Rethink Scalable M:N Threading on Modern Operating Systems

Lu Gong[1*], Zhanqiang Li[2], Tao Dong[2], Youkai Sun[2]

[1] Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China.
[2] Shengli Geophysical Research Institute of Sinopec, Beijing, China.

* Corresponding author. Tel.: 13918665921; email: iceboy@sjtu.edu.cn
Manuscript submitted June 4, 2015; accepted September 8, 2015.
doi: 10.17706/jcp.11.3.176-188

Abstract: In modern operating systems, thread is the only primitive to exploit the concurrency provided by multiple processors, which allows programmers to write sequential code that can be executed in parallel. However, these operating systems are known to be inefficient when the number of threads comes to the magnitude of 103. Therefore server developers have to avoid the use of naive threading when they need to handle thousands of simultaneous client sessions.

In this paper, we propose a highly scalable user-level threading library µThread. It enables programmer to write heavily concurrent threaded code without losing performance. Our framework achieves the M:N threading model, which hosts M user-level threads on top of N kernel-level threads. To demonstrate the usefulness of this framework, we provide a proof-of-concept implementation of a web server application.

According to our experiments, µThread provides good performance and great scalability. The web server based on µThread becomes the best performance user-mode web server under Windows.

Key words: Operating system, threading, scalability.

1. Introduction

The performance of a single core processor has come to its bottleneck. To further improve computational capability, making use of multiple processor cores has become the mainstream solution. On the other side, modern software development often needs to confront concurrency. For example, a typical web server needs to handle thousands of simultaneous client connections. The evolvement of hardware and software brings about mixed complexity which requires developers to build a system that can both handle concurrent requests and make use of multiple processors.

There are basically two kinds of abstraction which enable software developers to build software that can process in parallel [1]. The first abstraction comes with thread and blocking operations, namely multi-threading. Programmers write code in the same sequence as the control flow, which is executed in a thread environment. Multiple control flows can occur simultaneously by creating multiple threads. When an operation needs result that is currently unavailable, the thread issuing the operation is blocked until the result is resolved. The second abstraction makes use of asynchronous operations. Programmers do not write code in the control flow sequence. Instead, when an operation may need currently unavailable results, a callback must be specified. When the result is resolved, the callback is invoked.

Despite the underlying hardware is naturally asynchronous, commodity operating systems all provide

synchronous programming interfaces [2]. The hardware of a computer system is composed of devices and processors which can work in parallel. When a device is ready, it interrupts the processors for notification. Commodity operating systems wrapped this parallelism by providing multi-threaded interfaces, which enables programmers to write synchronous code to operate on the asynchronous system.

This design causes two problems. Firstly, a single thread can only run on a single processor at a time. To make full use of multiple processors, the application must create multiple threads where the number of threads is no less than the number of processors. Secondly, the threading mechanism may introduce tremendous overhead when the number of threads grows too large. As an impact, multi-threaded I/O becomes the main bottleneck in the performance of modern operating systems [3].

Commodity operating systems provide I/O multiplexing mechanisms to overcome these limitations. There are kqueue for FreeBSD [4], epoll for Linux [5], and I/O completion port for Windows [6]. These mechanisms allow a single thread to block on multiple I/O operations, thus greatly reduced the total number of threads required. With multi-threading and I/O multiplexing, developers of I/O intensive programs are encouraged to create multiple worker threads to wait on a single queue. This is not multi-threading meant to be, which causes program to be obscure, hard to debug, and cannot take full advantage of the programming language.

We solved this problem by building a user-level threading library μThread, with fibers scheduled by the I/O multiplexing mechanisms. The library provides a synchronous programming interface similar to a traditional threading library. Meanwhile, the performance of μThread is identical to making direct use of the underlying asynchronous mechanisms.

Our contributions are threefold:

- We introduce highly scalable user-level threads, which enable developers to write synchronous code to make use of the asynchronous I/O multiplexing mechanisms. Our threading library, μThread, comes with a similar API of a traditional threading library (e.g. pthread), which is sufficient for building a typical server program. Under the circumstances where functionality or performance is not satisfied, μThread can coexist with the native threading library of the operating system in the same process.
- We implement a web server which performs best among several user-mode web servers under Windows. The selected web servers for benchmark are Apache, Nginx, Node.js and IIS. Our web server performs best among all except IIS, which is implemented as a kernel-mode driver.
- We provide a way to use the exact same code to measure performance between native threaded I/O and multiplexed I/O. I/O multiplexing mechanisms always come with an asynchronous interface. Benchmarks between them are either limited to basic operations or have to face the criticism of different implementations.

The rest of the paper is organized as follows. Section 2 presents the design of μThread. Section 3 evaluates the various performance metrics and shows some findings. Section 4 introduces the related work and Section 5 concludes the paper.

2. Design

2.1. Motivation

We start by briefly illustrating the similarities and differences between synchronous and asynchronous programming.

Assume that we have a socket with stream-oriented connection and a buffer of 4,096 bytes. We are required to read data from the socket to fill up the entire buffer. Since the read operation of a stream ends up as soon as any data is arrived, we need to read multiple times.

In synchronous programming, a typical prototype of the read function would be read(fd, buffer, size)

where the parameters are the file descriptor, the pointer to the buffer and the size of the buffer. The read operation may complete successfully with a non-negative return value indicating the actual read size, where 0 implies the end of the stream. A negative return value suggests that the operation is failed. It is natural to embed the function call into a while loop to form an algorithm to fill up the buffer. There is a sliding pointer which points to the remaining part of the buffer, and a counter which counts the remaining size of the buffer, as shown below.

```
char buffer[4096];
char *p = buffer;
int remain = sizeof(buffer);

while (remain) {
        int result = read(socket, p, remain);
        if (result <= 0) {
                // error handling
  }
        p += result;
        remain -= result;
}
```

In asynchronous programming, the prototype of the read function should include a callback function and a user defined state parameter, which becomes async_read(fd, buffer, size, callback, state). The asynchronous read operation completes in one of the four situations:

- Asynchronous success. The operation is pending while the read function returns, until the callback function is invoked with the state parameter and the result.
- Asynchronous failure. The callback function will be invoked on completion with the state parameter and the error information.
- Synchronous success. Although the result is ready before the read function returns, most asynchronous programming libraries still invokes the callback function to provide results thus simplify programming.
- Synchronous failure. The read function will directly return with error information, and the callback function will not be invoked.

With this asynchronous interface, programmers cannot use while loop to fill up the buffer because that every call to the read function splits the control flow into two parts. Programmers are forced to explicitly implement a state machine even if the algorithm is simple and trivial. A typical implementation using the asynchronous interface is shown below.

```
typedef struct {
        char buffer[4096];
        char *p;
        int remain;
} state_t;

state_t *s = malloc(sizeof(state_t));
s->p = s->buffer;
s->remain = sizeof(s->buffer);

if (async_read(s->p, s->remain, on_read, s) < 0) {
        // error handling
}

void on_read(state_t *s, int result) {
        if (result <= 0) {
                // error handling
```

```
                free(s);
                return;
        }
        s->p += result;
        s->remain -= result;
        if (!s->remain) {
                free(s);
                return;
        }
        if (async_read(s->p, s->remain, on_read, s) < 0) {
                // error handling
    }
}
```

Programmer manually allocates the state structure on heap, which contains all of the data that need to be accessed across the read operation, including the buffer, the sliding pointer to the buffer, and the counter of remaining bytes. In this case, the use of asynchronous programming interface generates much longer code.

In general, the continuation of asynchronous pending operations is separated from the initiation of these operations, which causes disadvantages listed as follows.

Control Flow. In synchronous programming, the blocking functions can be directly included into complex control flows supported by the compiler (e.g. *for*, *while*), thus programmers can write code in the same sequence as the control flow. Additionally, structured exception handling [7] is supported by many compilers, which allows programmers to handle multiple error conditions in one place. In asynchronous programming, programmers cannot integrate pending operations into these control flows. Thus programmers are forced to implement every algorithm as an explicit state machine even though the actual control flow is relatively simple. Moreover, the sources of error conditions are scattered all around which makes programmer difficult to handle them correctly.

Memory Allocation. Most of the modern programming languages support stack based local variables, which is an ideal place to store local states. These variables are automatically allocated and deallocated on stack when the program flow enters and leaves the context of the variables. In asynchronous programming, programmers must explicitly allocate memory for local state and keep track of the variable's lifetime. Stack allocation is relatively fast and reliable, leveraging simple and useful semantics when combining with a programming language that supports resource management such as C++ [8].

We can see that there are cases where asynchronous programming is not suitable. In these cases, if there is a choice, programmers would prefer a synchronous programming interface.

2.2. I/O Multiplexing

Through the evolution of development, modern operating systems (Windows, Linux, FreeBSD) all provide two kinds of I/O model. Firstly, they provide the native model, which enables programmer to directly initiate an I/O operation meanwhile blocking the thread of initiation. Secondly, they provide the multiplexed model, which combines the wait of multiple I/O completions into one single blocking call. Fig. 1 compares the two models in respect of the interaction between the application and the operating system.

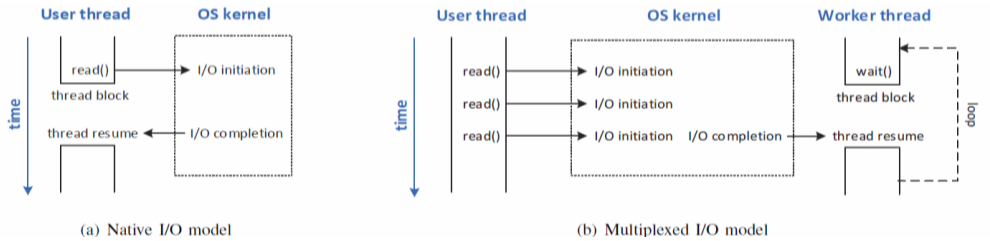


Fig. 1. The interaction between application and the OS in different I/O models.

In the native model, when a user thread calls an I/O function, such as read(), the operation is initiated and the thread is blocked. The thread is resumed after the operation is completed, as shown in Fig. 1(a). This model provides a synchronous programming interface which has been discussed in Section 2.1.

However, the native model must face the scalability problem that, when the number of threads becomes very large, the system's overall throughput may degrade heavily [9]. Because of the nature that in the native model, every pending I/O operation will occupy a thread, the number of threads must be equal to or greater than the number of the pending operations. In a typical threaded TCP server, every active connection has a pending read operation in order to actively listen for newly arrived messages. When the number of potential active connections grows high, the native model becomes out of place for its low scalability.

Modern operating systems offer multiplexed model to conquer this problem. In this model, when a user thread calls an I/O function, the operation is initiated while the thread continues to run. When an operation is completed, the kernel queues the notification. There are often worker threads waiting at the queue to manipulate the I/O completions. Worker threads may then initiate more I/O operations as the application continues to run. This model is shown in Fig. 1(b).

The multiplexed model provides an asynchronous programming interface since the processing of I/O completion is separated from the initiation. There are often libraries that slimly wrap the multiplexed model into an asynchronous programming interface described in Section 2.1 [10]. Applications that use this model often achieve good scalability but sacrifice code simplicity [11], [12].

2.3. The Duality Mapping

We have designed a user-level threading library, μThread, which has a similar programming interface with the native I/O model, and identical scalability to the direct use of the multiplexed I/O model.

The two basic operations of a blocking operation are the initiation and the notification of completion. We want to achieve good scalability, therefore we have to use the multiplexed I/O model as the base, where we can initiate a blocking operation while setting up a callback to handle the completion. On the other hand, we are going to provide a threaded programming interface, where calling a function of blocking operation not only initiates the operation but also blocks the current thread until the operation is completed.

The library with a threaded programming interface belongs to a *procedure-oriented* model, where the multiplexed I/O model is a *message-oriented* model. These two kinds of model are proved to be duals of each other. With proper mapping, a system of one model can be directly translated into the other with identical semantics and performance [13].

We introduce the use of fibers to achieve this goal. Fibers are lightweight scheduling units which are cooperatively scheduled. A fiber contains a stack for user code execution. When switching to another fiber, the stack is switched. Firstly, we create some number of native threads to host the fibers. These threads are called *worker threads*. The number of worker threads must be equal to or greater than the number of processors in the running machine in order to take full use of all of the processors. We create a *worker fiber* for each worker thread.

Secondly, we directly schedule the fiber *through* the I/O multiplexing mechanism. For each blocking operation we need to support, we provide a *wrapper* function. The wrapper function are required to be called in a user fiber other than the worker fiber. In this function, we first allocate an environment block on the current fiber's stack. We set up the environment block which contains a reference to the current fiber, and then switch to the worker fiber and initiate the operation with the I/O multiplexing mechanism, specifying the environment block as the state parameter. If the operation completed synchronously, we directly return the result or the error information. Otherwise, the worker fiber will wait on the I/O multiplexing mechanism for all pending operations after the initiation which will cause the worker thread

to be blocked. When the operation is completed, the wait will be satisfied and the worker thread will be resumed, and we switch back to the user fiber and return the result or the error information based on the reference in the environment block. A complete flow of an individual operation is shown in Fig. 2.

Thread Creation. Thread creation is a basic interface of a threading library. When a user creates a µThread, we allocate a thread control block in the user-space heap and create a fiber corresponds to the thread. We also need to add a dummy entry to the I/O multiplexing mechanism so that the fiber can be scheduled. We invoke the user-specified thread entry function in the callback of the dummy operation, and release the thread control block and destroy the fiber after the function is returned.

With thread creation implemented and all of the blocking operations mapped, we formed a complete user-level threading library. This is a duality mapping from the event-based multiplexed I/O model to a thread-based I/O model. µThread achieved an M:N threading model, which hosts M user-level threads on top of N kernel-level threads, as shown in Fig. 3.

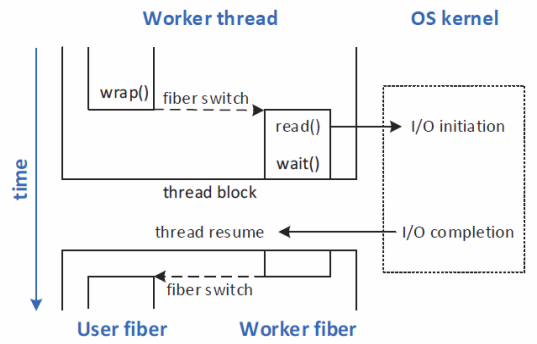


Fig. 2. Using fiber to build a synchronous programming interface on top of the asynchronous I/O multiplexing mechanisms.

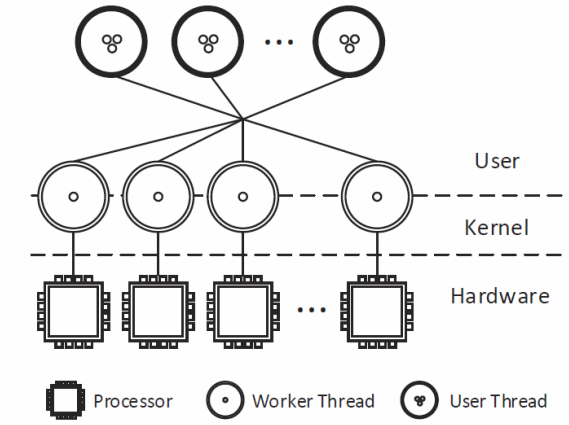


Fig. 3. The architecture of µThread, which hosts M user-level threads on top of N native worker threads running on N processors.

2.4. Discussion

In this section, we briefly discuss about some minor topics that we did not cover in previous sections.

A Race Condition. When mapping a blocking operation in a multiple processors environment to µThread, we need to first switch to the worker fiber and then initiate the operation. This is because that there are multiple worker threads waiting on the I/O multiplexing mechanism. If we switch the fiber after the initiation of operation, the operation may complete immediately and another worker thread will receive the completion before the current worker fiber switches out. The worker thread that receives the

completion will try to switch to the current fiber thus causes a race condition.

Unavoidable Blocking. A running thread may encounter unavoidable blocking, such as page faults. We may also need to use operations that are not supported by the I/O multiplexing mechanism, which will also cause the worker thread to be blocked. The solution is to have the number of *active* worker threads equal to or greater than the number of processors for all time. We may need to trace the number of active worker threads, and create new threads when necessary. This trace is supported by I/O completion port, which is the I/O multiplexing mechanism under Windows.

Yielding. Yield is a term in multi-threading which means to let the thread scheduler select a different thread to run instead of the current one. The yield implementation is trivial, we first switch to the worker fiber and add a dummy entry to the I/O multiplexing mechanism pointing at the calling fiber, and finally wait for I/O completions. However, adding the entry and wait for completions require two user-mode to kernel-mode transitions, which is far more expensive than the yield operation in the native threading library. In Section 3.2, we evaluate the performance of yield, which represents the lightest blocking operation in µThread.

Locks. Lock is an important mechanism for synchronizing multiple threads to access a resource. There are mainly three kinds of locks that can be implemented in µThread. When the threads are only likely to be blocked for a short period of time, a spin lock can be used, which behaves just like spin locks in other threading libraries. Users could directly use the lock in the underlying native thread library, which will cause blocking that discussed previously. We can explicitly implement a lock in µThread similar to the yield implementation. However, this implementation is heavy and may perform badly than previous approaches.

The Interface. We have faced a choice that we either substitute the interface of the native threading library or create a new interface. Substituting the native interface brings us benefit that users could directly link their applications to µThread without modifying the code. The drawback is that users would face a dilemma to only use one threading library in an application, suffering from the disadvantages while taking the benefits. We did not substitute the native threading interface and create an alternative interface, so that µThread can coexist with other threading libraries in the same process.

3. Evaluation

In this section, we present several benchmarks to study the performance of the threading library and its applications followed our design.

The testing machine consists of an Intel E3-1230 CPU (3.2GHz, 4-core with hyper-threading), Z68 chipset, 8GB DDR3 RAM running at 1333MHz and an Intel 82579V Gigabit Ethernet controller. In the benchmark requiring single core, we mask off other cores and disable hyper-threading in the BIOS setting. We implement two threading libraries, one following the description in Section 2 and the other being a direct wrapper of the native threading library provided by the operating system. These two threading libraries share the same interface described in Section 2.3, so that we can use the same workload to compare performance between µThread and the operating system's native threading library. The operating system for our evaluation is Windows Server 2008 R2 (64-bit).

We first create two micro-benchmark to understand the basic performance characteristics of µThread, and then present a web server benchmark to exhibit its great performance in real world applications.

3.1. Thread Creation Performance

Firstly we study the thread creation performance, where we create threads as many as possible using both the µThread threading library and the native threading library, and meanwhile we record the accumulated time usage for creating certain amount of threads. We specify a stack size of 4KB for each thread.

As shown in Fig. 4, the thread creation speed with µThread is relatively faster than the native threading

library. For instance, it takes 1.14 seconds to create 10^5 threads in μThread while the native threading library requires 2.15 seconds. We can also see that μThread allows more threads to be created.

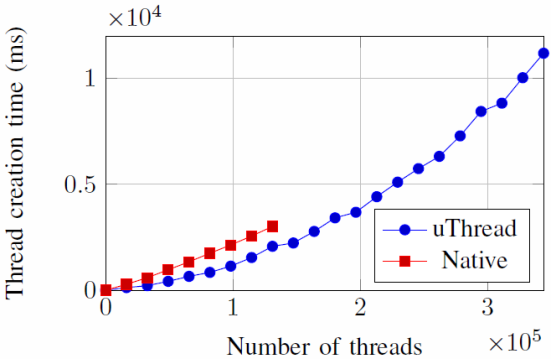


Fig. 4. A micro benchmark for thread creation performance. Lower is better.

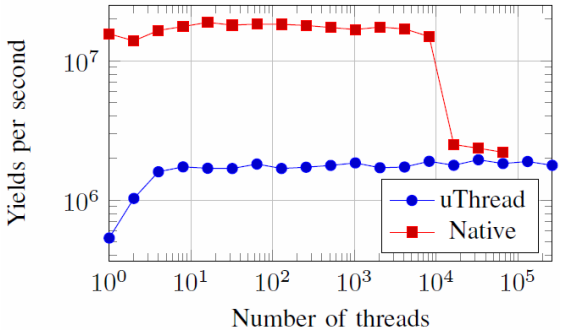


Fig. 5. A micro benchmark for yielding performance. Higher for better absolute performance. Flatter for better scalability.

The native threading library fails to create new thread when the number of threads grows to 1.3×10^5 while μThread goes along until 3.4×10^5 threads are created. This is mainly because of that a μThread only contains a user-space stack, while a native thread requires a kernel-space thread control block stored in the non-paged pool memory which can be easily run out when huge amount of threads are created. μThread allows creating more threads only limited to the total virtual memory required by thread stack.

3.2. Yielding Performance

Secondly we study the yielding performance, where we create certain amount of threads with thread bodies consist of infinite loop of yielding, which is described in Section 2.4. We start to count the number of yields when all threads start to yield for 1 second, and stop when we count for 10 seconds.

As shown in Fig. 5, the native threading library have much higher performance when the number of threads is less than 10^4. The yield operation in μThread is much heavier than the native one in the operating system, as it enters and leaves the kernel twice and operate on complex kernel data structures.

Despite the bad yielding performance, we can see that the scalability is much better. The performance grows when the number of threads grows from 1 to 4, as the system has a 4-core processor. The native threading library has a big gap of performance degradation when the number of threads grows up to 10^4, while μThread does not. The native threading library stops working when the number of threads comes near to 105, for running out of the kernel's non-paged pool memory.

This micro-benchmark suggests us that μThread is not suitable when most of the blocking operations are as light as yield. Fortunately this is not the case in most situations. We can see that the boundary of IOPS (I/O per second) in μThread is 2×10^6 in our testing machine. When the IOPS of a certain application is lower than 2×10^5, the impact of this boundary will be lower than 10% and will not become the bottleneck. We expect the good scalability takes main effect and leads to good performance in such applications.

3.3. Web Server Performance

We further evaluate our work in the context of a highly concurrent web server, which is a typical server application in real world usage.

We implemented a fully functional open-source web server named Ice, which accepts incoming connections, parses HTTP requests, reads static file from disk and sends the content back to the client. The web server is based on a threading library interface, which creates one thread per connection, and we evaluate its performance using both μThread and the native threading library.

We also collect various commonly used web servers for Windows in the benchmark. The selected web servers are IIS (7.5), Apache (2.2.27), Nginx (1.6.0) and Node.js (0.10.28), all of them are 64-bit version. IIS is the web server that comes with Windows, which contains a kernel-mode driver which accomplishes all of the work in kernel-mode when serving static files. Apache and Nginx are two widely used web servers, where Apache uses native threaded I/O and Nginx uses multiplexed I/O. Node.js is a scripting platform based on the Chrome's JavaScript runtime, which is often used as a web server for its high scalability. We write a simple script to serve static files on disk for Node.js. We turn off logging and other unused modules for all the web servers, retaining minimum functionality for maximum performance.

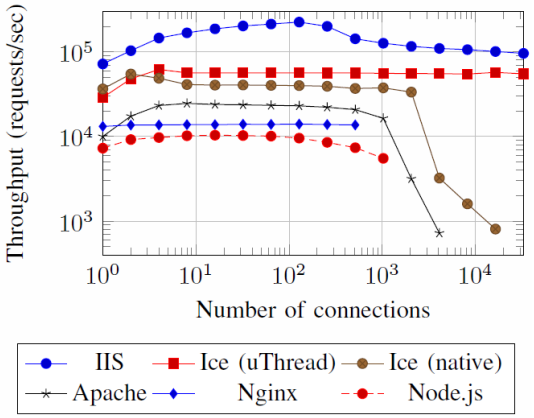


Fig. 6. A throughput performance benchmark of various web servers (multi core). Higher for better absolute performance. Flatter for better scalability.

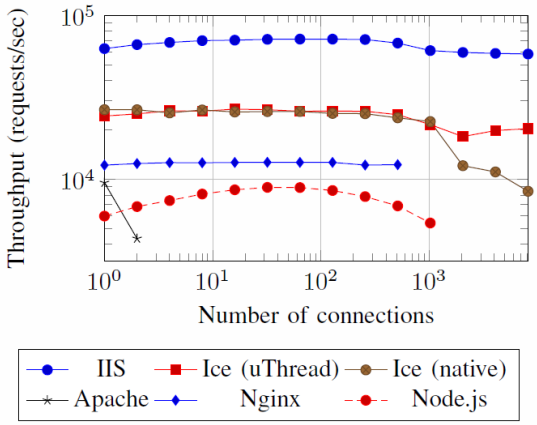


Fig. 7. A throughput performance benchmark of various web servers (single core). Higher for better absolute performance. Flatter for better scalability.

The experiment result is shown in Fig. 6. IIS takes advantage of the kernel-mode counterpart and its matured optimization therefore performed best among all web servers. Our web server, Ice, performed next to IIS when using µThread, which becomes the best user-mode web server of both absolute throughput and scalability under Windows. When Ice uses the native threading library, it perform slightly better when the number of connections is 1 or 2, slightly worse at 4 to 10^3 connections, and degrade heavily at more than $2*10^3$ connections. Apache uses native threaded I/O thus have similar scalability characteristics with Ice using the native threading library that degrade heavily when the number of connection grows higher. Nginx and Node.js use multiplexed I/O therefore have straight horizontal lines similar to Ice using µThread. Nginx accepts 103 connections at most on Windows as an implementation limitation. Despite the scalability characteristics, the absolute performance of Apache, Nginx and Node.js on Windows is lower than Ice due to the compromises made of code porting.

The comparison between the performances of same web server Ice on the two threading libraries proves that under certain circumstances in real applications, µThread enables programmer to write threaded code in a heavily concurrent environment without losing performance.

Single Core Performance. Nginx and Node.js are single-threaded and do not support multiple worker instances on Windows, so that they cannot take advantages of multiple processors. Thus we takes an additional benchmark in a single core environment, with other processor cores and hyper threading disabled in BIOS setting, to fairly compare their performance with other web servers. We expect that Nginx and Node.js will have same performance as they perform in the multi-core environment, while other web servers will perform worse due to the processor core reduction.

Fig. 7 shows the result as expected. We noticed that in single core environment, using µThread only has scalability advantage over using the native threading library, and the absolute throughput performance is

almost the same when the number of connections is lower than 10^3. Apache failed the test when the number of connections grows more than 2 in the single core environment, as further connections just stop responding requests when the first 2 connections are kept busy.

4. Related Work

Concurrency Programming Models. Thread-based and event-based are two categories of programming models that deal with concurrency [1]. The argument of which is ultimately better between the two models have lasted for several decades. In 1978, Lauer and Needham have proved that one model can be directly mapped to another, with both the same logical semantics and the absolute performance if only properly implemented [13]. However, many researchers still believe that one model is better than another regardless of the actual application scenario [9], [12], [14]. The performance of server applications of different programming models in modern operating systems is also evaluated [15], [16]. Researchers have also been working on to refrain from the weakness of both models [11], [17]-[19]. There are also researches that combine the two models to achieve both the ease of use and expressiveness of threads and the flexibility and performance of events [20], [21].

Threading Models. There are three kinds of threading models, namely N:1, 1:1 and M:N model [22]. N:1 model implements multi-threading by scheduling multiple user-level threads onto one single kernel-level thread. Thread creation and context switch are usually very fast due to the user-level implementation. There is no need for lock since only one thread is running at a time. Once the kernel-level thread is blocked, the entire process is blocked. Besides, N:1 model does not take advantage of multiple processors.

In 1:1 model, every user thread has its corresponding kernel thread. Multiple threads can run concurrently on different processors. Thread creation and context switch are often slower due to the user-kernel mode transition.

M:N model hosts M user threads on top of N kernel threads. It combines some advantages of the N:1 and 1:1 model, such as cheap creation and fast context switch [23]. This model is often criticized as hard to implement, and it does not perform best in all situations.

Using Fibers. Fibers are lightweight scheduling units which are cooperatively scheduled [24]. Modern operating systems come with preemptive scheduling, which is the opposite of cooperative scheduling. However, some of them still implement fibers in their user-level library primarily for making it easier to port applications that were designed to schedule their own threads [25]. In these operating systems, fibers run in the context of the threads that schedule them.

Capriccio presents a scalable thread package for use with high-concurrency servers [26]. It is a duality mapping from the asynchronous I/O mechanisms to the N:1 threading model, as it schedules multiple user-mode threads onto a single kernel-mode thread. Because of only one single kernel-mode thread is used, it cannot take advantages of multiple processors and the paper leaves it for future work.

C++ CSP2 presents an M:N threading model which can take full advantages of multiple processors [27]. It is a threading library that schedules their threads by itself, thus implementing high performance I/O with the library becomes a big challenge and the paper leaves it for future work.

Language Level Solutions. Computers are layered systems that a single problem can be solved at various layers. There are researches that propose language level constructs to simplify event-driven programming, such as Task [28] and Future [29]. Node.js, which is based on the JavaScript programming language, supports closures that makes it easier to directly use these constructs to build asynchronous applications [30]. The .NET Framework 4.5 introduces asynchronous programming by letting the compiler to compile synchronous, thread-based code into asynchronous, event-based code that make uses of the Task construct [31].

5. Conclusion

Concurrency and multi-processors bring about mixed complexity that needs to be carefully solved to achieve simplicity and good performance. In this paper, we present a user-level threading library, μThread, which can be directly implemented in modern operating systems without the need to modify the kernel. According to our experiments, μThread performs as well as the best user-level approach in modern operating systems on single and multiple processors in certain cases while remaining the simplicity of a synchronous threading interface. Our library can coexist with the native threading library in a single application, so that the developer will not need to balance the advantages and disadvantages of both libraries and take the best of both worlds, which pushes us to rethink the scalable M:N threading in modern operating systems.

Acknowledgment

The research work was supported by National Science and Technology Major Project (No. 2013ZX03002004) and National R&D Infrastructure and Facility Development Program (No. 2013FY111900).

References

[1] Henry, K., Verdi, M., Rita, Z., & Simon, S. (2008). Survey on parallel programming model. *Network and Parallel Computing*, 266–275.

[2] Andrew, S. T. (2007). *Modern Operating Systems*. Prentice Hall Press.

[3] Mendel, R., Edouard, B., Stephen, A. H., Emmett, W., & Anoop, G. (1995). *The Impact of Architectural Trends on Operating System Performance*, *29*.

[4] Jonathan, L. (2001). Kqueue — a generic and scalable event notification facility. *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (pp. 141–153). Berkeley, CA, USA, USENIX Association.

[5] Davide, L. (2002). /dev/epoll home page. From http://www.xmailserver.org/linux-patches/-nio-improve.html

[6] Mark, H. L., John, D. V., David, N. C., Darryl, E. H., & Steven, R. W. (April 24, 2001). Input/output completion port queue data structures and methods for using same. *US Patent 6,* 223, 207.

[7] John, B. G. (1975). Structured exception handling. *Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages* (pp. 204–224).

[8] Tahina, R., Gabriel, D. R., & Xavier, L. (2012). A mechanized semantics for C++ object construction and destruction, with applications to resource management. *ACM SIGPLAN Notices*, *47*, 521–532.

[9] John, O. (1996). Why threads are a bad idea (for most purposes). *Proceedings of the 1996 Usenix Annual Technical Conference*: *Vol. 5.* San Diego, CA, USA.

[10] William, K. J. (2007). An introduction to Libaio. From http://www.morphisms.net/~wkj/software/-libaio/intro.pdf

[11] Ryan, C., & Eddie, K. (2005). Making events less slippery with eel. *HotOS*.

[12] Jeremy, C., *et al.* (2003). Why events are a bad idea (for high-concurrency servers). *HotOS*, 19–24,

[13] Hugh, C. L., & Roger, M. N. (1979). On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, *13(2),* 3-19,

[14] Frank, D., Nickolai, Z., Frans, K., David, M., & Robert, M. (2002). Event-driven programming for robust software. *Proceedings of the 10th Workshop on ACM SIGOPS* (pp. 186–189).

[15] Shivakant, M., & Yang, R.-G. (1998). Thread-based vs event-based implementation of a group communication service. *Proceedings of International Symposium on Parallel Processing* (pp.

0398–0398).

[16] David, P., Tim, B., Ashif, H., Peter, B., Amol, S., & David, R. C. (2007). Comparing the performance of web server architectures. *ACM SIGOPS Operating Systems Review*, *41*, 231–243.

[17] Nickolai, Z., Alexander, Y., Frank, D., Robert, M., David, M., & Kaashoek, M. F. (2003). Multiprocessor support for event-driven programs. *Proceedings of USENIX Annual Technical Conference, General Track* (pp. 239–252).

[18] Philipp, H., & Martin, O. (2006). Event-based programming without inversion of control. *Modular Programming Languages*, 4–22.

[19] Srinivas, S., Brett, K., Richard, M., Surendar, C., & Peter, K. (2006). Thread migration to improve synchronization performance. *Proceedings of Workshop on Operating System Interference in High Performance Applications*.

[20] Peng, L., & Steve, Z. (2007). Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *Proceedings of ACM SIGPLAN Notices*: *Vol. 42* (pp. 189–199).

[21] Philipp, H., & Martin, O. (2009). Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, *410(2)*, 202–220.

[22] Ahmad, M., Syed, I. R., & Syda, F. (2014). Thread models semantics: Solaris and Linux M:N to 1:1 thread model.

[23] Nathan, J. W. (2002). An implementation of scheduler activations on the NetBSD operating system. *Proceedings of USENIX Annual Technical Conference on FREENIX Track* (pp. 99–108).

[24] Thread (computer science). From http://en.wikipedia.org/wiki/Thread_(computer_science)

[25] Fibers (windows). From: http://msdn.microsoft.com/en-us/library/windows/desktop/ms682661.aspx

[26] Rob, Von B., Jeremy, C., Feng, Z., George, C. N., & Eric, B. Capriccio: Scalable threads for internet services. *Proceedings of ACM SIGOPS Operating Systems Review*: *Vol. 37* (pp. 268–281).

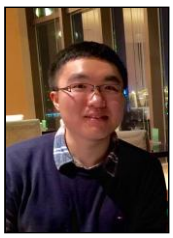
[27] Neil, C. C. B. (2007). C++ CSP2: A many-to-many threading model for multicore architectures. *Communicating Process Architectures 2007: WoTUG-30*, 183–205.

[28] Jeffrey, F., Rupak, M., & Todd, M. (2007). Tasks: language support for event-driven programming. *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (pp. 134–143).

[29] Frank, S. De B., Dave, C., & Einar, B. J. (2007). A complete guide to the future. *Programming Languages and Systems*, 316–330.

[30] Stefan, T., & Steve, V. (2010). Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, *14(6)*, 80–83.

[31] Richard, B., & Andrew, C. (2013). The evolution of the .NET asynchronous API. *Proceedings of Asynchronous Programming with .NET* (pp. 7–30).



Lu Gong was born in 1989 in Wuhan, China. He received his B.E. and M.E. degrees in computer science in Shanghai Jiao Tong University, China.

His research interests include system software, security and cloud computing. He once worked as a software development intern in Aliyun Computing Co., Ltd. He has been working as a software engineer in Google, Shanghai in Aug. 2015.

Youkai Sun was born in 1974 and got the bachelor degree in the China University of Petroleum in Dongying, China in 1995 majoring in applied electronic technology. His mainly research areas include system management, cloud computing and storage. He is working with the Shengli Geophysical Research Institute to apply could computing in seismic data processing and work as a system manager for Seismic Data Center.

Tao Dong was born in 1972 and got the bachelor degree in the China University of Petroleum (UPC). He works in Shengli Geophysical Research Institute SINOPEC, as a senior engineer. His mainly research areas include the applications of high performance computing in Shengli Geophysical Research Institute.

Zhanqiang Li was born in 1970 and got the bachelor degree of software engineering in Northwest University (NWU). He works in Shengli Geophysical Research Institute SINOPEC, as a senior engineer. His mainly research areas include the applications of high performance computing in Shengli Geophysical Research Institute.