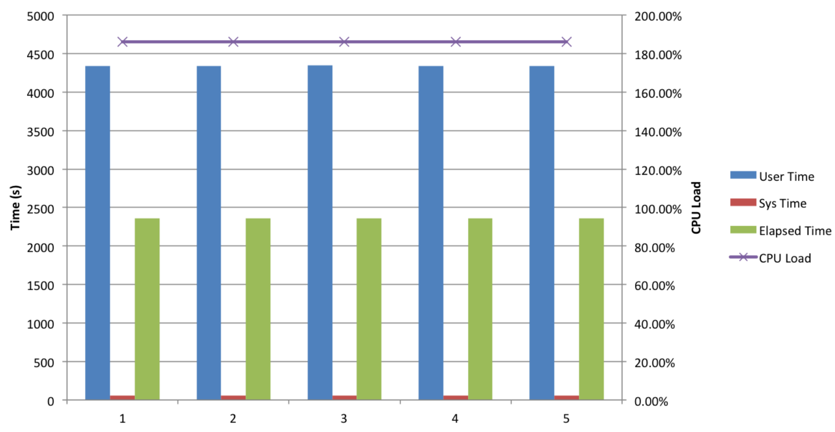


Figure 48: Simulation Results of HybridThreads on xi (core 0, 2, 4, ..., 10)

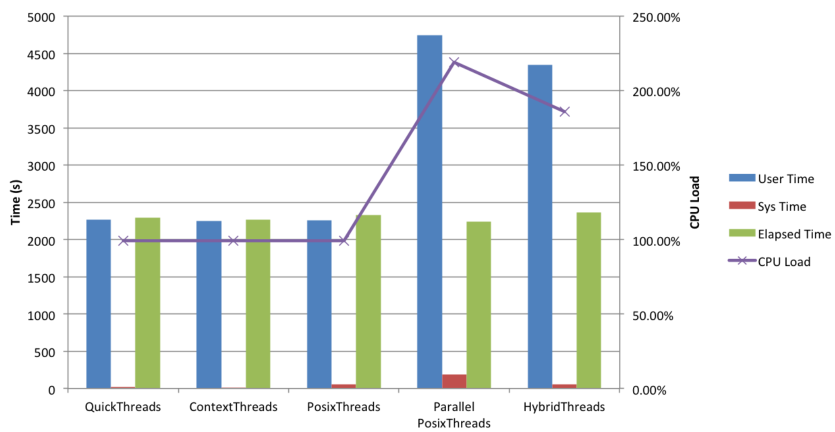


Figure 49: Simulation Results for H.264 Encoder on xi (_SPECC_NUM_SIMCPUS=6)

Table 16: Simulation Results for H.264 Encoder on xi (_SPECC_NUM_SIMCPUS=1)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	2268.89s	19.83s	2294.93s	99.00%	QuickThreads
	2251.5s	9.61s	2267.24s	99.00%	ContextThreads
	2261.52s	53.54s	2326.73s	99.00%	PosixThreads
	4318.02s	169.27s	4560.35s	98.00%	Parallel PosixThreads
	2265.02s	21.58s	2292.81s	99.00%	HybridThreads

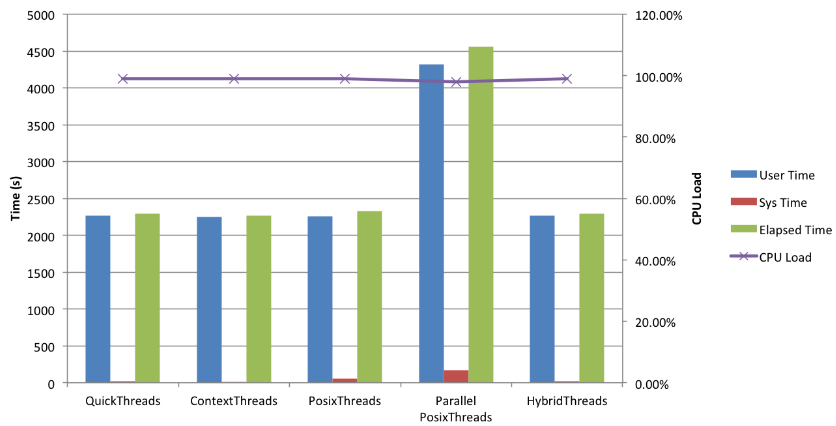


Figure 50: Simulation Results for H.264 Encoder on xi (_SPECC_NUM_SIMCPUS=1)

3.3 Time Profiling of the HybridThreads Library

On Figure 27, 28, 31, 32 and etc., it is shown that the user time and system time of parallel thread libraries (Parallel PosixThreads and HybridThreads) are always larger than those of sequential thread libraries (QuickThreads, ContextThreads and PosixThreads). In order to find out which part of the simulation program brings in the increments, we measured the locking time for the centralized mutex and running time in the application model. Table 17, 18, 19 and 20 show this timing information of HybridThreads library for all the benchmarks and examples on **mu** and **xi**.

Table 17: Time Profiling of HybridThreads on mu

Benchmark	Usr Time	Sys Time	Elapsed Time	Lock Time	App Time	% in Lock
Prod-Cons	220.1s	81s	405.75s	286.37s	39.97s	87.75%
Fibo20	232.42s	1.1s	59.75s	0.32s	231.75s	0.14%
JPEG Encoder	4.02s	0.64s	3.42s	1.18s	3.58s	24.75%
H.264 Decoder	193.94s	4.67s	123.9s	1.15s	198.03s	0.58%
H.264 Encoder	2777.62s	50.15s	1545.28s	11.56s	2774.65s	0.41%

Table 18: Time Profiling of HybridThreads on mu (_SPECC_NUM_SIMCPUS=1)

Benchmark	Usr Time	Sys Time	Elapsed Time	Lock Time	App Time	% in Lock
Prod-Cons	75.1s	8.02s	83.16s	4.22s	20.7s	16.92%
Fibo20	231.47s	1.15s	232.72s	0.08s	230.67s	0.03%
JPEG Encoder	2.35s	0.03s	2.39s	0.01s	2.25s	0.36%
H.264 Decoder	175.57s	4.06s	180.3s	0.01s	179.54s	0.01%
H.264 Encoder	2371.01s	31.15s	2403.1s	0.21s	2367.48s	0.01%

Table 17 and 18 list the time profiling of HybridThreads on **mu** machine. The first four columns show the user time, system time and elapsed time of all five examples measured by the Linux *time* command. The fifth column lists the locking time for the centralized mutex and sixth column shows the total time in the application model (sum of all parallel threads), which are measured by inserting timestamps before and after the locking functions and application models. The difference between Table 17 and 18 is that all examples in Table 17 are running with the maximum number of simulation cores while examples in 18 are on only one core. From these two tables, it shows that except for the Producer-Consumer (Prod-Cons in the tables) model which has intensive thread synchronization and almost no computation, all other benchmarks spend most of their time in the application model ("useful time" in simulation). The locking time for the centralized mutex ("wasted time") is always quite short and is less than 1% (shown in the seventh column) of the sum of useful time (time in application model) and wasted time (locking time for centralized mutex). For the JPEG Encoder example, as the computation is quite simple and there exists a lot of communication between threads, the percentage of locking time is about 25%. When the number of simulation cores is restricted to be only one, the locking time is decreased tremendously as the mutex in a sequential program is always available and can be locked immediately when one thread is trying to grab the lock.

Table 19: Time Profiling of HybridThreads on xi

Benchmark	Usr Time	Sys Time	Elapsed Time	Lock Time	App Time	% in Lock
Prod-Cons	231.61s	270.38s	514.34s	326.4s	56.99s	85.14%
Fibo20	419.14s	2.33s	25.53s	6.49s	418.02s	1.53%
JPEG Encoder	4.42s	1.17s	3.8s	1.59s	4s	28.44%
H.264 Decoder	319.99s	5.64s	208.49s	1.85s	324.35s	0.57%
H.264 Encoder	4354.77s	62.96s	3595.21s	68.09s	4344.98s	1.54%

Table 20: Time Profiling of HybridThreads on xi (_SPECC_NUM_SIMCPUS=1)

Benchmark	Usr Time	Sys Time	Elapsed Time	Lock Time	App Time	% in Lock
Prod-Cons	50.22s	8.37s	58.79s	3.19s	12.25s	20.66%
Fibo20	157.1s	0.7s	158.45s	0.21s	156.75s	0.13%
JPEG Encoder	2.06s	0.03s	2.11s	0.01s	2s	0.50%
H.264 Decoder	163.06s	2.87s	167.84s	0.01s	166.66s	0.01%
H.264 Encoder	2266.73s	21.5s	2294.44s	0.21s	2262.27s	0.01%

On Table 19 and 20, they show the same timing information on **xi** machine, which is also quite similar to those on **mu**. The locking time for the centralized mutex is only a small part of the total simulation time for most benchmarks. The large amount of time in application models might result from more page faults in the application data structs and heavier communication overhead among cores. To minimize these overheads, improvements should be made on the thread-to-core mapping and number of simulation cores in the future.

4 Conclusion

4.1 Summary

In this technical report we discussed the design and implementation of a new hybrid mode of thread library — HybridThreads. HybridThreads library integrates the kernel-level (PosixThreads) thread library with the user-level (QuickThreads) thread library to improve the simulation performance and guarantee multiprocessor access. The traditional kernel-level threads can run on multiple CPUs at the same time, but the system load of maintaining sharing resources and safe synchronization among different threads will burden the performance of the application. After building the user-level threads above kernel-level threads, the context switching among the user-level threads on the same kernel-level thread is completely managed by the userspace library. Only when there is synchronization between two different kernel-level threads, the thread library will call the in-kernel scheduler. In this way, the system overhead of thread manipulation is reduced and a performance speedup of more than 1.4 is achieved over the sequential thread library for some parallel system-level applications.

4.2 Lessons Learned

During the implementation of HybridThreads library, we learned more about the features of the user-level threads and kernel-level threads. The management of the user-level threads requires no kernel intervention and they are also invisible to the OS kernel. These properties of the user-level threads reduce the cost of manipulating user-level threads but also decide that user-level threads cannot run in parallel. On the other hand, kernel threads are created and scheduled by the operating system kernel. Thus, the kernel-level thread library will place a more balanced load on multiple processors but also bring to the application heavy load of synchronization among threads. A good trade-off to design the thread library is to combine the features of both the kernel-level threads library and user-level threads library.

From the simulation results listed in Section 3.1 and 3.2, we can conclude that the implemented HybridThreads library has the following advantages: when the application has a high level of parallelism and the sequential part is insignificant, the new thread library performs superior and even enables more cores to run simultaneously (as indicated in the CPU load in Table 31 and 32); the mapping of the user-level threads is totally managed by the user-level scheduler and thus the userspace application has a better control of the thread-to-core mapping; in the case that the parallelism is limited in the program, HybridThreads can be reconfigured easily by modifying the maximum number of simulation cores and achieve a higher performance close to QuickThreads. Of course the HybridThreads library brings in some user-level overhead for the parallel execution, but it is adjustable and can be configured to achieve the ”best” performance for a given application.

4.3 Limitations and Future Work

Based on the simulation results for the H.264 AVC Encoder application, we can find that the current thread-to-core mapping in HybridThreads is quite simple (picks up the next thread to run and attaches it to an available kernel-level thread in a First-Come-First-Serve manner). In some cases, the new thread library would perform poorly as the userspace runtime library has no knowledge about the kernel and underlying hardware architecture. While the kernel-level thread in the HybridThreads library will stay on a core for its lifetime, the user-level threads will migrate among different cores as scheduled directly by the userspace library. In the future, we should develop a more sophisticated and dynamic mapping mechanism to optimize the many-threads-to-many-cores mapping (as Figure 51) and minimize the inter-thread communication. In addition, the current HybridThreads library is constructed by integrating PosixThreads and QuickThreads. We can extend the same ideas described here to integrate PosixThreads with other user-level thread library, say ContextThreads, so as to improve the portability and fit more platforms.

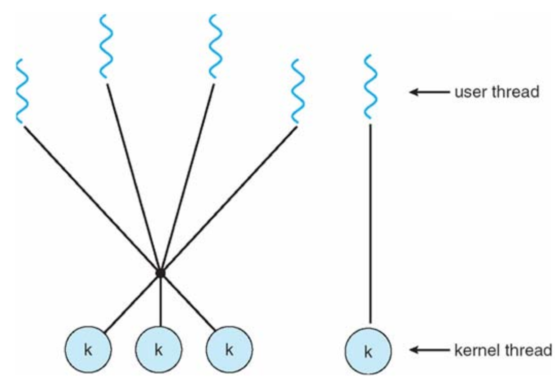


Figure 51: Two Level Threading Model [19]

References

[1] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[2] F. Warren Burton and M. Ronan Sleep. Executing Functional Programs on a Virtual Tree of Processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire, 1981.

[3] Che-Wei Chang and Rainer Dömer. System Level Modeling of a H.264 Video Encoder. Technical Report CECS-11-04, Center for Embedded Computer Systems, University of California, Irvine, June 2011.

[4] Weiwei Chen and Rainer Dömer. A Distributed Parallel Simulator for Transaction Level Models with Relaxed Timing. Technical Report CECS-11-02, Center for Embedded Computer Systems, University of California, Irvine, May 2011.

[5] Weiwei Chen and Rainer Dömer. An Optimized Compiler for Out-of-Order Parallel ESL Simulation Exploiting Instance Isolation. In *Proceedings of the Asia and South Pacific Design Automation Conference*, Sydney, Australia, 2012.

[6] Weiwei Chen, Xu Han, and Rainer Dömer. Multicore Simulation of Transaction-Level Models Using the SoC Environment. *IEEE Design and Test of Computers*, 28(3):20–31, May-June 2011.

[7] Weiwei Chen, Xu Han, and Rainer Dömer. Out-of-Order Parallel Simulation for ESL Design. In *Proceedings of the Design, Automation and Test in Europe*, Dresden, Germany, March 2012.

[8] Bastien Chopard, Philippe Combes, and Julien Zory. A Conservative Approach to SystemC Parallelization. In *International Conference on Computational Science (4)*, pages 653–660, 2006.

[9] Rainer Dömer, Weiwei Chen, and Xu Han. Parallel Discrete Event Simulation of Transaction Level Models. In *Proceedings of the Asia and South Pacific Design Automation Conference*, Sydney, Australia, 2012.

[10] Rainer Dömer, Weiwei Chen, Xu Han, and Andreas Gerstlauer. Multi-Core Parallel Simulation of System-Level Description languages. In *Proceedings of the Asia and South Pacific Design Automation Conference*, Yokohama, Japan, 2011.

[11] P Ezudheen, Priya Chandran, Joy Chandra, Biju Puthur Simon, and Deepak Ravi. Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines. In *PADS '09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 80–87, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Xu Han, Weiwei Chen, and Rainer Dömer. A Parallel Transaction-Level Model of H.264 Video Decoder. Technical Report CECS-11-03, Center for Embedded Computer Systems, University of California, Irvine, June 2011.

[13] Guantao Liu. A Hybrid Thread Library for Efficient Electronic System Level Simulation. Master's thesis, University of California, Irvine, USA, May 2013.

[14] 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.

[15] Guantao Liu and Rainer Dömer. A User-level Thread Library Built on Linux Context Functions for Efficient Electronic System Level Simulation. Technical Report CECS-13-07, Center for Embedded Computer Systems, University of California, Irvine, May 2013.

[16] Tony Mathew and Rainer Dömer. A Custom Thread Library Built on Native Linux Threads for Faster Embedded System Simulation. Technical Report CECS-11-10, Center for Embedded Computer Systems, University of California, Irvine, December 2011.

[17] Aline Mello, Issac Maia, Alain Greiner, and Francois Pecheux. Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations. In *Proceedings of the Design, Automation and Test in Europe*, Dresden, Germany, March 2010.

[18] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Scottsdale, AZ, USA, October 2010.

[19] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, eighth edition, 2009.

A Benchmark Examples

A.1 Fibo20 Model

Listing 2: Fibo20 Model

```
// Fibo20.sc: parallel Fibonacci benchmark
// author:      Rainer Doemer, Guantao Liu
// 02/15/13 GL modified to test HybridThreads library
// 09/02/11 RD created to test parallel simulators

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

// number of threads
#ifndef MAXLOOP
#define MAXLOOP 5000
#endif

// value of Fibonacci number
#define FIBONUM 25

// type of Fibonacci numbers
typedef unsigned long long number;

number fibo(number n)
{
  if (n <= 1)
    return n;
  else
    return fibo(n-1) + fibo(n-2);
}

behavior Fibo
{
  number result;

  void main(void)
  {
    result = fibo(FIBONUM);
  }
};


behavior Main
{
  Fibo  fibo0, fibo1, fibo2, fibo3, fibo4, fibo5, fibo6, fibo7, fibo8, fibo9,
        fibo10, fibo11, fibo12, fibo13, fibo14, fibo15, fibo16, fibo17, fibo18, fibo19;

  int main(void)
  {
    int i;
    printf("Fibo_par[%d,%d] starting... \n", FIBONUM, MAXTHREAD);
    for(i = 0; i < MAXLOOP; i++)
    {
      par { fibo0; fibo1; fibo2; fibo3; fibo4; fibo5; fibo6; fibo7; fibo8; fibo9;
            fibo10; fibo11; fibo12; fibo13; fibo14; fibo15; fibo16; fibo17; fibo18; fibo19;
          }
```

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

// EOF Fibo20.sc
```

B Measured Simulation Times for All Benchmarks and Applications

B.1 Simulation Time for Producer-Consumer Model

Table 21: 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%	
	236.43s	93.18s	410.56s	80.00%	
	224.66s	104.16s	416.52s	78.00%	
mu	202.11s	112.24s	390.27s	80.00%	Parallel PosixThreads
	266.14s	83.42s	424.93s	82.00%	
	233.21s	110.85s	413.37s	83.00%	
	193.4s	96.95s	385.78s	75.00%	
	208.85s	76.48s	384.89s	74.00%	
mu	204.59s	83.63s	387.25s	74.00%	HybridThreads
	209.94s	77s	387.99s	73.00%	
	210.56s	75.98s	387.01s	74.00%	

Table 22: 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%	
xi	188.41s	209.21s	408.22s	97.00%	Parallel PosixThreads
	161.07s	227.13s	393.82s	98.00%	
	222.07s	238.68s	464.63s	99.00%	
	146.75s	282.02s	429.51s	99.00%	
	177.25s	313.18s	490.11s	100.00%	
xi	261.72s	230.66s	510.3s	96.00%	HybridThreads
	219.23s	268.58s	506.34s	96.00%	
	181.58s	319.82s	514.88s	97.00%	
	262.36s	235.17s	510.12s	97.00%	
	261.31s	231.72s	507.92s	97.00%	

Table 23: Producer-Consumer Model on mu (_SPECC_NUM_SIMCPUS=1)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	320.09s	211.83s	722.31s	73.00%	Parallel PosixThreads
	308.93s	237.57s	724.76s	75.00%	
	344.8s	198.43s	736.14s	73.00%	
	349.03s	192.26s	750.98s	72.00%	
	337.18s	200.36s	736.8s	72.00%	
mu	71.19s	8.7s	79.92s	99.00%	HybridThreads
	70.26s	8.36s	78.65s	99.00%	
	70.16s	7.71s	77.91s	99.00%	
	70.95s	8.7s	79.69s	99.00%	
	69.06s	7.97s	77.07s	99.00%	

Table 24: Producer-Consumer Model on xi (_SPECC_NUM_SIMCPUS=1)

Hostname	**Usr Time**	**Sys Time**	**Elapsed Time**	**CPU Load**	**Thread Library**
xi	199.08s	435.93s	696.22s	91.00%	Parallel PosixThreads
	249.23s	421.64s	747.61s	89.00%	
	218.95s	435.84s	725.8s	90.00%	
	241.24s	425.59s	738.63s	90.00%	
	252.27s	381.4s	693.59s	91.00%	
xi	46.64s	7.46s	54.29s	99.00%	HybridThreads
	46.5s	7.75s	54.45s	99.00%	
	46.56s	8.14s	54.89s	99.00%	
	46.58s	7.66s	54.44s	99.00%	
	48.83s	7.55s	56.58s	99.00%	

B.2 Simulation Time for Fibo20 Model

Table 25: Fibo20 Model on mu

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	234.56s	0.25s	234.89s	99.00%	QuickThreads
	233.92s	0.25s	234.25s	99.00%	
	234.55s	0.28s	234.91s	99.00%	
	234.1s	0.26s	234.45s	99.00%	
	234.92s	0.27s	235.27s	99.00%	
mu	227.14s	0.31s	227.53s	99.00%	ContextThreads
	227.64s	0.28s	228.01s	99.00%	
	227.19s	0.29s	227.55s	99.00%	
	227.4s	0.31s	227.79s	99.00%	
	226.78s	0.32s	227.18s	99.00%	
mu	230.84s	1.83s	232.93s	99.00%	PosixThreads
	230.97s	1.81s	233.04s	99.00%	
	230.82s	1.86s	232.94s	99.00%	
	231.19s	1.8s	233.25s	99.00%	
	231.11s	1.84s	233.2s	99.00%	
mu	265.7s	4.32s	96.43s	280.00%	Parallel PosixThreads
	266.48s	4.23s	95.75s	282.00%	
	266.11s	4.3s	95.92s	281.00%	
	266.08s	4.35s	95.96s	281.00%	
	266.11s	4.36s	95.93s	281.00%	
mu	232.97s	1.32s	60s	390.00%	HybridThreads
	234.49s	0.85s	60.12s	391.00%	
	233.93s	1.19s	60.02s	391.00%	
	234.44s	0.9s	60.18s	391.00%	
	233.3s	1.09s	60.04s	390.00%	

Table 26: Fibo20 Model on xi

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	160.06s	0.16s	160.69s	99.00%	QuickThreads
	159.93s	0.14s	160.55s	99.00%	
	160.05s	0.15s	160.7s	99.00%	
	159.92s	0.15s	160.57s	99.00%	
	160.04s	0.17s	160.69s	99.00%	
xi	160.27s	0.2s	160.93s	99.00%	ContextThreads
	160.11s	0.2s	160.79s	99.00%	
	160.13s	0.18s	160.8s	99.00%	
	160.25s	0.18s	160.92s	99.00%	
	160.4s	0.21s	161.09s	99.00%	
xi	160.59s	1.65s	162.89s	99.00%	PosixThreads
	160.66s	1.65s	162.94s	99.00%	
	160.55s	1.66s	162.86s	99.00%	
	160.42s	1.67s	162.74s	99.00%	
	160.76s	1.65s	163.06s	99.00%	
xi	442.41s	7.54s	58.05s	775.00%	Parallel PosixThreads
	443.12s	7.33s	57.83s	778.00%	
	443.54s	7.15s	58.24s	773.00%	
	443.79s	7.11s	58.25s	774.00%	
	442.93s	7.21s	57.93s	777.00%	
xi	396.056s	2.38s	23.92s	1665.00%	HybridThreads
	424.37s	2.54s	25.5s	1674.00%	
	380.2s	2.35s	23.06s	1658.00%	
	431.17s	2.62s	25.93s	1672.00%	
	380.74s	2.32s	22.96s	1668.00%	

Table 27: Fibo20 Model on mu (_SPECC_NUM_SIMCPUS=1)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	350.37s	4.17s	356.44s	99.00%	Parallel PosixThreads
	350.53s	4.25s	356.63s	99.00%	
	351.01s	4.16s	356.97s	99.00%	
	350.98s	4.15s	356.94s	99.00%	
	350.78s	4.13s	356.8s	99.00%	
mu	231.74s	1.39s	233.25s	99.00%	HybridThreads
	231.1s	0.84s	232.03s	99.00%	
	231.06s	1.43s	232.61s	99.00%	
	231.19s	0.87s	232.16s	99.00%	
	231.5s	1.38s	233s	99.00%	

Table 28: Fibo20 Model on xi (_SPECC_NUM_SIMCPUS=1)

Hostname	**Usr Time**	**Sys Time**	**Elapsed Time**	**CPU Load**	**Thread Library**
xi	298.96s	5.14s	306.91s	99.00%	Parallel PosixThreads
	299.06s	5.16s	307.08s	99.00%	
	298.55s	5.54s	306.87s	99.00%	
	298.66s	5.52s	307.13s	99.00%	
	299.21s	4.86s	306.84s	99.00%	
xi	157.19s	0.74s	158.62s	99.00%	HybridThreads
	157.19s	0.7s	158.5s	99.00%	
	157.12s	0.73s	158.54s	99.00%	
	157.02s	0.71s	158.4s	99.00%	
	157.23s	0.69s	158.6s	99.00%	

B.3 Simulation Time for JPEG Encoder

Table 29: JPEG Encoder on mu

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	2.24s	0.04s	2.3s	99.00%	QuickThreads
	2.23s	0.05s	2.29s	99.00%	
	2.24s	0.04s	2.29s	99.00%	
	2.23s	0.05s	2.29s	99.00%	
	2.23s	0.05s	2.29s	99.00%	
mu	2.16s	0.09s	2.26s	99.00%	ContextThreads
	2.16s	0.09s	2.26s	99.00%	
	2.15s	0.1s	2.26s	99.00%	
	2.17s	0.09s	2.26s	99.00%	
	2.16s	0.1s	2.26s	99.00%	
mu	2.28s	0.35s	2.64s	99.00%	PosixThreads
	2.25s	0.38s	2.64s	99.00%	
	2.29s	0.33s	2.64s	99.00%	
	2.27s	0.36s	2.64s	99.00%	
	2.27s	0.36s	2.64s	99.00%	
mu	3.77s	0.74s	3.32s	136.00%	Parallel PosixThreads
	3.76s	0.75s	3.34s	135.00%	
	3.78s	0.76s	3.35s	135.00%	
	3.77s	0.78s	3.35s	135.00%	
	3.78s	0.77s	3.36s	135.00%	
mu	4s	0.63s	3.4s	136.00%	HybridThreads
	3.97s	0.65s	3.41s	135.00%	
	3.97s	0.66s	3.4s	136.00%	
	3.98s	0.65s	3.4s	136.00%	
	4.01s	0.61s	3.39s	136.00%	

Table 30: JPEG Encoder on xi

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	2.21s	0.03s	2.26s	99.00%	QuickThreads
	1.98s	0.03s	2.02s	99.00%	
	1.99s	0.03s	2.03s	99.00%	
	2.19s	0.04s	2.24s	99.00%	
	1.99s	0.02s	2.02s	99.00%	
xi	2.02s	0.03s	2.06s	99.00%	ContextThreads
	2s	0.06s	2.07s	99.00%	
	2.21s	0.05s	2.3s	98.00%	
	2.23s	0.06s	2.33s	98.00%	
	1.99s	0.06s	2.06s	99.00%	
xi	2.08s	0.35s	2.45s	99.00%	PosixThreads
	2.08s	0.35s	2.44s	99.00%	
	2.26s	0.4s	2.67s	99.00%	
	2.27s	0.36s	2.65s	99.00%	
	2.29s	0.36s	2.66s	99.00%	
xi	4.32s	1.19s	3.7s	149.00%	Parallel PosixThreads
	4.5s	1.43s	3.92s	151.00%	
	4.75s	1.29s	4.01s	150.00%	
	4.58s	1.29s	3.82s	153.00%	
	4.56s	1.38s	4.05s	146.00%	
xi	4.37s	0.99s	3.7s	145.00%	HybridThreads
	4.43s	1.12s	3.8s	146.00%	
	4.44s	1.11s	3.79s	146.00%	
	4.43s	1.14s	3.8s	146.00%	
	4.44s	1.14s	3.81s	146.00%	

Table 31: JPEG Encoder on mu (_SPECC_NUM_SIMCPUS=1)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	3.78s	0.5s	4.53s	94.00%	Parallel PosixThreads
	3.76s	0.47s	4.48s	94.00%	
	3.81s	0.48s	4.57s	94.00%	
	3.68s	0.5s	4.4s	94.00%	
	3.68s	0.46s	4.4s	94.00%	
mu	2.33s	0.04s	2.38s	99.00%	HybridThreads
	2.33s	0.05s	2.38s	99.00%	
	2.33s	0.04s	2.38s	99.00%	
	2.34s	0.04s	2.39s	99.00%	
	2.34s	0.04s	2.38s	99.00%	

Table 32: JPEG Encoder on xi (_SPECC_NUM_SIMCPUS=1)

Hostname	**Usr Time**	**Sys Time**	**Elapsed Time**	**CPU Load**	**Thread Library**
xi	4.43s	1.17s	5.81s	96.00%	Parallel PosixThreads
	4.4s	1.21s	5.82s	96.00%	
	4.36s	1.28s	5.86s	96.00%	
	4.26s	1.34s	5.79s	96.00%	
	4.45s	1.17s	5.79s	97.00%	
xi	2.3s	0.03s	2.34s	99.00%	HybridThreads
	2.27s	0.03s	2.32s	99.00%	
	2.27s	0.03s	2.32s	99.00%	
	2.27s	0.03s	2.32s	99.00%	
	2.27s	0.04s	2.32s	99.00%	

B.4 Simulation Time for H.264 Decoder

Table 33: H.264 Decoder on mu

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	175.61s	2.46s	181.59s	98.00%	QuickThreads
	175.54s	2.8s	179.76s	99.00%	
	175.6s	2.72s	180.35s	98.00%	
	175.7s	2.65s	180.96s	98.00%	
	175.74s	2.8s	182.33s	97.00%	
mu	172.82s	2.75s	176.64s	99.00%	ContextThreads
	172.83s	2.75s	176.63s	99.00%	
	172.8s	2.73s	177.01s	99.00%	
	172.84s	2.71s	176.6s	99.00%	
	172.85s	2.79s	176.81s	99.00%	
mu	174.88s	2.96s	179.61s	99.00%	PosixThreads
	174.95s	3.05s	178.82s	99.00%	
	174.96s	2.96s	181.54s	98.00%	
	174.82s	3.11s	178.94s	99.00%	
	174.83s	3.04s	178.72s	99.00%	
mu	196.65s	3.69s	126.83s	157.00%	Parallel PosixThreads
	196s	5.21s	127.54s	157.00%	
	196.64s	5.31s	128.48s	157.00%	
	195.95s	5.32s	127.28s	158.00%	
	196.27s	5.15s	127.64s	157.00%	
mu	194.35s	4.52s	123.58s	160.00%	HybridThreads
	194.16s	4.52s	124.13s	160.00%	
	193.82s	4.59s	123.5s	160.00%	
	194.2s	4.62s	124.01s	160.00%	
	193.77s	4.55s	124.27s	159.00%	

Table 34: H.264 Decoder on xi

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	162.67s	1.82s	165.06s	99.00%	QuickThreads
	162.82s	2.18s	167.13s	98.00%	
	163.09s	2.8s	167.67s	98.00%	
	162.33s	2.87s	167.21s	98.00%	
	162.78s	2.85s	166.33s	99.00%	
xi	163.24s	2.57s	167.23s	99.00%	ContextThreads
	163.75s	2.71s	167.15s	99.00%	
	163.26s	2.56s	167.85s	98.00%	
	163.18s	2.74s	167.02s	99.00%	
	163.18s	2.74s	167.73s	98.00%	
xi	164.63s	4.04s	170.43s	98.00%	PosixThreads
	164.71s	3.88s	170.06s	99.00%	
	164.1s	3.36s	168.92s	99.00%	
	164.58s	3.42s	169.99s	98.00%	
	164.3s	3.38s	168.32s	99.00%	
xi	320.81s	5.88s	208.94s	156.00%	Parallel PosixThreads
	320.64s	5.9s	210.3s	155.00%	
	321.75s	5.95s	210.18s	155.00%	
	321.91s	5.8s	209.85s	156.00%	
	319.16s	5.77s	208.79s	155.00%	
xi	319.47s	5.27s	207.12s	156.00%	HybridThreads
	319.61s	5.6s	207.62s	156.00%	
	319.82s	5.54s	208.64s	155.00%	
	319.79s	5.4s	208.01s	156.00%	
	320s	5.52s	207.94s	156.00%	

Table 35: H.264 Decoder on xi (_SPECC_NUM_SIMCPUS=1)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	311.34s	5.56s	319.01s	99.00%	Parallel PosixThreads
	311.66s	5.85s	319.75s	99.00%	
	311.05s	6.03s	319.67s	99.00%	
	311.5s	5.66s	319.03s	99.00%	
	311.15s	5.74s	319.61s	99.00%	
xi	164.48s	2.69s	170.08s	99.00%	HybridThreads
	164.04s	3.02s	169.94s	98.00%	
	163.39s	3.16s	168.75s	98.00%	
	164.69s	3.71s	169.94s	99.00%	
	164.77s	3.58s	172.72s	97.00%	

B.5 Simulation Time for H.264 Encoder

Table 36: H.264 Encoder on mu

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
mu	2368.49s	29.87s	2399.27s	99.00%	QuickThreads
	2368.59s	29.86s	2399.35s	99.00%	
	2368.79s	29.68s	2399.37s	99.00%	
	2368.26s	30.18s	2399.34s	99.00%	
	2368.04s	29.92s	2398.86s	99.00%	
mu	2356.69s	13.13s	2370.7s	99.00%	ContextThreads
	2356.9s	13s	2370.79s	99.00%	
	2356.86s	13.19s	2370.94s	99.00%	
	2357.26s	13.99s	2372.36s	99.00%	
	2357.37s	13.13s	2371.44s	99.00%	
mu	2364.12s	57.71s	2428.51s	99.00%	PosixThreads
	2364.12s	57.7s	2428.53s	99.00%	
	2364.02s	57.74s	2428.41s	99.00%	
	2363.91s	57.59s	2428.17s	99.00%	
	2364.17s	57.25s	2428.07s	99.00%	
mu	2964.86s	142.94s	1823.44s	170.00%	Parallel PosixThreads
	2965.45s	142.11s	1822.57s	170.00%	
	2966.02s	143.11s	1823.64s	170.00%	
	2965.78s	142.79s	1823.65s	170.00%	
	2964.64s	144.12s	1824.45s	170.00%	
mu	2770.74s	51.84s	1575.23s	179.00%	HybridThreads
	2778.79s	51.21s	1548.9s	182.00%	
	2775.37s	50.92s	1545.12s	182.00%	
	2779.07s	51.75s	1546.93s	182.00%	
	2778.08s	50.49s	1546.41s	182.00%	

Table 37: H.264 Encoder on xi

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	2269.93s	20.05s	2296.19s	99.00%	QuickThreads
	2268.89s	19.83s	2294.93s	99.00%	
	2268.9s	20.13s	2295.24s	99.00%	
	2268.62s	20.09s	2294.92s	99.00%	
	2268.45s	20.23s	2294.9s	99.00%	
xi	2252.25s	9.42s	2267.81s	99.00%	ContextThreads
	2250.49s	9.46s	2266.09s	99.00%	
	2251.55s	9.53s	2267.21s	99.00%	
	2251.93s	9.56s	2267.62s	99.00%	
	2251.5s	9.61s	2267.24s	99.00%	
xi	2259.43s	53.53s	2324.73s	99.00%	PosixThreads
	2259.28s	53.95s	2325.03s	99.00%	
	2261.52s	53.54s	2326.73s	99.00%	
	2261.91s	54.58s	2328.16s	99.00%	
	2262.16s	55.01s	2328.82s	99.00%	
xi	5817.07s	206.59s	1976.63s	304.00%	Parallel PosixThreads
	5820.55s	203.17s	1978.13s	304.00%	
	5823.13s	204.95s	1976.75s	304.00%	
	5835.46s	205.86s	1984.39s	304.00%	
	5838.96s	202.43s	1978.37s	305.00%	
xi	6473.3s	131.28s	1812.12s	364.00%	HybridThreads
	4351.96s	58.84s	3730.8s	118.00%	
	4352.45s	59.31s	3730.74s	118.00%	
	4352.13s	59.75s	3736.23s	118.00%	
	4352.91s	61.48s	3575.33s	121.00%	

Table 38: H.264 Encoder on xi (_SPECC_NUM_SIMCPUS=12)

Hostname	Usr Time	Sys Time	Elapsed Time	CPU Load	Thread Library
xi	5680.96s	200.43s	2077.6s	283.00%	Parallel PosixThreads
	5672.61s	201.47s	2076.25s	282.00%	
	5657.28s	205.26s	2079.84s	281.00%	
	5788.42s	202.12s	2087.45s	286.00%	
	5749.75s	202.34s	2084.39s	285.00%	
xi	4358.95s	79.74s	1765.1s	251.00%	HybridThreads (core 0,1,2,...,11)
	4353.94s	62.33s	2706.05s	163.00%	
	4349.17s	60.08s	2874.86s	153.00%	
	4356.25s	80.65s	1765.22s	251.00%	
	4355.04s	80.53s	1763.93s	251.00%	
xi	6723.55s	82.7s	2012.48s	338.00%	HybridThreads (core 0,2,4,...,22)
	4343.05s	54.83s	2890.15s	152.00%	
	6724.97s	83.21s	2012.87s	338.00%	
	6725.81s	82.26s	2013.54s	338.00%	
	4341.04s	55s	2888.65s	152.00%	

Table 39: H.264 Encoder on xi (_SPECC_NUM_SIMCPUS=6)

Hostname	**Usr Time**	**Sys Time**	**Elapsed Time**	**CPU Load**	**Thread Library**
xi	4740.68s	183.64s	2238.9s	219.00%	Parallel PosixThreads
	4737.38s	186.38s	2240.57s	219.00%	
	4740.17s	187.27s	2241.21s	219.00%	
	4734.11s	179s	2235.04s	219.00%	
	4726.38s	182.49s	2234.29s	219.00%	
xi	4342.36s	54.31s	2360.76s	186.00%	HybridThreads (core 0,2,4,,10)
	4335.22s	54.4s	2357.52s	186.00%	
	4343.75s	55.4s	2361.92s	186.00%	
	4342.16s	55.52s	2361.66s	186.00%	
	4339.63s	54.99s	2359.19s	186.00%	

Table 40: H.264 Encoder on xi (_SPECC_NUM_SIMCPUS=1)

Hostname	**Usr Time**	**Sys Time**	**Elapsed Time**	**CPU Load**	**Thread Library**
xi	4323.56s	171.19s	4567.59s	98.00%	Parallel PosixThreads
	4316.51s	164.35s	4553.21s	98.00%	
	4318.02s	169.27s	4560.35s	98.00%	
	4326.82s	167.4s	4566.88s	98.00%	
	4319.82s	164.9s	4557.1s	98.00%	
xi	2265.09s	21.99s	2293.29s	99.00%	HybridThreads
	2265.13s	21.67s	2293.01s	99.00%	
	2264.84s	21.52s	2292.57s	99.00%	
	2265.02s	21.58s	2292.81s	99.00%	
	2265.27s	21.28s	2292.75s	99.00%	

B.6 Time Profiling for All Benchmarks and Examples on mu

Table 41: Time Profiling of HybridThreads Library on mu

Benchmark	Usr Time	Sys Time	Elapsed Time	Lock Time	App Time	% in Lock
Prod-Cons	205.95s	98.37s	400.96s	280.92s	40.45s	87.41%
	208.16s	94.4s	404.55s	284.78s	39.26s	87.89%
	247.68s	70.21s	413.43s	285.2s	43.05s	86.89%
	220.1s	81s	405.75s	286.37s	39.97s	87.75%
	223.99s	79s	407.03s	286.53s	41.75s	87.28%
Fibo20	232.45s	1.13s	59.8s	0.33s	231.83s	0.14%
	232.18s	1.07s	59.66s	0.31s	231.47s	0.14%
	232.42s	1.1s	59.75s	0.32s	213.75s	0.14%
	232.27s	1.1s	59.74s	0.32s	213.59s	0.14%
	233.09s	1.13s	60.03s	0.36s	232.51s	0.15%
JPEG Encoder	3.99s	0.67s	3.43s	1.19s	3.58s	24.93%
	4.02s	0.64s	3.42s	1.18s	3.58s	24.75%
	4.04s	0.6s	3.4s	1.15s	3.57s	24.38%
	3.98s	0.68s	3.42s	1.2s	3.57s	25.15%
	4.07s	0.6s	3.42s	1.18s	3.57s	24.84%
H.264 Encoder	2778.41s	51.63s	1549.05s	13.17s	2779.37s	0.47%
	2778.2s	50.95s	1546.64s	11.47s	2776.33s	0.41%
	2777.19s	50.07s	1545.36s	11.38s	2774.19s	0.41%
	2778.41s	50.13s	1546.02s	11.44s	2775.54s	0.41%
	2777.62s	50.15s	1545.28s	11.56s	2774.65s	0.41%
H.264 Decoder	194s	4.33s	123.52s	1.09s	197.98s	0.55%
	194.04s	4.56s	125.23s	1.01s	199.71s	0.50%
	193.94s	4.67s	123.9s	1.15s	198.03s	0.58%
	193.96s	4.5s	123.67s	1.1s	198s	0.55%
	194.55s	4.48s	124.21s	1.1s	198.56s	0.55%

Table 42: Time Profiling of HybridThreads Library on mu (_SPECC_NUM_SIMCPUS=1)

Benchmark	Usr Time	Sys Time	Elapsed Time	Lock Time	App Time	% in Lock
Prod-Cons	75.1s	8.07s	83.21s	4.22s	20.89s	16.80%
	74.85s	8.08s	82.96s	4.22s	20.84s	16.83%
	75.38s	7.92s	83.33s	4.28s	21.05s	16.89%
	75.1s	8.02s	83.16s	4.22s	20.7s	16.92%
	74.94s	7.96s	82.94s	4.22s	21s	16.73%
Fibo20	231.49s	1.36s	232.97s	0.09s	230.7s	0.04%
	230.94s	0.81s	231.85s	0.05s	230.2s	0.02%
	231.47s	1.15s	232.72s	0.08s	230.67s	0.03%
	230.77s	0.88s	231.75s	0.05s	230.19s	0.02%
	231.73s	1.33s	233.18s	0.09s	230.9s	0.04%
JPEG Encoder	2.35s	0.03s	2.39s	0.01s	2.25s	0.36%
	2.34s	0.04s	2.39s	0.01s	2.26s	0.35%
	2.34s	0.04s	2.38s	0.01s	2.25s	0.36%
	2.34s	0.04s	2.38s	0.01s	2.25s	0.36%
	2.34s	0.04s	2.39s	0.01s	2.25s	0.36%
H.264 Encoder	2371.12s	30.89s	2402.92s	0.21s	2367.28s	0.01%
	2371.01s	31.15s	2403.1s	0.21s	2367.48s	0.01%
	2371.59s	31.09s	2403.59s	0.22s	2367.64s	0.01%
	2370.45s	30.92s	2402.28s	0.2s	2366.62s	0.01%
	2371.22s	31.04s	2403.17s	0.2s	2367.22s	0.01%
H.264 Decoder	175.25s	3.94s	179.94s	0.01s	179.21s	0.01%
	175.23s	4.23s	180.8s	0.01s	180.05s	0.01%
	175.28s	4.16s	179.77s	0.01s	179.01s	0.01%
	175.36s	4.21s	181.47s	0.01s	180.73s	0.01%
	175.57s	4.06s	180.3s	0.01s	179.54s	0.01%

B.7 Time Profiling for All Benchmarks and Examples on xi

Table 43: Time Profiling of HybridThreads Library on xi

Benchmark	Usr Time	Sys Time	Elapsed Time	Lock Time	App Time	% in Lock
Prod-Cons	261.64s	247.73s	521.83s	323.31s	60.58s	84.22%
	259.33s	253.84s	522.68s	321.47s	57.33s	84.87%
	233.99s	266.73s	513.23s	323.73s	55.59s	85.34%
	231.61s	270.38s	514.34s	326.4s	56.99s	85.14%
	228.34s	273.77s	514.33s	326.87s	57.13s	85.12%
Fibo20	419.14s	2.33s	25.53s	6.49s	418.02s	1.53%
	442.89s	2.44s	26.73s	6.5s	441.78s	1.45%
	423.74s	2.38s	25.39s	6.3s	422.66s	1.47%
	407.22s	2.18s	24.42s	6.37s	406.01s	1.54%
	461.3s	2.56s	27.55s	6.69s	460.19s	1.43%
JPEG Encoder	4.35s	1s	3.72s	1.33s	3.93s	25.29%
	4.43s	1.18s	3.83s	1.66s	4s	29.33%
	4.43s	1.15s	3.82s	1.61s	3.99s	28.75%
	4.48s	1.11s	3.8s	1.59s	4s	28.44%
	4.42s	1.17s	3.8s	1.59s	4s	28.44%
H.264 Encoder	4351.72s	60.19s	3735.29s	59.56s	4342.79s	1.35%
	4353.78s	62.47s	3584.18s	67.6s	4344.54s	1.53%
	4354.77s	62.96s	3595.21s	68.09s	4344.98s	1.54%
	4354.92s	62.52s	3581.76s	68.07s	4345.31s	1.54%
	4354.03s	60.88s	3736.13s	59.41s	4344.77s	1.35%
H.264 Decoder	319.73s	5.22s	207.57s	1.92s	324.16s	0.59%
	319.99s	5.64s	208.49s	1.85s	324.35s	0.57%
	319.76s	5.62s	209.2s	2.26s	325.35s	0.69%
	319.98s	5.47s	209.27s	1.87s	324.81s	0.57%
	319.94s	5.55s	208.28s	1.85s	324.23s	0.57%

Table 44: Time Profiling of HybridThreads Library on xi (_SPECC_NUM_SIMCPUS=1)

Benchmark	Usr Time	Sys Time	Elapsed Time	Lock Time	App Time	% in Lock
Prod-Cons	49.47s	7.85s	57.52s	3.16s	12.06s	20.76%
	52.47s	8.99s	61.67s	3.27s	13.77s	19.19%
	50.15s	7.78s	58.15s	3.25s	12.2s	21.04%
	50.22s	8.37s	58.79s	3.19s	12.25s	20.66%
	50.9s	7.73s	58.84s	3.24s	12.45s	20.65%
Fibo20	157.1s	0.7s	158.45s	0.21s	156.75s	0.13%
	157.02s	0.7s	158.39s	0.25s	156.67s	0.16%
	157.22s	0.72s	158.65s	0.25s	156.87s	0.16%
	157.02s	0.72s	158.42s	0.25s	156.67s	0.16%
	157.2s	0.72s	158.61s	0.24s	156.86s	0.15%
JPEG Encoder	2.31s	0.03s	2.36s	0.01s	2.23s	0.45%
	2.06s	0.04s	2.11s	0.01s	2s	0.50%
	2.06s	0.03s	2.11s	0.01s	2s	0.50%
	2.07s	0.03s	2.11s	0.01s	2s	0.50%
	2.07s	0.03s	2.11s	0.01s	2s	0.50%
H.264 Encoder	2267.7s	21.87s	2295.79s	0.19s	2263.43s	0.01%
	2265.95s	21.78s	2293.94s	0.18s	2261.76s	0.01%
	2266.62s	21.83s	2294.67s	0.18s	2262.64s	0.01%
	2264.95s	21.93s	2293.08s	0.2s	2260.77s	0.01%
	2266.73s	21.5s	2294.44s	0.21s	2262.27s	0.01%
H.264 Decoder	163.6s	3.06s	167.29s	0.01s	166.4s	0.01%
	163.06s	2.87s	167.84s	0.01s	166.66s	0.01%
	163.59s	3.11s	169.12s	0.01s	167.16s	0.01%
	163.33s	3.06s	167.11s	0.01s	166.22s	0.01%
	163.36s	3.01s	168.38s	0.01s	167.07s	0.01%