Electronic Preprint for Journal of Information Processing Vol.30

Regular Paper

ComposableThreads: Rethinking User-level Threads with Composability and Parametricity in C++

Wataru Endo[1,a)] Shigeyuki Sato[2,b)] Kenjiro Taura[2,c)]

Received: May 1, 2021, Accepted: December 3, 2021

Abstract: User-level threading or task-parallel systems have been developed over decades to provide efficient and flexible threading features missing from kernel-level threading for both parallel and concurrent programming. Some of the existing state-of-the-art user-level threading libraries provide interfaces to customize the implementation of thread scheduling to adapt to different workloads from both applications and upper-level systems. However, most of them are typically built as huge sets of monolithic components which achieve customizability with additional costs via concrete C APIs. We have noticed that the zero-overhead abstraction of C++ is beneficial for assembling flexible user-level threading in a clearer manner. To demonstrate our ideas, we have implemented a new user-level threading library ComposableThreads which provides customizability while minimizing the interfacing costs. We show that the users can pick up, insert, or replace the individual classes of ComposableThreads for their own purposes. ComposableThreads offers several characteristic abstractions to build high-level constructs of user-level threading including suspended threads (one-shot continuations) and lock delegators. We evaluate both the customizability and performance of our runtime system through the microbenchmark and application results.

Keywords: user-level threading, fibers, work stealing, zero-overhead abstraction, composability, lock delegation

1. Introduction

As multi-core processors have become dominant in commodity computer systems, it is increasingly important to use multithreading to accelerate programs. Most of the modern operating systems provide multithreading interfaces to user programs, e.g., POSIX threads, which internally schedules threads inside the kernel. Because kernel-level threading suffers from substantial overhead via system calls, it is also common to develop user-level threading runtime systems to handle fine-grained parallelism more flexibly.

There are many user-level threading systems that support generic threading [16], [19], [20], [26]. It is also known that specialized systems provide better performance in specific cases. To fill the gap of these, Argobots [23] was designed as a stackable scheduler that can serve as a low-level threading and tasking framework for both of the scenarios. Instead of providing a uniform scheduling policy such as random work stealing, Argobots enables users to customize the scheduling policy for their own purposes.

In addition to the scheduling policy, we have noticed other demands to customize the default threading framework. First, threading frameworks should provide structured components that facilitate implementing different locking schemes because no locking scheme systematically performs the best in all cases [10]. However, existing threading libraries tend to hide their components for implementing locking in the implementation details and merely provide monolithic interfaces hard to customize. Second, to improve the concurrency of I/O operations, it is common to allocate a dedicated polling thread on top of user-level threading systems, where we require suspending and restarting it (as in Ref. [7]). Existing threading libraries provide condition variables for this purpose, but they are tightly coupled with mutexes and add the synchronization overhead when the users implement their own synchronization method for I/O delegation apart from the mutexes. Third, programmers demand the profiling and logging of user-level threading systems with minimal overhead because the total performance of applications can suffer considerably from the underlying threading layers. Although the hooks of injected function pointers enable us to implement threading systems to be customizable, the function-pointer-call overhead is not negligible particularly when we would like to observe behaviors as close to the original execution as possible. Therefore, existing threading systems often contain a bunch of C preprocessor macros and conditionals for profiling and logging with minimal overhead, making the code hard to customize. Finally, user-level threading can be extended to distributed-memory computing, namely, distributed work stealing (e.g., Ref. [1]). Distributed work stealing has the possibility of sharing many components with ordinary shared-memory implementations, but the actual runtime systems are entirely redeveloped in most cases.

We think that these problems conceptually arise from the same cause: most of the existing threading systems cannot provide useful abstractions to users without costs because they are hard-

[1] No affiliation. This work was done at Graduate School of Information Science and Technology, The University of Tokyo
[2] Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] endo.wataru@gmail.com
[b)] sato.shigeyuki@mi.u-tokyo.ac.jp
[c)] tau@eidos.ic.i.u-tokyo.ac.jp

coded in the C programming language. Since customizing a library inherently requires higher-order functions to be capable of changing the behavior, existing threading systems either use function pointers with additional runtime costs or preprocessor magic with production costs. Fortunately, generic programming in C++ enables us to develop zero-overhead abstractions demanded in threading systems. Although there are some threading systems written in C++ such as Intel TBB [11], they often focus on the productivity in developing applications and not on that in developing threading systems. Our main concern is the following research question: *how can we design a composable and parametric user-level threading system without interfacing costs?*

To answer this question, we have designed and implemented a user-level threading library ComposableThreads in C++. ComposableThreads is a header-only library written in C++ which extensively uses template meta-programming for customizability. ComposableThreads is *composable* so that we can easily extend, specialize, and exchange its components for different purposes and platforms. The zero-overhead abstraction of C++ enables us to create customizable threading libraries with minimal overhead. Each component of ComposableThreads is also designed to statically check the types utilizing C++'s features.

Our main contributions are summarized as follows:

- We have designed and implemented a user-level threading library ComposableThreads focusing on composability and customizability (Sections 3 and 4). The source code of ComposableThreads is publicly available online [*1].
- We experimentally demonstrate that our library can achieve comparable performance to the other user-level threading libraries (Section 5).
- We give an actual case of extensive use of ComposableThreads and show how well ComposableThreads works for developing an elaborate system (Section 6).

2. Notions in User-level Threading and C++

There are two levels of parallelism in user-level threading libraries: a (user-level) thread and a worker. User-level threads (ULTs), also called fibers, are lightweight threads under cooperative scheduling managed by user-level runtime systems. Each user-level thread has its own call stack. A worker (or an execution stream) is an object associated with a kernel-level thread that concurrently schedules multiple ULTs.

User-level threading systems depend on the capability of user-level context switching. Context switching transforms the current state into a *context*, which can be a single C pointer to the top of the call stack. The context enables us to resume itself on the current thread, but it is insufficient for handling ULTs.

In this paper, we define a *continuation* as the reference to a suspended ULT. Continuations allow accessing the saved contexts of the corresponding ULTs. We only consider one-shot continuations, which can be resumed only once, because it is costly to resume the same continuation multiple times in C or C++.

In this paper, we assume APIs of ULTs shown in **Fig. 1**, which are partly based on the C++ standard threading but designed for programming with explicit context switching and delegation.

```
class thread { // move-only
public:
  template <typename Func>
  static thread fork(Func&& func);
  void join();
};
namespace this_thread {
  void yield(); // Explicit context switching
  void exit();
}
// Similar to std::mutex and std::
    condition_variable
class mutex;
class condition_variable;
class barrier;
// For delegation
template <typename Consumer> class delegator;
```

Fig. 1 ULT APIs in C++.

```
template <typename R, typename F, typename... Args>
concept returned_from_invocable =
  invocable<F, Args...> &&
  convertible_to<invoke_result_t<F, Args...>, R>;
template <typename Func, typename... Args>
concept boolean_invocable =
  invocable<Func, Args*...> &&
  boolean<invoke_result_t<Func, Args*...>>;
```

Fig. 2 Concepts for special kinds of function types.

This paper also assumes readers' familiarity with C++ template programming. In particular, we use the concepts [25] of C++20 to describe the interfaces for templates. **Figure 2** shows concepts of special kinds of function types.

3. Design of ComposableThreads

3.1 Overall Design

The design of ComposableThreads has two key features:

- Policy-based parametrization [2] for everything;
- Higher-order interfaces at every level.

The former feature means that all the main components (shown in **Fig. 3**) are defined as class templates parametrized with policies, which are instances to implement code fragments. This policy-based design significantly facilitates extending, specializing, and exchanging the implementations of most of the components for different purposes and platforms. This parametrization is a zero-overhead abstraction because template parameters are fixed at compile time.

The latter feature means that both high-level operations exposed to users and low-level operations to implement high-level ones are defined as higher-order functions (or corresponding classes). Unlike common threading libraries, which provide first-order APIs except for thread creation, ComposableThreads provides the higher-order API of delegation in the form of delegators, which take user-defined code and delegate its execution to the internals of ComposableThreads. By designing most of the internal interfaces as higher-order ones, ComposableThreads can handle delegation uniformly with other threading functionality.

The following two subsections describe these key features more concretely, and then the later subsections describe how ComposableThreads deals with the notions of user-level threading.

[*1] https://github.com/endowataru/composablethreads

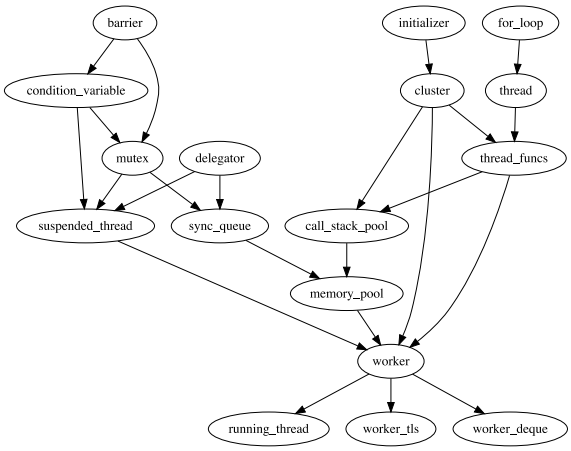

Fig. 3 Dependency graph of the main component classes in ComposableThreads.

3.2 Policy-based Design in C++

ComposableThreads extensively employs policy-based design [2], one of the generic programming techniques in C++. There is a default policy class that can generate the default user-level threading scheduler:

```
struct default_sct_policy {
  template <typename P>
  using log_aspect_t = basic_log_aspect<P>;
  template <typename P>
  using context_policy_t = context_policy<P>;
  template <typename P>
  using worker_deque_t = chaselev_worker_deque<P>;
  template <typename P>
  using worker_t = basic_worker<P>;
  template <typename P>
  using task_pool_t = basic_sct_task_pool<P>;
  template <typename P>
  using thread_t = basic_thread<P>;
  template <typename P>
  using mutex_t = basic_sct_mcs_mutex<P>;
  template <typename P>
  using scheduler_t = basic_sct_scheduler<P>;
  // ...
};
```

For example, the internal logging of the threading components can be controlled via `log_aspect_t` in the policy class. Parameterization through template meta-programming brings zero-overhead instrumentation of the components without the heavy use of preprocessor conditionals because calling the functions in the policy class is processed at compile time and can be completely inlined. This approach also applies to performance profiling.

To customize some of the classes, users can define a derived policy class:

```
struct my_sct_policy : cmpth::default_sct_policy {
  template <typename P>
  using worker_deque_t = my_worker_deque<P>;
};
```

With this derived policy, a new threading module can be instantiated at compile time:

```
using my_sct_itf = cmpth::sct_itf<my_sct_policy>;
my_sct_itf::thread t(f);
t.join();
```

The rest of this paper explains the components as if they are not using policy classes for brevity. The real implementation provides most of the components as template classes, which can change their policy classes.

3.3 Higher-order Context Switching

Context switching is the core functionality of user-level threading. Because of cooperative scheduling, every ULT API (not limited to `yield()`) implicitly involves context switching. Because most of the APIs are first-order operations, they are commonly implemented with lower-level first-order ones. In contrast, ComposableThreads implements context switching with lower-level higher-order operations, which contributes to making it so modular and compact.

Figure 4 shows the context-switching functions of ComposableThreads. The important difference from ordinary implementations is that these functions take a user-defined function called on top of the call stack of another thread. This is similar to *call with current continuation* of Scheme programming language [22], but this context switching is a more low-level functionality for efficient threading.

The idea itself, calling the function on another user-level context in C/C++, is not novel; e.g., `ontop_fcontext()` in Boost.Context [15]. However, such higher-order operations are merely optional and provided for specific purposes in existing threading libraries. Our key observation is that higher-order context-switching primitives are *practically inevitable* for efficient parallel implementation. This is because context switching (i.e., taking the current continuation) and other operations with the continuation should be atomically done. Although it is still possible to guard the resources for saving continuations via mutexes, for example, such a method degrades the performance because it requires hardware atomic instructions by nature.

Instead of such costly interactions between threads, delegating operations that eventually involve context switching to threads is simpler and more efficient. Moreover, this delegating higher-order primitive naturally leads to the implementation of the delegation API of ULTs. Everything thus becomes uniformly connected.

Table 1 and **Table 2** show the comparison of context switching functionalities in different threading systems. For higher-order switching functions provided by ComposableThreads, which take user-defined functions executed on top of another call stack, other libraries may or may not support them. In addition, as far as we know, no other threading systems support `cond_swap_context()`, which is useful for implementing mutexes on user-level threading described later. ComposableThreads does not support first-order functions without a function argument, but they can be emulated by passing a function that does nothing in its body.

In detail, there are five context switching primitives:

- `save_context()` saves the current context and calls the specified user-defined function with the saved context on a new call stack area. If this function normally returns, the saved context of the original thread is resumed.
- `restore_context()` abandons the current context and calls the specified user-defined function on top of the next thread's context. After that, it resumes the context of the

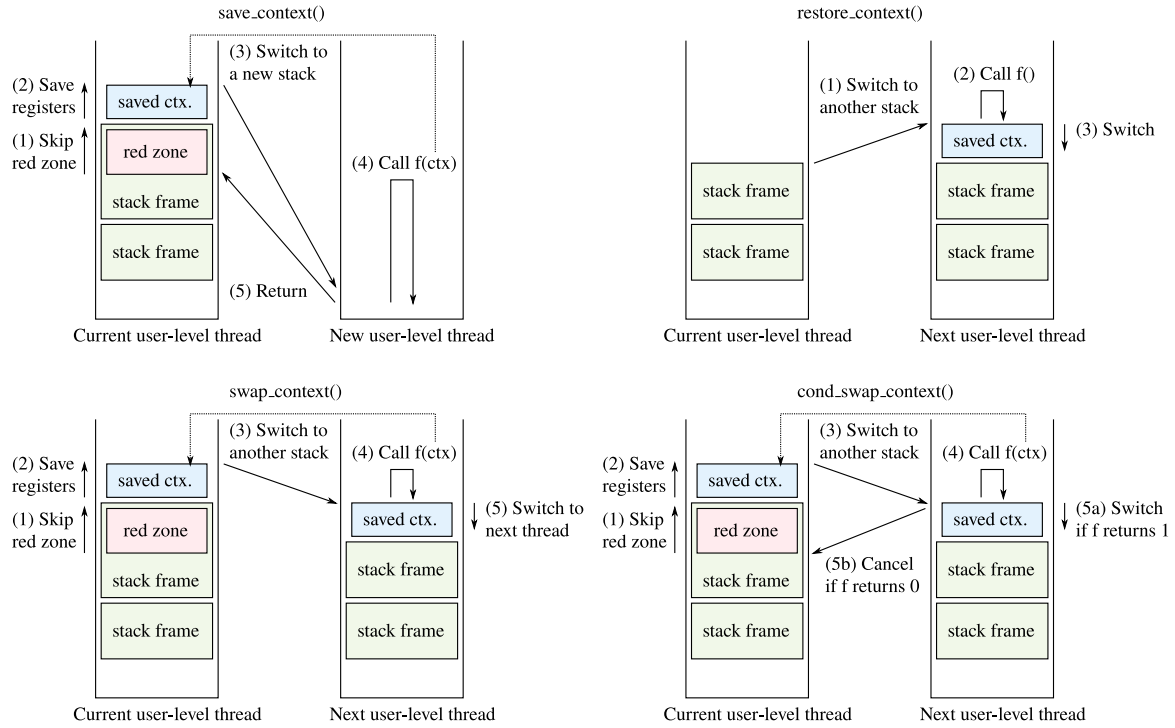

Fig. 4 Four context switching primitives of ComposableThreads. Each user-level thread has its own call stack with multiple stack frames. A *red zone* is the special area of a call stack that may be used by the program without adjusting the stack pointers.

Table 1 Support of higher-order delegating context switching functions in user-level threading systems.

ComposableThreads	`save_context()`	`swap_context()`	`restore_context()`
MassiveThreads [19]	Not supported [*1]	`myth_swap_context_withcall()`	`myth_set_context_withcall()`
Boost.Context [15]	Not supported	`ontop_fcontext()`	Not supported
Argobots [23]	`init_and_call_fcontext()`	Not supported	Not supported
libfibre [3], [13]	`stackDirect()`	`stackSwitch()`	Not supported

[*1] This is achieved via the combination of myth_make_context_empty() + myth_swap_context_withcall(). This functionality is important for MassiveThreads because it employs a work-first policy.

Table 2 Support of first-order context switching functions in user-level threading systems.

ComposableThreads	`make_context()`	`restore_context()` + empty func.	`swap_context()` + empty func.
MassiveThreads [19]	`myth_make_context_voidcall()`	`myth_set_context()`	`myth_swap_context()`
Boost.Context [15]	`make_fcontext()`	Not supported	`jump_fcontext()`
Argobots [23]	`make_fcontext()`	`take_fcontext()`	`jump_fcontext()`
libfibre [3], [13]	`stackInit()`	Not supported	Not supported

next thread.

- `swap_context()` saves the current context and calls the user-defined function with the saved context on top of the context of the next thread.
- `cond_swap_context()` saves the current context and calls the user-defined function as in `swap_context` first. The difference is that it can change the next execution path by the return value of the user-defined function. In other words, this function *conditionally* swaps the context depending on the function result.
- `make_context()` (not shown in Fig. 4) prepares a new context for the resumption. This function does not switch the current context by itself and returns the newly created context.

3.4 States of ULTs

The states of the ULTs are mapped to different types. When a ULT is initialized, its type is `call_stack` because it does not contain any context. When one of the workers start executing or resumes the ULT, the ULT is converted to the running ULT (`running_thread`) of the corresponding worker. If the worker switches to another ULT or the scheduler loop, its running ULT is extracted as a `continuation`.

3.5 Workers

The interface of ComposableThreads exposes workers for flexibility as Argobots does. In ComposableThreads, one worker consists of the following components:

- *A scheduler context* that represents the context of the scheduler. It is initialized as a null context. When the scheduler loop saves its continuation for resuming a ULT, the scheduler context is saved to this member.
- The pointer to *a running ULT*. It is initialized as a dummy pointer representing the scheduler thread.
- *A worker deque* that pools a set of resumable ULTs. Because resumable ULTs must be suspended, each worker deque

stores a list of continuations.
Because the worker itself does not need to be aware of the underlying real thread, it is not included as a data member.

In addition, users can retrieve *the current worker thread* by calling `worker::get_cur_worker()`. This is internally implemented as a (kernel-level) thread-local storage shared by all of the workers. Most of the methods of the worker class need to be called from the associated kernel-level thread. For debugging, their instances can be checked at runtime. One of the exceptions is a thief function that steals the ULT from a remote worker.

Figure 5 shows the function dependency graph related to the worker class. Based on the context switching functions, the worker class implements several functions:

- `wk.suspend_to_new()` suspends the current context and calls the specified function at the specified call stack. The saved continuation is passed as an argument of the specified function. This call stack is converted to a thread, which is registered as the running task of the worker. This method internally calls `save_context()`.
- `wk.exit_to_cont()` abandons the current context and calls the specified function at the specified continuation. After the specified function returns, the thread of the specified continuation is resumed and registered to the worker as its running task. Because this function does not save the current context, it is mapped to `restore_context()`.
- `wk.suspend_to_cont()` suspends the current context and calls the specified function at the specified continuation. This is implemented on `swap_context()`.
- `wk.cond_suspend_to_cont()` suspends the current context, calls the specified function at the continuation, and then conditionally resumes to the continuation. This method uses `cond_swap_context()` for conditional context switching.

These function calls can be used for implementing the other functions:

- `wk.execute()` resumes the specified continuation from the scheduler thread. This method calls `wk.suspend_to_cont()` with the function which stores the previous context to the scheduler context.
- The functions `wk.X_to_sched()` first pop the scheduler context and then call `wk.X_to_cont()` with the continuation of the scheduler context.

3.6 Suspended Threads

A suspended thread (`suspended_thread`) is an object that holds a continuation, which we introduce to implement variants of synchronization objects. While a continuation is a low-level object irrelevant to workers, suspended threads implicitly access the workers to push and pop the ULTs. This feature extends the synchronization mechanism called *uncond* variables introduced by Fukuoka et al. [8] for implementing software communication offloading with a ULT library MassiveThreads.

The interface of suspended threads is shown in **Fig. 6**. A suspended thread is a first-class move-only object. The move-only constraint guarantees suspended threads to be one-shot continuations, while it allows us to pass them to functions as arguments safely and efficiently. The default constructor initializes it as the

```
class suspended_thread {
public:
  suspended_thread();
  ~suspended_thread();
  suspended_thread(suspended_thread&) = delete;
  suspended_thread&
    operator=(const suspended_thread&) = delete;
  suspended_thread(suspended_thread&&);
  suspended_thread& operator=(suspended_thread&&);
  // Wait
  template <typename Func, typename... Args>
  void wait_with(Args*... args)
    requires boolean_invocable<Func, Args*...>;
  // Notification
  void notify();
  void enter();
  // Swap (Wait & Notification)
  void swap(suspended_thread& restored);
  template <typename Func, typename... Args>
  void swap_with(suspended_thread& restored,
    Args*... args)
    requires boolean_invocable<Func, Args*...>;
  // Observer
  explicit operator bool() const;
};
```

Fig. 6 Interface of class suspended_thread.

empty state, which is not associated with any thread. When the method of a suspended thread saves the continuation of the current thread, it transitions to a non-empty state associated with the continuation.

The methods of a suspended thread `sth` are described as follows:

- `sth.wait_with()` suspends the current thread and saves its continuation to the suspended thread. It takes a user-defined function which is invoked on top of the call stack of the resumed thread with the specified pointer arguments. The returned boolean from the function instructs whether the worker actually saves the continuation or cancels it. If it is canceled, the suspended thread remains empty.
- `sth.notify()` resumes the continuation associated with the suspended thread. It will put the continuation into one of the worker deques, which is (usually) owned by the caller's worker. `sth.enter()` also resumes the continuation and works similarly as `sth.notify()`, but it differs in the actual behavior because it saves the continuation of the current thread, put it into the deque, and then resumes the continuation associated with `sth`. Both `sth.notify()` and `sth.enter()` are interchangeable if performance does not significantly matter.
- `sth.swap_with()` first suspends the current thread and saves its continuation as `sth.wait_with()` does. It also resumes the continuation associated with the suspended thread. The suspension can be canceled in the same way as defined in `sth.wait_with()`. `sth.swap()` is an alternative of `sth.swap_with()` without the specified function.
- The observer function (i.e., casting to `bool`) returns the state of the suspended thread. If a continuation is associated with the suspended thread, the observer returns `true`.

Note that higher-order APIs `sth.wait_with()` and `sth.swap_with()` are zero-overhead abstractions because they take the code of argument functions via template parameters at compile time.

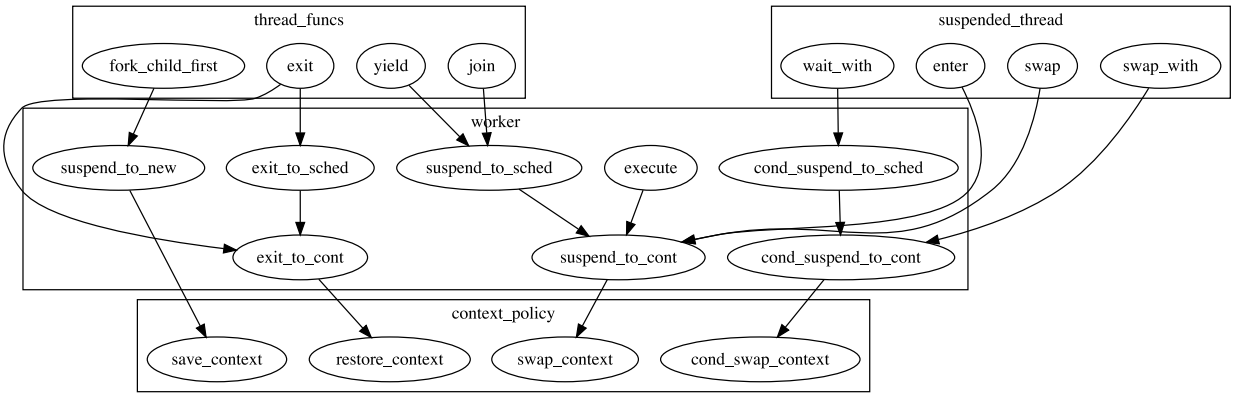

Fig. 5 Function dependency graph related to the worker class.

3.7 Mutexes

Mutexes and condition variables have been implemented on suspended threads. In addition to ordinary methods such as `lock()` and `unlock()`, ComposableThreads' mutexes introduce `unlock_and_wait()` which saves the continuation to a suspended thread while it also unlocks the mutex.

3.8 Delegators

A delegator is a class that synchronizes between producer and consumer threads for a shared resource. The design of delegators in ComposableThreads is based on suspended threads. Lock delegation has many implementation options and is often tightly coupled with threading as in mutexes because lock delegation is a variant of realizing mutual exclusion. To separate the implementation of delegators from the underlying threading system, suspended threads are useful again because they enable zero-overhead manipulation of continuations while hiding the detail of the threading system. As described in Section 4.5 later, the examples of delegator implementations introduce a helper ULT for lock delegation to maximize concurrency, which requires suspended threads to efficiently handle the helper ULT.

We describe the interface of delegators here. To use delegators, the users define a consumer class as follows:

```
class my_consumer {
  using exec_ret = tuple<bool, suspended_thread>;
public:
  using delegated_func_type = /* ... */;
  exec_ret execute(delegated_func_type&);
  suspended_thread progress();
  bool is_active() const;
};
```

It is required for the consumer object `c` to have the following methods:

- `c.execute()`: This method executes the delegated function specified as the argument. The first element of the return value is `true` if the delegated function is executed by calling `c.execute()`. Otherwise, the delegator will try to execute the delegated function again later. The second element of the return value is a suspended thread which is resumed by the delegator later. Instead of waking it up inside `execute()`, returning this suspended thread may improve the performance because the worker thread of the delegator can directly switch to the returned suspended thread.

```
template <typename Consumer, typename Func>
concept on_delegator_delegate =
  returned_from_invocable<
    suspended_thread*, Func,
    typename Consumer::delegated_func_type&>;
template <typename Func>
concept on_delegator_imm_exec =
  returned_from_invocable<
    tuple<bool, suspended_thread*>, Func>;

template <typename Consumer>
class delegator {
public:
  template <typename DF>
  bool lock_or_delegate(DF&& delegate_func)
    requires on_delegator_delegate<Consumer, DF>;
  template <typename IEF, typename DF>
  bool execute_or_delegate(
    IEF&& imm_exec_func, DF&& delegate_func)
    requires on_delegator_imm_exec<IEF> &&
             on_delegator_delegate<Consumer, DF>;
  void lock();
  void unlock();
  void unlock_and_wait(suspended_thread& wait_sth);

  void start_consumer(Consumer&);
  void stop_consumer();
};
```

Fig. 7 Interface of template class delegator.

- `c.progress()`: The progress function of the consumer. The return value of this method works in the same way as `c.execute()`.
- `c.is_active()`: This method returns `true` if the delegator needs to invoke the progress of the consumer.

The interface of lock delegation producers is shown in **Fig. 7**. The producer methods of a delegator `d` are described as follows:

- `d.lock_or_delegate()` tries to lock the delegator first. If it succeeds, it will immediately return to the caller returning `true`. Otherwise, it calls `delegate_func` to delegate a function to the consumer and returns `false`. `delegate_func` can return a pointer to a suspended thread where the continuation of the caller thread is saved. If this returned pointer is null, the continuation is not saved. If lock acquisition succeeds, the caller thread needs to unlock the same delegator after it completes its critical section. If not, because the caller thread is not locking the delegator, it shall not unlock it either.
- `d.execute_or_delegate()` first tries to lock the delegator

as `d.lock_or_delegate()` does. If it succeeds, it executes `imm_exec_func` on the same thread, unlocks the delegator, and returns `true`. If not, `delegate_func` is called for delegation and returns `false`. This method is convenient when a caller can encapsulate the critical section as a function object.

- `d.lock()` waits for the lock acquisition of a delegator as in a normal mutex and `d.unlock()` releases the lock. `d.unlock_and_wait()` works in the same way described in the mutex interface. With these methods, it is apparent that delegators simply extend the interface of mutexes.

The consumer thread for delegator `d` can be started by `d.start_consumer()` and stopped by `d.stop_consumer()`.

The delegator for the specific consumer `my_consumer` is instantiated as `delegator<my_consumer>`.

4. Implementation of ComposableThreads

In this section, we describe the implementation of ComposableThreads based on template metaprogramming. We follow MassiveThreads [19] for the core techniques to implement threading. Here, we focus on how our implementation is composed.

4.1 Context Switching for the x86-64 Architecture

Figure 8 shows the interface of the context switching policy. Because we tried to hide architecture-specific issues from the users of this policy, we suppose that this context switching policy can be replaced easily with other implementations.

There are two technical points in this interface definition. First, we can see that all of the parameters passed to the user-defined function are pointers. This is because pointers can be efficiently mapped to hardware integer registers in most of the processor architectures. It is possible to pass multiple pointer parameters as in MassiveThreads [19], and also, there is no limitation of its count owing to variadic templates. Second, in context switching, it is possible to pass a pointer wrapped as `transfer<T*>`. The return value of the user-defined function is transferred to the next resumed context. We know that the same technique is used in Boost.Context. Our interface adds the type safety because each context switching call requires that the equality of `T` between `context<T*>` and `transfer<T*>`.

For the x86-64 architecture, we define the data structures for context switching as follows:

```
struct context_frame {
  void* rbp; void* rip; void* rsp;
};
struct x86_64_context_policy {
  template <typename T>
  struct context { context_frame* p; };
  template <typename T>
  struct transfer { T p0; };
  template <typename T>
  struct cond_transfer { T p0; int64_t flag; };
  // other members...
};
```

To save the context efficiently in C or C++, it is necessary to write architecture-specific assembly code. We use inline assembly as in MassiveThreads to minimize the interfacing costs of context switching. We now show the function

```
struct context_policy {
  template <typename T> struct context;
  template <typename T> struct transfer;
  template <typename T> struct cond_transfer;
  template <typename Func,
    typename T, typename... Args>
  static transfer<T*> save_context(
    void* sp, size_t size, Args*... args)
    requires returned_from_invocable<
      transfer<T*>, Func, context<T*>, Args*...>;
  template <typename Func,
    typename T, typename... Args>
  static void restore_context(
    context<T*> ctx, Args*... args)
    requires returned_from_invocable<
      transfer<T*>, Func, Args*...>;
  template <typename Func,
    typename T, typename... Args>
  static transfer<T*> swap_context(
    context<T*> ctx, Args*... args)
    requires returned_from_invocable<
      transfer<T*>, Func, context<T*>, Args*...>;
  template <typename Func,
    typename T, typename... Args>
  static transfer<T*> cond_swap_context(
    context<T*> ctx, Args*... args)
    requires returned_from_invocable<
      cond_transfer<T*>, Func,
      context<T*>, Args*...>;
  template <typename Func,
    typename T, typename... Args>
  static context<T*> make_context(
    void* sp, size_t size, Args*... args)
    requires invocable<
      Func, transfer<T*>, Args*...>;
};
```

Fig. 8 Interface of a context policy class.

`save_context_void()` for the x86-64 architecture, the type-erased version of `save_context()`:

```
using save_func_t =
  transfer<void*> (*)(context<void*>, void*,
    void*, void*, void*, void*);
transfer<void*> save_context_void(
    void* sp, size_t size, save_func_t func,
    void* arg1, void* arg2, void* arg3,
    void* arg4, void* arg5) {
  transfer<void*> ret;
  // bind arguments to specific registers
  register void* arg4_r8 asm ("r8") = arg4;
  register void* arg5_r9 asm ("r9") = arg5;
  // adjust the new stack pointer
  int64_t new_sp =
    (reinterpret_cast<int64_t>(sp) & ~0xF) - 0x8;
  asm volatile (
    "movq   %%rsp, %%r15\n\t" // prepare for saving
    "andq   $-0x10, %%rsp\n\t" // align RSP
    "subq   $0x88, %%rsp\n\t" // skip red zone
    "pushq  %%r15\n\t" // save original RSP
    "leaq   1f(%%rip), %%rax\n\t" // calculate RIP
    "pushq  %%rax\n\t" // save resumed RIP
    "pushq  %%rbp\n\t" // save RBP
    "xchg   %%rdi, %%rsp\n\t" // swap stack ptrs.
    "pushq  %%r15\n\t" // save RSP to new stack
    "call   %%rbx\n" // call user-defined function
  "1:\n\t"
    "popq   %%rsp" // restore RSP
  : // output constraints
    "+b"(func), "+D"(new_sp), "=a"(ret.p0),
    "+S"(arg1), "+d"(arg2), "+c"(arg3),
    "+r"(arg4_r8), "+r"(arg5_r9)
  : // no input constraints
  : "cc", "memory", "%r10", "%r11",
    "%r12", "%r13", "%r14", "%r15" // clobber all
  );
```

```
  return ret;
}
```

This function allows the number of pointer parameters up to five. When exceeding, the wrapper function `save_context()` automatically packs and unpacks the rest of the parameters. The registers of the current context are saved as defined in `context_frame`. Then, the stack pointer is swapped with the next thread's context and the user-defined function is invoked.

The important difference from the implementation of MassiveThreads is that ours does not always save callee-saved registers but instead put every register into the clobbered list. This is because compilers can detect whether the program needs to save the registers or not. MassiveThreads' context switching pessimistically saves all of the callee-saved registers, but when they are not used in the user code, this is unnecessary. When we disable inlining around this function, it will safely save all of the callee-saved registers because the compiler knows that they must be saved for the callee.

Restoring the context is implemented with only three instructions:

```
using restore_func_t =
  transfer<void*> (*)(
    void*, void*, void*, void*, void*, void*);
void restore_context_void(
    context<void*> ctx, restore_func_t func,
    void* arg0, void* arg1, void* arg2, void* arg3,
    void* arg4, void* arg5) {
  register void* arg4_r8 asm ("r8") = arg4;
  register void* arg5_r9 asm ("r9") = arg5;
  asm volatile (
    "movq   %[ctx], %%rsp\n\t" // restore RSP
    "popq   %%rbp\n\t" // restore RBP
    "jmp    %%rbx\n\t" // call user-defined func.
  : "+b"(func)
  : [ctx] "g"(ctx.p), "D"(arg0), "S"(arg1),
    "d"(arg2), "c"(arg3),
    "r"(arg4_r8), "r"(arg5_r9)
  : "cc", "memory"
  );
}
```

This function simply restores the saved RSP and RBP and jumps to the user-defined function. We intentionally used `jmp` because the return address is already recorded on top of the call stack; it is the saved RIP of the resumed context (e.g., `1:` of `save_context_void()`). This is one of the hacks used in `ontop_fcontext()` of Boost.Context to reduce one jump instruction.

Why do we put the frame pointer RBP exceptionally on top of a saved context? It is because it contains the stack information for the function invoked on top of the context. Although it is possible to correctly run the x86-64 programs without adjusting frame pointers [*2], it still helps debugging tools and enables us to show backtraces correctly.

The remaining three functions are implemented in the combination of these techniques. For `cond_swap_context()`, we could not use the return address hack to implement conditional context switching.

[*2] -fomit-frame-pointer in GCC will skip adjusting the frame pointer.

4.2 Workers

We assume that the following members in the worker deque class:

```
class worker_deque {
public:
  void local_push_top(continuation cont);
  void local_push_bottom(continuation cont);
  continuation try_local_pop_top();
  continuation try_remote_pop_bottom();
};
```

Three local functions are called by the same worker thread. `try_remote_pop_bottom()` is called by thief worker threads.

Based on the worker deque class, the worker class is defined as follows:

```
class worker : public worker_deque {
  continuation sched_cont;
  running_thread cur_th;
  continuation local_pop_top() {
    if (auto c = this->try_local_pop_top())
      return c;
    else return move(sched_cont);
  }
public:
  static worker& get_cur_worker();
  template <typename Func, typename... Args>
  worker& suspend_to_cont(
    continuation next_cont, Args*... args)
    requires returned_from_invocable<
      worker&, Func, continuation, Args*...>;
  template <typename Func, typename... Args>
  worker& suspend_to_sched(Args*... args) {
      return this->template suspend_to_cont<Func>(
        this->local_pop_top(), args...);
  }
  // other members...
};
```

4.3 High-level Interface for Generic Threading

Creating and joining ULTs are considered primitive operations for user-level threading systems. In ComposableThreads, because the worker class already defines most of the features necessary for switching ULTs, the high-level functions for ULTs can be implemented as simple wrapper functions with resource management.

First, we show that `yield()` can be simply implemented as follows:

```
struct on_yield {
  worker& operator() (worker& wk, continuation c) {
    wk.local_push_bottom(move(c)); return wk;
  }
};
void this_thread::yield() {
  worker& wk = worker::get_cur_worker();
  wk.suspend_to_sched<on_yield>();
}
```

`yield()` relinquishes the current worker thread. Returning to the scheduler context implemented by `suspend_to_sched()` is almost what `yield()` internally does, but it is still possible to change minor behaviors such as where the continuation is pushed to. In this implementation, the continuation is pushed to the bottom of the worker deque (i.e., the thief side in work stealing), but this is not the only choice; we can provide other implementations by composing the primitive operations of the worker class.

Although the help-first version of fork can be simply translated

to an invocation of `local_push_top()`, it is considered difficult to implement the work-first version because it requires an efficient combination of context switching and the user-defined computation. In our composable approach, the work-first `fork()` can also be easily implemented as follows:

```
template <typename F>
struct on_fork {
  worker& operator() (
      worker& wk, continuation cont, F* func) {
    wk.local_push_top(move(cont));
    (*func)(); func->~F();
    this_thread::exit();
  }
};
template <typename Func>
thread thread::fork(Func&& func) {
  worker& wk = worker::get_cur_worker();
  call_stack stk = call_stack_pool::allocate(wk);
  thread ret{stk};
  using F = decay_t<Func>;
  F& sf = stk.construct<F>(forward<Func>(func));
  wk.suspend_to_new<on_fork<F>>(move(stk), &sf);
  return ret;
}
```

This function first allocates a call stack from the pool (if available) and then switches to the new stack via `suspend_to_new()`. Upon the new stack, the generated continuation is pushed to the worker deque. Then, the user-defined function object is called and destructed. Thanks to the generic programming techniques, it is observed that both the thread management and the user-defined computation are seamlessly combined as a simple code fragment.

Other functions including `exit()` and `join()` are implemented similarly. Since these functions need to change the control flow based on the ULT state, they are slightly more complicated than the previous ones.

4.4 Synchronization Objects

Mutexes can be implemented on suspended threads. Here, we show how suspended threads can be used to implement mutexes based on MCS locks [18]. We first define the queueing node class `mcs_node` and the members of the mutex class:

```
struct mcs_node {
  atomic<bool> ready; suspended_thread sth;
};
class mutex {
  atomic<mcs_node*> tail; mcs_node* head;
public:
  void lock(); void unlock();
  void unlock_and_wait(suspended_thread&);
};
```

Then, the lock method is defined as follows:

```
struct on_lock {
  bool operator() (worker& wk, mcs_node* prev) {
    prev->ready.store(true, memory_order::release);
    return true;
  }
};
void mutex::lock() {
  worker& wk = worker::get_cur_worker();
  mcs_node* cur = mcs_node_pool::allocate(wk);
  mcs_node* prev =
    tail.exchange(cur, memory_order::acq_rel);
  if (prev == nullptr)
    prev->sth.wait_with<on_lock>(wk, prev);
  head = cur;
}
```

This implementation first allocates a node and replaces the tail pointer with the allocated one. When there is no other thread locking (or waiting for locking) the same mutex, this method immediately succeeds. Otherwise, the current thread waits for the notification of the lock release. To accomplish this, the suspended thread can be used as the notification mechanism.

To ensure that another thread that will notify the waiting thread properly loads its continuation, this method sets the `ready` flag on the function passed to `wait_with()`. Because the interface of suspended threads allows user-defined functions for switching, synchronization objects can be implemented apart from the detail of other scheduling components.

The unlock method is also implemented with suspended threads:

```
void mutex::unlock() {
  worker& wk = worker::get_cur_worker();
  mcs_node* cur = head; head = nullptr;
  auto expected = cur;
  if (!tail.compare_exchange_strong(
      expected, nullptr, memory_order_release)) {
    while (!cur->ready.load(memory_order::acquire))
      /* busy wait */;
    cur->sth.enter();
  }
  mcs_node_pool::deallocate(cur);
}
```

This example uses `enter()` to wake up the waiting thread because it is usually preferable to execute the critical sections for the mutex in the same worker thread for reusing the caches, but it is technically possible to replace `enter()` with `notify()`.

`unlock_and_wait()` is implemented similarly to `unlock()`, but when the mutex is being unlocked, this method needs to atomically save the continuation for the next acquirer thread. To ensure that, the CAS operation for unlocking is issued inside the function passed to `wait_with()`. If the CAS operation succeeds, this worker thread can immediately switch to another thread. If it fails, this thread cancels the context switching and goes back to the previous function body by returning `false` in the passed function. The higher-order delegating nature of `wait_with()` offers the ability to efficiently implement this complex switching pattern.

With the above implementation of mutexes, condition variables can also be implemented easily with suspended threads:

```
class condition_variable {
  struct wait_entry {
    wait_entry* next; suspended_thread sth;
  };
  mutex wait_mtx; wait_entry* wait_cur;
public:
  void wait(unique_lock<mutex>& l) {
    wait_entry e;
    wait_mtx.lock();
    e.next = wait_cur; wait_cur = &e;
    l.unlock();
    wait_mtx.unlock_and_wait(e.sth);
    l.lock();
  }
  void notify_one() {
    lock_guard<mutex> lk{wait_mtx};
    if (wait_cur) {
      wait_cur->sth.notify();
      wait_cur = wait_cur->next;
    }
  }
```

```
};
```

It is necessary to correctly wait for the notification while unlocking the mutex for guarding the entries for waiting threads. This can be achieved by `unlock_and_wait()` implemented in mutexes.

The current implementation of barriers is a centralized barrier that serializes all of the threads accessing the barriers. Centralized barriers can be implemented on top of mutexes and condition variables.

4.5 Delegators

In our implementation of delegators, each delegator object has its own user-level thread which can execute critical sections independently from other threads. This strategy generalizes the software communication offloading technique proposed in the previous work [7]. Assume the case when the lock acquisition is contended:

(1) If the next acquirer thread needs to execute the critical section, it always needs to wait for the current lock owner.
(2) If the owner thread needs to execute the critical sections of other succeeding threads, the owner thread cannot return to its execution and causes starvation.

Introducing a helper thread naturally solves this problem by resuming the helper thread when necessary. Because this approach oversubscribes threads, it is only feasible with user-level threading.

When `start_consumer()` (described in Section 3.8) is called, the delegator forks a consumer thread and saves its continuation as a suspended thread for preparing the delegation. Producer threads accessing the delegator resume the consumer thread if necessary.

We provide two kinds of implementations: list-based (queue-based) or circular buffer-based implementations. The list-based implementation is similar to that of the mutexes presented above. Another implementation based on circular buffers imitates the method of Ref. [7]. These two implementations can be switched without changing the users' code because the delegator interface is defined to be independent of the actual scheduling policy.

4.6 Delegating I/O

As an example of using delegators, we have implemented a wrapper for the socket I/O functions. The delegation strategy is useful to handle I/O polling separately from the application threads.

Our I/O delegator wrapper uses the `epoll` functionality of Linux, which enables users to scale monitoring multiple file descriptors. When calling `socket()`, the wrapper registers the returned file descriptor to its epoll instance. The file descriptor is internally set to the non-blocking mode. When an application thread issues I/O operations via `read()` or other system calls, it may fail to execute the I/O operation immediately due to the error codes `EAGAIN` or `EWOULDBLOCK`. In that case, the application thread tries to start polling on a delegator thread.

The consumer class for I/O delegation is defined as follows:

```
class io_consumer {
  using sth_list = vector<suspended_thread>;
  unordered_map<int, sth_list> pends;
  size_t num; int epfd;
public:
  suspended_thread& make_pending(int fd) {
    return pends[f.fd].emplace_back();
  }
  struct delegated_func_type {
    int fd; suspended_thread sth;
  };
  tuple<bool, suspended_thread>
  execute(delegated_func_type& f) {
    make_pending(f.fd) = move(f.sth));
    ++num; return { true, nullptr };
  }
  suspended_thread progress() {
    suspended_thread ret_sth;
    epoll_event es[NUM_EVENTS];
    int r = epoll_wait(epfd, es, NUM_EVENTS, 0);
    for (int i = 0; i < r; ++i) {
      if ((es[i].ef & (EPOLLIN|EPOLLIN)) != 0) {
        auto& wake_list = pends[es[i].data.fd];
        for (auto& sth: wake_list) {
          --num;
          if (ret_sth) sth.notify();
          else ret_sth = move(sth);
        }
        wake_list.clear();
      }
    }
    return ret_sth;
  }
  bool is_active() const { return num > 0; }
};
```

`execute()` consumes the delegated operation for I/O, which simply pushes the continuation of the application thread. `progress()` is enabled when there are any pending I/O operations and invokes `epoll_wait()` with an immediate timeout (the fourth parameter). If there is an event notification on one of the file descriptors, all of the suspended threads are resumed. One of those resumed threads is returned directly as the return value of `progress()` enabling the implementation of delegators to suspend the delegator thread and switch to another immediately.

The producer class, which uses the consumer via a delegator and is directly accessed by application threads, is defined as follows:

```
class io_producer {
  io_consumer con;
  delegator<io_consumer> del;
public:
  ssize_t execute_read(int sockfd, /*...*/) {
    while (true) {
      ssize_t ret = read(sockfd, /*...*/);
      if (ret == -1 && (errno == EAGAIN
          || errno == EWOULDBLOCK))
        del.execute_or_delegate(
          [=, this] () {
            return &con.make_pending(sockfd); },
          [=] (delegated_func_type& f) {
            f.fd = sockfd; return &f.sth; });
      else return ret;
    }
  }
};
```

`execute_read()` wraps the original `read()` system call as an example. Other I/O functions are defined in the same manner. When the actual system call fails, the producer informs the delegator thread to wait for the corresponding file descriptor. There are two cases as described: whether the delegator thread is work-

ing or not. In either case, this wrapper function waits for the completion of this application thread adding its continuation to the pending list of the consumer object. This is accomplished by returning the address of the (empty) suspended thread object to the callee function implemented by the delegator class.

5. Evaluation

5.1 Experimental Settings

We have conducted two microbenchmarks to evaluate ComposableThreads. The first microbenchmark is the Fibonacci number calculation. The *fib(n)* microbenchmark calculates the n-th Fibonacci number in parallel to measure the performance of the primitive ULT features: creating and destructing ULTs.

The second microbenchmark is the fetch-and-add microbenchmark. This benchmark creates multiple ULTs that execute critical sections for incrementing the single shared integer variable. For mutual exclusion, we tried the experiments either with a mutex or a delegator. We measured the throughput with different numbers of threads. The number of ULTs executing critical sections is set to the same as the number of worker threads. We evaluated the list-based implementation of delegators.

To evaluate the performance of I/O delegation, we used the web server program of libfibre [3], [13], a user-level threading library focused on accelerating I/O operations. This program uses ULTs for multiplexing HTTP connections from clients. We modified the original program for ComposableThreads because libfibre has a different fiber interface from ours.

Table 3 shows the evaluation environment. We compared the performance of the microbenchmarks with MassiveThreads and Argobots. The experiments are conducted with almost the latest versions of these libraries: MassiveThreads v1.0 and Argobots with the commit hash 09ec354 [*3]. For the web server program, we have also evaluated the original program with libfibre with the commit hash 7367d83 [*4].

Argobots provides the ability to select the scheduling policies and pool implementations. For fair comparison, we set the schedulers to the random work stealing mode. We allocated a FIFO MPMC (multiple-producer multiple-consumer) worker deque for each worker thread.

To generate the load for the web server program, we used wrk [9], which was used in the paper of libfibre. Because our evaluation environment had two NUMA nodes, we allocated one NUMA node for the server and the other for the client. The client program creates 28 threads and communicates with the local server using 100 connections in total.

5.2 Microbenchmark Results: Fibonacci

Figure 9 shows the speedup comparison of calculating fib(45). All of the speedup values are relative to the sequential execution time of ComposableThreads (147.4 seconds). The performance of ComposableThreads was slightly slower than that of MassiveThreads with high core counts. With 56 worker threads, ComposableThreads was 21% slower than MassiveThreads. One of the possible reasons is that the default policy of worker deques in ComposableThreads, which can be replaced if necessary, uses a balancing technique in memory pools (similar to Ref. [17]) for reusing call stacks to avoid resource imbalance. This technique adds overhead especially with a large number of threads. Argobots could not execute fib(45) due to too much memory consumption. We think that there is a problem in the memory pool of Argobots for call stacks.

[*3] Committed on December 11, 2019.
[*4] Committed on December 17, 2019.

Table 3 Evaluation environment (Oakbridge-CX [24]).

CPU	Intel Xeon Platinum 8280 2.7 GHz (max. 4.00 GHz with Turbo boost) 26 cores × 2 sockets (Hyperthreading disabled)
Memory	192 GiB (DDR4)
OS	CentOS Linux release 7.6.1810 (Core)
Compiler	Intel C++ compiler 19.0.4.243

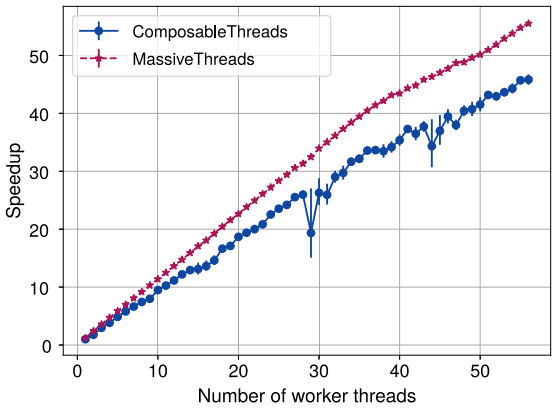

Fig. 9 Speedup comparison of fib(45).

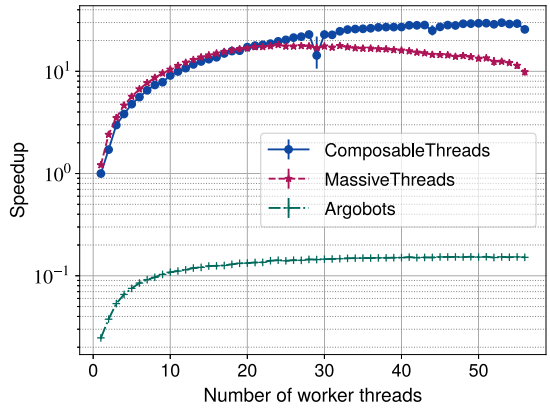

Fig. 10 Speedup comparison of fib(34). The Y-axis is logarithmic.

We also measured the execution time of fib(34) to compare with Argobots as shown in **Fig. 10**. The sequential time of ComposableThreads is 0.737 second, which is too small for scaling to many cores. The sequential performance of Argobots was 40.5 times slower than that of ComposableThreads. The parallel speedup of Argobots is also apparently limited compared to the other two systems. Although Argobots can work as a work stealing scheduler, we can see that it is not optimized for this purpose and exhibits a huge overhead for creating fine-grained threads. It is also observed that ComposableThreads performs slightly better than MassiveThreads with more than 30 cores in this configuration.

5.3 Microbenchmark Results: Fetch-and-add

Figure 11 shows the throughput of critical sections using a sin-

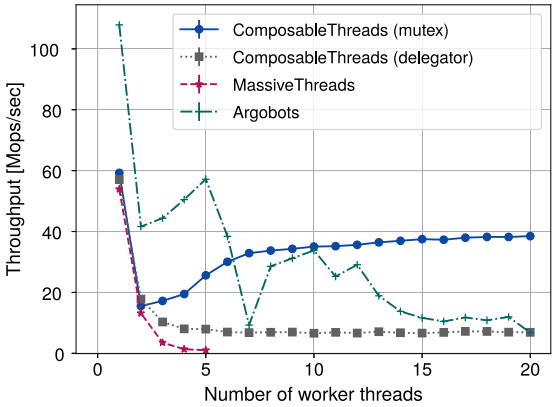

Fig. 11 Throughput of executing critical section for a mutex (or a delegator) in the fetch-and-add microbenchmark.

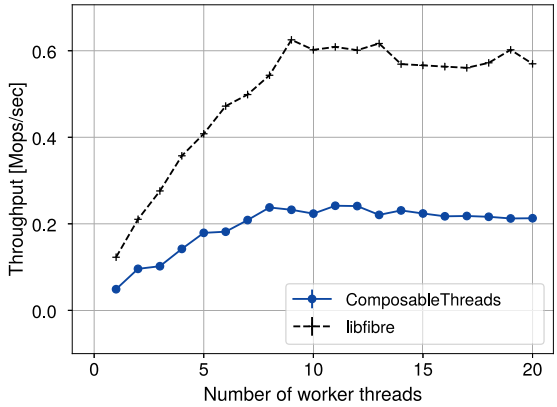

Fig. 12 Throughput of HTTP server benchmark.

gle mutex with varying numbers of threads. The throughput of the single-threaded case is the best in all of the systems because it can avoid synchronization between cores. In this microbenchmark, Argobots performs better than others with small core counts, but with more threads, ComposableThreads' mutexes provide the stable throughput. Because the mutex implementations of both Argobots and MassiveThreads are based on spinlocks, the performance in highly contended situations is degraded compared to our queue locking approach. Since the mutex of MassiveThreads scaled very poorly, we have omitted its result with more than six threads.

Figure 11 also shows the throughput with a lock delegator. We initially expected that introducing delegators would improve the throughput of mutexes, but currently, they do not achieve better performance using 3 threads or more. We are currently investigating this result and suspecting that the resource management for delegation entries is the bottleneck.

5.4 Application Benchmark Results: HTTP Server

Figure 12 shows the throughput of the HTTP server program. The original implementation with libfibre was faster than our modified version with ComposableThreads. Yet, our I/O delegator scaled the I/O throughput up to 4.87 times using 8 threads compared to the sequential one. From the result of the fetch-and-add microbenchmark, we think that the delegator mechanism was not the bottleneck of this benchmark. This result indicates room for improvement in the implementation of the I/O delegator wrapper.

We should note that libfibre can avoid busy waiting when all of the threads are waiting for I/O requests, whereas the current implementation of ComposableThreads cannot. Implementing an I/O-aware scheduling method for improving efficiency is left for future work.

6. Use Case: A Distributed Shared Memory Library

ComposableThreads was originally designed as the threading component of a distributed shared memory (DSM) library MENPS [5], [6] and later separated as a standalone library. DSM is a form of shared-memory systems implemented on top of distributed-memory machines. MENPS was implemented in about 50 k LOC of C++ and composed of multiple library components: the core library of DSM, the simple OpenMP runtime system, the inter-node communication library [7], and ComposableThreads. As in ComposableThreads, MENPS utilized policy classes for customizing its behavior. MENPS was able to select the underlying threading system at compile time through the policy classes from ComposableThreads, MassiveThreads, and kernel-level threading. ComposableThreads provided interface classes for ULT libraries and kernel-level threading that implement the threading functions listed in Fig. 1, which enabled us to easily switch the threading systems and debug the threading behavior of MENPS.

There were three reasons why we introduced a new threading library for building a DSM system. First, as described in Section 4.5, inter-node communication offloading is closely related to lock delegation, which necessitated the suspended threads as a lightweight context switching mechanism. Second, to build a purely library-based system without special compilers, MENPS was implemented as everything-shared DSM [4] which places not only global memory but also call stacks on the memory regions shared via DSM. It was necessary for MENPS to invoke special DSM operations for switching contexts from a normal call stack to a shared call stack placed on DSM (and vice versa), which could not be easily implemented in the existing libraries. To implement this, the OpenMP runtime system of MENPS picked up the context switching component of ComposableThreads described in Section 4.1 to handle call stacks of application-side threads with special care. Third, MENPS required an efficient user-level threading library to schedule both application-side and system-side threads concurrently. MassiveThreads was initially used for handling systems-side threads, but ComposableThreads superseded it. As a result, ComposableThreads sufficed all of the requirements of MENPS for threading libraries.

ComposableThreads eventually served as a *glue* to connect different components seamlessly in MENPS. The beauty of ComposableThreads is best represented by the fact that it successfully satisfied the three independent demands for building a DSM system with minimal interfacing costs and code duplication. It benefited particularly to profiling MENPS with MassiveLogger[*5]. ComposableThreads enabled logging globally in a uniform man-

[*5] https://github.com/massivethreads/massivelogger

ner like aspect-oriented programming, with a minimal performance penalty. It thus met our desire for performance tuning. Since the components of ComposableThreads were tested and experimented with from multiple purposes within MENPS, we could find and correct the interfacing issues in the early stage of its development. We have also extracted and generalized useful abstractions such as the suspended threads because we expected that many scenarios in system programming require lower-level threading abstractions without runtime costs than the existing threading APIs.

Last but not least, we faced one downside of extensive use of ComposableThreads in MENPS. ComposableThreads is a header-only library full of templates. Almost all the components are parameterized as templates and instantiated at compile time to numerous classes and functions. This design incurs a long compilation time and sometimes unnecessarily expands the compiled code. We dealt with this problem by making appropriate-grained modules composed of components of MENPS and ComposableThreads as fully instantiated templates. Such modules suppress code expansion because they are monolithic and monomorphic. Moreover, they save compilation time because the separate compilation of them is feasible. The downside of ComposableThreads thus has been manageable.

In summary, ComposableThreads worked for MENPS very successfully, which deserves the justification of the design and implementation of ComposableThreads.

7. Related Work

7.1 User-level Threading Systems

For supercomputers, it is common to develop user-level threading systems as work-stealing schedulers. The typical examples are MassiveThreads [19], Argobots [23], Qthreads [26], Nanos++ [20], and libfibre [3], [13] [*6].

Argobots can switch the implementations of several components including schedulers and worker deques, but because those components are separated as individual C modules and switched with function pointers (like virtual function tables), they can never be inlined and always suffer from function invocation overhead beyond the modules. Shared pools are pessimistically implemented with spinlocks because they cannot assume how the other components are implemented, which results in poor scalability shown in our evaluation.

MassiveThreads focused on improving the performance of generic threading rather than parametricity. Although MassiveThreads also implements an I/O wrapper, it is hard-coded inside the library and needs the `configure` option to be enabled.

Boost.Context is a context switching library in Boost, which is used by Boost.Fiber [16]. Argobots also uses the modified version of Boost.Context.

Unlike Argobots and MassiveThreads, libfibre dealt with scheduling for applications in a shared computing environment and so was designed to avoid busy waiting. We can implement the core part of the libfibre scheduler, which consisted of local queues and a shared queue, on ComposableThreads by giving an appropriate scheduler policy as `scheduler_t` although we have not done it yet. In contrast, it is nontrivial to implement the idle sleep of libfibre on the current ComposableThreads.

[*6] In Ref. [13], libfibre was called *fred* for anonymity reason. See the supplementary material of Ref. [13] for the details: https://cs.uwaterloo.ca/~mkarsten/papers/sigmetrics2020_supplementary/

7.2 Lock Delegation

ffwd [21] is a lock delegation method that uses dedicated threads for executing critical sections. ffwd focused on increasing the throughput of critical sections. The performance results of ffwd were, in general, much faster than conventional methods using spinlocks, queue-based locks, software transactional memory, and so on. This is because ffwd does *not* depend on any atomic operations for mutual exclusion but on the round-robin monitoring on dedicated spinning threads.

Klaftenegger et al. [14] proposed queue delegation locking to implement lock delegation based on MCS locks and circular buffers.

The main difference of our approach for lock delegation is separating the user-defined delegated operations from the detail of thread schedulers.

8. Conclusions

ComposableThreads is a user-level threading library with compile-time parametricity. ComposableThreads is implemented as a C++ header-only minimal library that extensively uses zero-overhead abstraction and static type checking for threading. In this work, we have focused on the design of threading libraries that facilitates the efficient plug-and-play of components and allows us to integrate delegation APIs with common threading functionality in a uniform manner. In the microbenchmark evaluation, it has been confirmed that the performance of ComposableThreads was comparable to that of MassiveThreads.

ComposableThreads implements only a small set of user-level threading features. To this library feasible for other purposes, it is required to implement other missing features and evaluate with other applications. Implementing tasklets (e.g., Refs. [12], [23]) or other acceleration techniques with the primitives of ComposableThreads may also be interesting future work.

References

[1] Akiyama, S. and Taura, K.: Uni-Address Threads: Scalable Thread Management for RDMA-Based Work Stealing, *HPDC '15: Proc. 24th International Symposium on High-Performance Parallel and Distributed Computing*, pp.15–26, ACM Press (online), DOI: 10.1145/2749246.2749272 (2015).

[2] Alexandrescu, A.: *Modern C++ Design: Generic Programming and Design Patterns Applied* (2001).

[3] Barghi, S.: Improving the Performance of User-level Runtime Systems for Concurrent Applications, PhD Thesis, University of Waterloo (2018).

[4] Costa, J.J., Cortes, T., Martorell, X., Ayguade, E. and Labarta, J.: Running OpenMP applications efficiently on an everything-shared SDSM, *Journal of Parallel and Distributed Computing*, Vol.66, No.5, pp.647–658 (online), DOI: 10.1016/j.jpdc.2005.06.018 (2006).

[5] Endo, W.: A Decentralized Implementation of Software Distributed Shared Memory, PhD Thesis, The University of Tokyo (2020).

[6] Endo, W., Sato, S. and Taura, K.: MENPS: A Decentralized Distributed Shared Memory Exploiting RDMA, *4th IEEE/ACM Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware, IPDRM@SC 2020*, pp.9–16, IEEE (online), DOI: 10.1109/IPDRM51949.2020.00006 (2020).

[7] Endo, W. and Taura, K.: Parallelized Software Offloading of Low-

Level Communication with User-Level Threads, *HPC Asia 2018: Proc. International Conference on High Performance Computing in Asia-Pacific Region*, Vol.Part F1346, pp.289–298, ACM Press (online), DOI: 10.1145/3149457.3149475 (2018).
[8] Fukuoka, T., Endo, W. and Taura, K.: An Efficient Inter-Node Communication System with Lightweight-Thread Scheduling, *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems* (*HPCC/SmartCity/DSS*), pp.687–696, IEEE (online), DOI: 10.1109/HPCC/SmartCity/DSS.2019.00103 (2019).
[9] Glozer, W.: wrk, available from ⟨https://github.com/wg/wrk⟩.
[10] Guerraoui, R., Guiroux, H., Lachaize, R., Quéma, V. and Trigonakis, V.: LockUnlock: Is That All? A Pragmatic Analysis of Locking in Software Systems, *ACM Trans. Computer Systems*, Vol.36, No.1, pp.1–149 (online), DOI: 10.1145/3301501 (2019).
[11] Intel Corporation: Intel Threading Building Blocks, available from ⟨https://www.threadingbuildingblocks.org/⟩.
[12] Iwasaki, S., Amer, A., Taura, K. and Balaji, P.: Lessons Learned from Analyzing Dynamic Promotion for User-Level Threading, *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp.293–304, IEEE (online), DOI: 10.1109/SC.2018.00026 (2018).
[13] Karsten, M. and Barghi, S.: User-Level Threading: Have Your Cake and Eat It Too, *Proc. ACM Meas. Anal. Comput. Syst.*, Vol.4, No.1, pp.17:1–17:30 (online), DOI: 10.1145/3379483 (2020).
[14] Klaftenegger, D., Sagonas, K. and Winblad, K.: Queue Delegation Locking, *IEEE Trans. Parallel and Distributed Systems*, Vol.29, No.3, pp.687–704 (online), DOI: 10.1109/TPDS.2017.2767046 (2018).
[15] Kowalke, O.: Boost.Context, available from ⟨http://www.boost.org/doc/libs/1_63_0/libs/context/doc/html/index.html⟩.
[16] Kowalke, O.: Boost.Fiber, available from ⟨http://www.boost.org/doc/libs/release/libs/fiber⟩.
[17] Liétar, P., Butler, T., Clebsch, S., Drossopoulou, S., Franco, J., Parkinson, M.J., Shamis, A., Wintersteiger, C.M. and Chisnall, D.: snmalloc: a message passing allocator, *Proc. 2019 ACM SIGPLAN International Symposium on Memory Management, ISMM 2019*, Singer, J. and Xu, H., (Eds.), pp.122–135, ACM (online), DOI: 10.1145/3315573.3329980 (2019).
[18] Mellor-Crummey, J.M. and Scott, M.L.: Synchronization without contention, *ASPLOS IV: Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol.19, No.2, pp.269–278, ACM Press (online), DOI: 10.1145/106972.106999 (1991).
[19] Nakashima, J. and Taura, K.: MassiveThreads: A Thread Library for High Productivity Languages, *Concurrent Objects and Beyond*, Vol.8665, pp.222–238 (online), DOI: 10.1007/978-3-662-44471-9 (2014).
[20] Programming Models Group, Barcelona Supercomputing Center: Nanos++, available from ⟨https://pm.bsc.es/nanox⟩.
[21] Roghanchi, S., Eriksson, J. and Basu, N.: ffwd: delegation is (much) faster than you think, *Proc. 26th Symposium on Operating Systems Principles - SOSP '17*, pp.342–358, ACM Press (online), DOI: 10.1145/3132747.3132771 (2017).
[22] Scheme Reports: Scheme Reports, available from ⟨http://www.scheme-reports.org/⟩.
[23] Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Carns, P., Castello, A., Genet, D., Herault, T., Iwasaki, S., Jindal, P., Kale, L.V., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y., Taura, K. and Beckman, P.: Argobots: A Lightweight Low-Leve Threading and Tasking Framework, *IEEE Trans. Parallel and Distributed Systems* (online), DOI: 10.1109/TPDS.2017.2766062 (2018).
[24] Supercomputing Division, Information Technology Center, The University of Tokyo: Introduction to the Oakbridge-CX Supercomputer System, available from ⟨https://www.cc.u-tokyo.ac.jp/en/supercomputer/obcx/system.php⟩.
[25] Sutton, A.: Wording Paper, C++ extensions for Concepts, Technical Report P0734R0 (Rev. N4674), ISO/IEC (2017).
[26] Wheeler, K.B., Murphy, R.C. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads, *IPDPS '08: Proc. 22nd IEEE International Parallel and Distributed Processing Symposium* (online), DOI: 10.1109/IPDPS.2008.4536359 (2008).

Wataru Endo received his B.E., M.S., and Ph.D. from the University of Tokyo in 2015, 2017, and 2020, respectively. He is currently working as a professional engineer.

Shigeyuki Sato is an Assistant Professor in the Graduate School of Information Science and Technology at the University of Tokyo. He received his Ph.D. from the University of Electro-Communications in 2015. His research interest is in compilers and parallel programming, especially, automatic parallelization, program synthesis, high-level optimizations, domain-specific languages, parallel patterns, and tree/graph processing. He is also a member of ACM and JSSST.

Kenjiro Taura is a Professor in the Department of Information and Communication Engineering at the University of Tokyo. He received his B.S., M.S., and Ph.D. from the University of Tokyo in 1992, 1994, and 1997, respectively. His major research interests spread parallel and distributed computing, system software, and programming languages. He is also a member of ACM, IEEE, and USENIX.