Lessons Learned from Analyzing Dynamic Promotion for User-Level Threading

Shintaro Iwasaki
The University of Tokyo
Tokyo, Japan
iwasaki@eidos.ic.i.u-tokyo.ac.jp

Abdelhalim Amer
Argonne National Laboratory
Lemont, IL, USA
aamer@anl.gov

Kenjiro Taura
The University of Tokyo
Tokyo, Japan
tau@eidos.ic.i.u-tokyo.ac.jp

Pavan Balaji
Argonne National Laboratory
Lemont, IL, USA
balaji@anl.gov

***Abstract*—A performance vs. practicality trade-off exists between user-level threading techniques. The community has settled mostly on a black-and-white perspective; *fully fledged* threads assume that suspension is imminent and incur overheads when suspension does not take place, and *run-to-completion* threads are more lightweight but less practical since they cannot suspend. Gray areas exist, however, whereby threads can start with minimal capabilities and then can be *dynamically promoted* to acquire additional capabilities when needed. This paper investigates the full spectrum of threading techniques from a performance vs. practicality trade-off perspective on modern multicore and many-core systems. Our results indicate that achieving the best trade-off highly depends on the suspension likelihood; dynamic promotion is more appropriate when suspension is unlikely and represents a solid replacement for run to completion, thanks to its lower programming constraints, while fully fledged threads remain the technique of choice when suspension likelihood is high.**

I. Introduction

Threads of execution are one of the major forms of concurrency within the same shared address space, not only on high-performance computing (HPC) systems but also on servers and embedded systems. Because of their heavyweight and rigid nature, OS-level threads have been criticized as being inadequate for fine-grained concurrency to efficiently utilize the modern parallel hardware. As a result, user-level threads have been widely adopted as an alternative for lightweight concurrent execution. Their lightweight nature is achieved by decoupling them from the OS intervention and satisfying their functional requirements with mostly user-space operations.

Perhaps the most widely known form of user-level threading is the one that matches closely the capabilities of OS-level threads; that is, a thread is an execution unit that can be dynamically created and scheduled and can also yield control and resume later (i.e., supporting the *suspension capability*). We refer to such a thread as *fully fledged* (or *Full* for brevity), since it supports all capabilities expected from a user-level thread. Given its wide scope of applicability, numerous parallel programming systems and libraries—including Cilk [1], Intel CilkPlus [2], Qthreads [3], OmpSs [4], MassiveThreads [5], and Argobots [6]—adopted this technique. In order to correctly support the suspension capability, providing a private stack space to a newly spawned thread and performing the necessary context switching operations are mandatory.

Suspension is an important capability that allows, for instance, asynchrony to be achieved when waiting on I/O and communication operations and when synchronizing with other threads. The associated stack and context management operations, however, incur costs that can be significant for fine-grained threads. To avoid these costs, some systems, such as Filaments [7] and Argobots [6], offer the possibility of spawning a thread without requiring the suspension capability. Since it cannot suspend, such an execution unit must *run to completion* (we refer to this technique as *RtC*) and execute on the stack of the caller. By eliminating the suspension capability overheads, *RtC* can be significantly lighter than *Full*, but it is less practical. To leverage this technique, the programmer would use a priori knowledge about the behavior of the unit to infer the possibility of suspension. Doing so is not always possible, however. For instance, if two threads compete for a lock acquisition, the winner would proceed without suspending, while the loser would get suspended. As a result, any possibility of suspension would compel the programmer to avoid *RtC* even if the chances of suspension at runtime were low or nonexistent.

Full and *RtC* have opposing trade-offs with respect to performance and practicality; *Full* is heavier but is fully capable, while *RtC* is lighter but is less practical. There exist alternative methods, however, whereby threads can start with minimal capabilities and later acquire additional capabilities if needed. We refer to such methods as exploiting *dynamic promotion* and note that they potentially offer different trade-offs from those of *Full* and *RtC*. Although hints to dynamic promotion-type threads can be found in the literature [8], [9], [10], little insight into their performance and practicality aspects has been provided. In particular, the circumstances where dynamic promotion-based methods are more suitable than *Full* and *RtC* and whether they can supersede them remain open questions.

This paper fills this gap with an in-depth investigation of the fundamental costs and capabilities of the full spectrum of techniques from *Full* down to *RtC*. By understanding the fundamental differences between them, we systematically depart from *Full* and incrementally trim down costs toward *RtC*. This optimization process successfully locates intermediate threading techniques that have reduced start-up costs and exploits dynamic promotion when additional capabilities are needed. More specifically, we make the following contributions:

1. We provide an in-depth analysis at the instruction and cache levels of the full spectrum of threading techniques from *Full* through *RtC*. We also contribute two new techniques to the spectrum: *return on completion* and *stack separation* (Section III) that, to our knowledge, are missing from the

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

literature. Furthermore, our analysis includes not only the performance aspect of these techniques but also their programming constraints, in order to cover the practicality aspect.

2. We demonstrate that achieving the best trade-off highly depends on the suspension likelihood. Dynamic promotion is more appropriate when suspension is unlikely and represents a solid replacement for *RtC*, thanks to its lower programming constraints, whereas *Full* remains the technique of choice when suspension is likely.

3. We provide highly optimized implementations of these techniques in the same production threading library, Argobots [6], which was also integrated as the substrate threading layer of an OpenMP runtime.

4. We evaluate the performance characteristics of all the methods with representative real-world codes from N-body, graph analytics, and machine learning fields. This study also covers a range of processor architectures: Intel Haswell, Intel Knights Landing, and ARM 64 processors.

Scope of this work. This paper is not an attempt to revive the old *threads* vs. *event* debate [11]. Although running to completion is the mode of operation in event-driven programming models, our target is traditional threading models. That is, on encountering a blocking operation, we do not require the programmer to rip the code [12] and register event handlers as done in event-driven programming. Furthermore, the trade-off between *Full* and *RtC* can also be encountered in the OS context, such as the distinction between work queues and tasklets in the Linux kernel [13]. Our work targets user space applications and leaves the kernel space out of the scope. All the techniques described in this paper are adequate for building generic threading libraries because they do not rely on compiler modifications, special compiler extensions, kernel modifications, or source-to-source translations.

II. Background

Managing user-level threads (ULTs) shares many similarities with how language stack frames are managed, which include register saving and stack pointer manipulation. The major difference is that on encountering a thread *create* or *spawn* call, instead of an immediate function call a thread descriptor and potentially a separate private stack are first created. Then, the execution order of the parent and child threads would depend on the scheduling policy. Here, we assume parent-first scheduling; that is, on encountering a thread creation call, the child thread will be pushed to a thread pool while the parent thread continues its execution. Although some of the methods that will be presented here are applicable to child-first scheduling [14], some exceptions and constraints exist and will be covered in Section III-F. The descriptions below also assume a scheduler that pulls work units from thread pools to be executed. Such a scheduler has a private stack space and runs on an OS-level thread, a scheduling model adopted by most threading packages. In the following, we describe the details of *Full* and *RtC* as commonly implemented by many user-level threading packages.

```
void scheduler() {
  while (true)
    if (thread_desc *thd <- pop_pool())
      thd->f(thd->arg) // schedule thd.
}
```

Fig. 1: Pseudocode of *RtC*. Real schedulers might sleep in order to avoid busy loop and have a branch to finish.

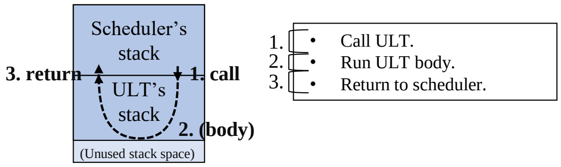


Fig. 2: Flow of fork-join (*RtC*).

A. Run-to-Completion Threads

At creation time, *RtC* requires only that a thread descriptor be allocated, which includes a function pointer and its arguments. Later, the created ULT gets pulled by the scheduler from a thread pool and *called*. Figure 1 shows the pseudocode of a scheduler executing an *RtC* thread. The scheduler runs on an OS-level thread and pulls ULTs from a pool and executes them by calling their function pointers with their arguments. The ULT called on top of the scheduler stack returns after completion in a similar way to a sequential function call (Figure 2). The overheads of managing *RtC* threads over directly calling function pointers lie in the memory management of their thread descriptors and the scheduling costs (e.g., pool-related operations). These overheads are minimal costs to support concurrency, and thus *RtC* is considered as the lower bound in terms of thread management costs.

RtC sacrifices the suspension feature because of several reasons. When a ULT yields, it is supposed to return to the calling scheduler. Restoring the scheduler requires its stack pointer, which can be obtained only with difficulty. Without compiler help, it is hard to retrieve a stack pointer that a compiler automatically saves on the call stack. Callee-saved registers are also difficult to extract from the call stack. Moreover, even if these values could be restored, resuming the execution of the scheduler would grow over the stack frame of the child ULT, which is still alive. Although one can move the child's stack to another place, it is unsuitable for building general threading libraries because it invalidates all addresses pointing to the call stack of the child [15]. A scheduler, therefore, cannot increase the call stack (e.g., call a function) to avoid collapsing the child's stack, making launching another ULT impossible.

B. Fully Fledged Threads

Full, on the other hand, has all the threading capabilities expected from a thread of execution, including the suspension capability. This allows a ULT to yield execution when it is waiting on an event (e.g., completion of an I/O or a network operation) or on synchronization (e.g., critical section, barrier, or joining other threads). By yielding, other units can be scheduled, improving the overall system efficiency instead of wasting CPU time.

```
1  void switch_ctx(ctx_t **self_ctx, ctx_t *target_ctx) {
2    Push callee-saved registers // save the current context.
3    Push the parent instruction address
4    *self_ctx <- stack_pointer
5    stack_pointer <- target_ctx // restore the target context.
6    Pop the target instrcution address to regA // regA is caller-saved.
7    Pop callee-saved registers
8    Jump to *regA
9  }
```

Fig. 3: Pseudo assembly code of a context switch.

```
1  void start_ctx(ctx_t **self_ctx, void *stack, void (*f)(void *),
2                 void *arg) {
3    Push callee-saved registers // save the current context.
4    Push the parent instruction address
5    *self_ctx <- stack_pointer
6    stack_pointer <- stack // start f on top of stack.
7    f(arg)
8  }
9  void end_ctx(ctx_t *target_ctx) {
10   stack_pointer <- target_ctx // restore the target context.
11   Pop the target instrcution address to regA // regA is caller-saved.
12   Pop callee-saved registers
13   Jump to *regA
14 }
```

Fig. 4: Pseudo assembly code to start and end a context.

Before discussing the performance disparity between *Full* and *RtC*, we will first describe the basics of a user-level context switch, which is a key aspect of implementing suspension. Such a switch can explicitly handle an execution state apart from sequential order, which is often called a function context or simply a context. Contexts are saved and restored by manipulating the call stack and values saved in the hardware registers.[1]

Figure 3 describes an implementation of a user-level context switch typically found in practice, such as those in the Boost C++ Libraries [16]. Two pointers are used by `switch_ctx` as arguments: `self_ctx` is a pointer to save the current context, and `target_ctx` points to a target context. This implementation saves register values on top of the stack, so a context is expressed by a single pointer to its call stack. A compiler is responsible for maintaining values in *caller-saved registers* before and after calling `switch_ctx`. Thus, `switch_ctx` just needs to push *callee-saved registers* and an instruction address to the stack (lines 2 and 3). After saving the current stack pointer to `self_ctx` (line 4), the stack pointer is updated to that of the target context (line 5). The target instruction address is loaded into a caller-saved register (line 6) and callee-saved registers are restored by popping them in reverse order (line 7). The target context then is resumed by jumping into the target instruction address (line 8). All these operations are accomplished with user-level instructions.

The `switch_ctx` routine is meant to be called by a scheduler with `target_ctx` being the context of the target ULT to be executed and `self_ctx` being its own context that is needed to resume. This method, however, assumes a preexisting live `target_ctx`; in other words, it assumes that the ULT context has been initialized. In our model, `switch_ctx` is called only when resuming the execution of a suspended ULT. In order to account for the special cases of when ULT executes for the first time and when it terminates, the switching mechanism is split into two methods, `start_ctx` and `end_ctx`, for starting and finishing contexts, respectively (Figure 4). The `start_ctx` routine is meant to execute a ULT for the first time on a fresh `stack` and to start the function `f` given as an argument (line 7). The `end_ctx` routine is called by a terminating ULT to resume the target context (i.e., a scheduler's context) and discards the current context, which is no longer used.

```
1  thread_local ctx_t *g_sched_ctx // g_sched_ctx is worker-local.
2  void scheduler() {
3    while (true)
4      if (thread_desc *thd <- pop_pool()) {
5        if (!thd->is_started) {
6          thd->is_started <- true
7          start_ctx(&g_sched_ctx, thd->stack, thd_wrapper, thd)
8        } else
9          switch_ctx(&g_sched_ctx, thd->ctx)
10       if (!thd->is_finished)
11         add_pool(thd) // return thd to pool.
12     }
13 }
14 void thd_wrapper(thread_desc *thd) {
15   thd->f(thd->arg) // thd->f and thd->arg are given by users.
16   thd->is_finished <- true
17   end_ctx(&thd->ctx, g_sched_ctx)
18 }
```

Fig. 5: Pseudocode of a fully fledged thread technique.

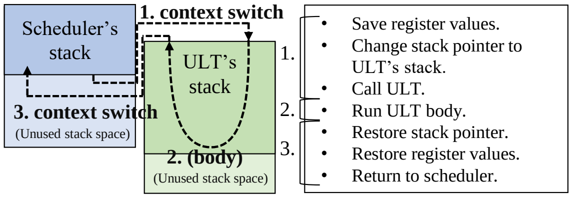


Fig. 6: Flow of fork-join without a suspension (*Full*).

Based on these three functions, we describe an implementation of *Full* as presented in Figure 5. Assume that a scheduler gets a ULT from a pool and starts it. The scheduler invokes the ULT with `start_ctx` (line 7) if the ULT has not been executed yet or `switch_ctx` if it has been suspended (line 9). Since the scheduler context is properly saved in `g_sched_ctx` by `start_ctx` and `switch_ctx`, *Full* ULTs can yield and return to a scheduler at any time by restoring `g_sched_ctx`. This model assumes a preallocated private stack for the ULT before executing it for the first time (`thd->stack` at line 7). A wrapper function, `thd_wrapper`, is used to call `end_ctx` on completion (shown at line 17), since a *Full* thread spawned with a user-level context switch cannot simply resume the scheduler with a standard `return` statement.

Management of independent call stacks and callee-saved registers enables *Full* to suspend. However, as we can see when comparing Figure 6 with Figure 2, the following overheads are incurred even if threads do not happen to suspend.

1. Save callee-saved registers and a stack pointer when a ULT is invoked (`start_ctx`).
2. Restore callee-saved registers and a stack pointer when a ULT finishes (`end_ctx`).
3. Manage call stacks for `thd->stack`.

Ultimately, these additional operations make *Full* slower than *RtC* when threads never suspend.

C. Trade-Off between Capabilities and Performance

As we discussed, *Full* and *RtC* have a trade-off relationship between thread capabilities and their overheads. To gain insight

[1] Unlike OS-level threads, we do not manage signal masks and compiler-level thread-local storage, so these are shared among ULTs running on the same OS-level thread.

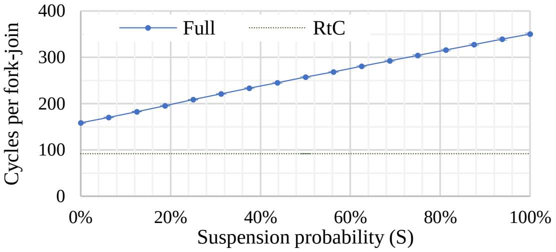


Fig. 7: Fork-join overheads on an Intel Haswell machine. *RtC* shows the performance at $S = 0\%$ because *RtC* cannot suspend.

into the raw performance gap between the two methods, we ran a microbenchmark that repeats 5,000 iterations where each iteration creates and joins a set of 128 empty ULTs. Furthermore, each ULT has an $S\%$ chance of suspending. We ran the benchmark on a single core of an Intel Haswell processor (the detailed experimental environment is described in Section IV). Figure 7 shows the fork-join overhead with respect to the suspension probability. Each data point is the arithmetic mean of all iterations across 50 runs. Since *RtC* cannot suspend, we show only a horizontal line representing no suspension for reference. We observe that the suspension probability increases the fork-join cost linearly for *Full*. More important, at $S = 0\%$, the overhead of *Full* is 1.75x higher than that of *RtC* even though threads never suspend.

Ideally, *Full* would perform similarly to *RtC* when suspension does not occur. Prior work has not made clear whether the additional costs incurred by *Full* in this case are compulsory in order to support the suspension capability. In the next section we address this issue by performing an in-depth investigation of fundamental costs between the two methods.

III. Toward Minimal Overhead with Dynamic Promotion

This section aims at bringing insight into the extra costs of implementing the fully fledged thread and deriving techniques that avoid them when the suspension probability is low. We proceed by incrementally trimming down costs at creation and execution time and moving them toward the suspension path. As will be seen, some of these techniques incur programming constraints that limit their applicability. These restrictions, however, are significantly less constraining than *RtC*, and we encourage their adoption. The following analysis relies on the microbenchmark explained in the preceding section. Complete data, including performance and instructions, is summarized in Figure 13 and Figure 14 at the end of this section.

A. Lazy Stack Allocation

In the implementation of *Full*, we considered a ULT private stack preallocated before the first invocation. Such an eager approach is an optimization often encountered in practice. It allows the runtime to reduce memory management overheads by allocating together thread descriptors and their corresponding stacks. A large difference between *Full* and *RtC* is that an *RtC*-type thread is run on top of the scheduler's stack, whereas a *Full*-type thread has its own stack space allocated at creation time. *Full* accesses a different call stack region on every spawn, thus incurring 1.7x more L1 cache misses when S is 0%.

However, we do not always need independent stack regions for every thread; only simultaneously active ULTs require independent stacks. To improve cache locality, we devise a *lazy stack allocation* (*LSA*) method that assigns stacks at *execution* time. Since most ULTs are forked and joined sequentially when the suspension probability is low, a call stack can be reused by assigning it on invocation. The other flow is the same as that of *Full* presented in Figure 6.

We find that *LSA* shows slightly higher performance than does *Full* by successfully reducing L1 cache misses; moreover, the number of misses gets close to that of *RtC*. *LSA* potentially adds five instructions to manage a stack, however, because *Full* allocates a stack and a thread descriptor at the same time on creation, whereas *LSA* allocates them separately. In addition, at high suspension probability, most ULTs have unique stacks, and the effect of stack reuse gets diminished. Therefore, *LSA* degrades performance at high suspension probability.

B. Context Switching on Return

A large performance gap still remains between *LSA* and *RtC*. User-level context switches (`switch_ctx`) inflate the number of instructions. The first context switch is indispensable for saving a context of a scheduler so that a scheduler can be resumed at any point. When a created thread does not suspend and straightforwardly returns to a scheduler, however, the last context switch does not need to restore callee-saved registers because these values are restored at the end of the spawned function. In other words, a user-level context switch on spawning is inevitable, but one on joining can be simplified if a spawned thread never suspends.

A *return on completion* (*RoC*) removes the second context switch by replacing it with a standard return procedure. It resumes a scheduler by a standard return procedure when a thread is completed without suspensions. Figure 8 shows the pseudocode of a function invoking threads, and Figure 9 illustrates the flow.

```
void switch_ctx_RoC_invoke(ctx_t **self_ctx, void *stack,
                           void (*f)(void *), void *arg) {
  Push callee-saved registers
  Push an instruction address
  *self_ctx <- stack_pointer
  stack_pointer <- stack
  f(arg) // a user function is directly called.
  return
}
void switch_ctx_RoC_return() {
  thread_desc *thd <- get_self_thread()
  end_ctx(&thd->ctx, g_sched_ctx)
}
```

Fig. 8: Pseudo assembly code of a context switch in *RoC*.

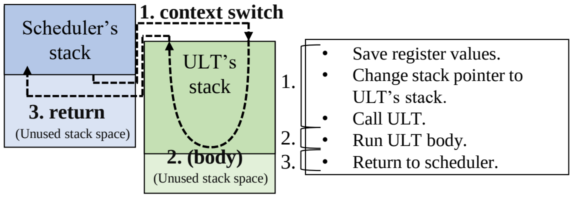


Fig. 9: Flow of *RoC*.

The first part saves callee-saved registers and calls a thread function `f` at line 7. If a created thread does not suspend, this function just returns to a caller with a `return` instruction.[2] The explicit management of callee-saved registers is not needed because callee-saved registers are maintained in `f`. If a thread suspends, the context of the calling scheduler is modified when resumed, making callee-saved registers preserved by `f` outdated. By changing a return address, we solve this issue without adding an extra branch in `switch_ctx_RoC_invoke`. When a thread suspends for the first time, it calculates an address pointing to a return address of `switch_ctx_RoC_invoke` from a ULT stack space and rewrites the return address to `switch_ctx_RoC_return` so that `switch_ctx_RoC_invoke` can jump to `switch_ctx_RoC_return` at line 8 and restore callee-saved registers stored in `g_sched_ctx`. We note that a thread needs to manipulate the return address only on the first suspension, so the overhead after the first suspension becomes the same as that of *Full*.

As a result, *RoC* keeps the number of L1 cache misses low and successfully saves 27 instructions compared with *LSA* when the suspension probability is 0% and therefore reduces fork-join overheads by 35% compared with *Full*. However, *RoC* worsens the performance at higher suspension probability (13% worse when S is 100%) because of the complicated control flow handling a return address.

C. Context Switching on Invocation

An *RtC* thread is still faster than other techniques when no threads yield. *RoC* suffers from overheads of managing stacks and contexts to restore a scheduler context if threads suspend. If the scheduler has no state, however, we can discard a context of the current scheduler (e.g., all local variables, data in the stack frame) and freshly start a new one. We call this property *statelessness*. Specifically, a stateless scheduler must have no execution state preserved across thread invocation so that it can be terminated while invoking a thread and newly started. A few frameworks [8], [9], [10] have adopted this technique, which we call *scheduler creation* (*SC*). *SC* removes the whole cost of user-level context switches and stack management when threads finish without suspending. When a thread does suspend, however, this technique incurs additional costs to restart a scheduler from the beginning of the function, although such action is unnecessary if the scheduler context is properly saved.

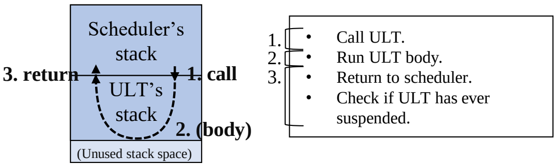


Fig. 10: Flow of *SC*.

Figure 10 shows how *SC* works. A thread function is called on top of a scheduler's stack. When a thread has never yielded, in comparison with *RtC*, *SC* imposes no extra overhead except a single branch to check whether a thread has suspended. When an *SC* thread suspends for the first time, because a parent scheduler that has **called** this thread directly cannot be resumed, it spawns a ULT with a new stack and starts a scheduler on top of it. At the same time, the thread is marked as suspended in order to prevent the original parent scheduler from resuming after the completion of this thread. This invalidation mechanism maintains the number of active schedulers. We note that from the second suspension *SC* threads become fully fledged threads, so superfluous schedulers are not created. The experiment shows that *SC* adds only two instructions for a branch and achieves performance as high as *RtC* does when threads do not suspend.

D. Overcoming the Stack Size Constraint

Although *SC* achieves high performance when $S = 0\%$, *SC* imposes two constraints. First, the scheduler needs to be stateless because the context of the scheduler is possibly lost. A carefully designed random work-stealing scheduler [19] satisfies this property, while some schedulers managing counters stored in local variables, for example, to select a victim of work stealing or sleep when work stealing fails several times, are not stateless. In addition, dynamically allocated heap memory, exclusive locks, and file handles obtained in a scheduler must be carefully managed so as not to lose track of them across thread executions. Second, this technique runs a thread on top of a stack of a scheduler, so the stack size of *SC* threads is actually shared with that of a scheduler. Because we must use the largest stack size among various types of threads in a program, *SC* imposes an unnecessarily large stack size. This problem becomes significant when one application has multiple types of threads each of which requires a different stack size.

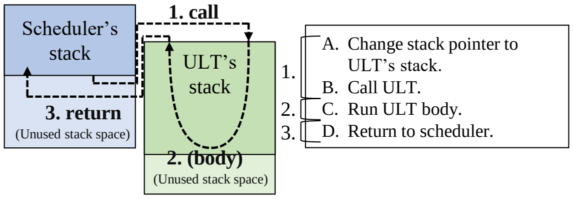


Fig. 11: Flow of *SS*.

```
void switch_ctx_SS_invoke(ctx_t **self_ctx, void *stack,
                          void (*f)(void *), void *arg) {
  Push an instruction address
  *self_ctx <- stack_pointer
  stack_pointer <- stack
  f(arg) // a user function is directly called.
  return
}
```

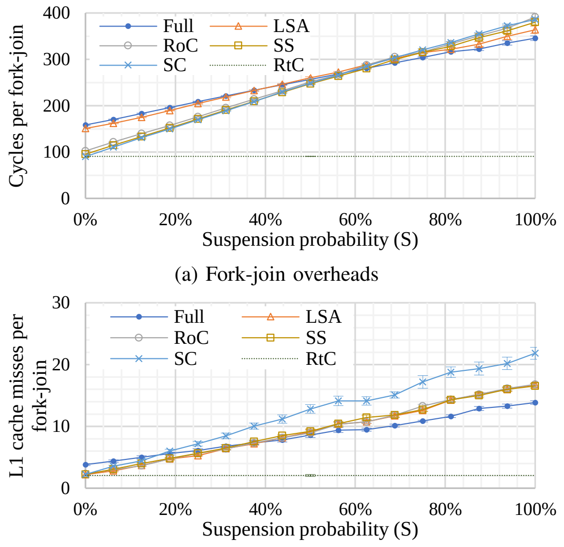
Fig. 12: Pseudo assembly code of a context switch in *SS*.

The second constraint derives from the fact that stacks are shared between threads and schedulers. To address this, we propose a threading technique that calls a thread on top of an independent stack, hence the name *stack separation* (*SS*). Figure 11 shows how *SS* works. *SS* does not save registers while it runs a thread on a unique stack rather than on top of

[2] This pseudocode assumes a `return` instruction that pops an instruction address and jumps to that address and a certain calling convention. Although our current implementation supports only x86-64 [17] and ARM64 [18], this technique should be widely applicable to various calling conventions and instruction sets.

TABLE I: Summary of threading techniques

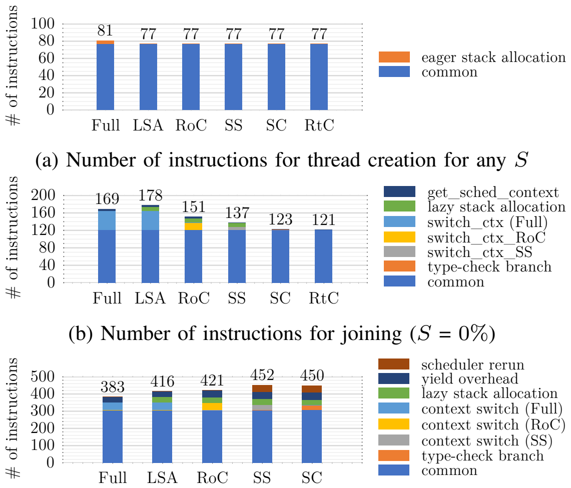
	No suspension			Suspension		Constraints
	Change stacks?	# of register managements?	Overheads	Rerun scheduler?	Overheads	
Full (*Fully Fledged Thread*)	Yes	2	Highest	No	Lowest	No
LSA (*Lazy Stack Allocation*)	Yes	2		No		No
RoC (*Return on Completion*)	Yes	1	∧	No	∨	No
SS (*Stack Separation*)	Yes	0		Yes		Scheduler must be stateless.
SC (*Stack Creation*)	No	0		Yes	Highest	Scheduler must be stateless. Stack size is shared.
RtC (*Run to Completion*)	No	0	Lowest	-	-	Suspension is not allowed.



(a) Fork-join overheads

(b) Number of L1 cache misses obtained by PAPI [20]

Fig. 13: Performance of the six methods on Haswell.



(a) Number of instructions for thread creation for any S

(b) Number of instructions for joining (S = 0%)

(c) Number of instructions for joining (S = 100%)

Fig. 14: Instruction breakdown of the six methods on Haswell.

the scheduler's stack. Because the context of a scheduler is not fully saved, *SS* requires a stateless scheduler, as does *SC*. As presented in Figure 12, a fork function run by a scheduler is that of *RoC* (Figure 8) without a part saving registers. *SS* uses the same technique that utilizes a general return mechanism in nonsuspended cases.

Figure 13[3] shows the performance of the six threading techniques, and Figure 14 summarizes the number of instructions for thread creation and joining. We can see that *SS* achieves slightly worse performance than *SC* does since 14 instructions are added to manage stacks, as shown in Figure 14b. When the suspension probability is high, *SS* shows better performance than *SC* does, because a scheduler can reuse its uniquely associated stack region, thus reducing cache misses, as presented in Figure 13b.

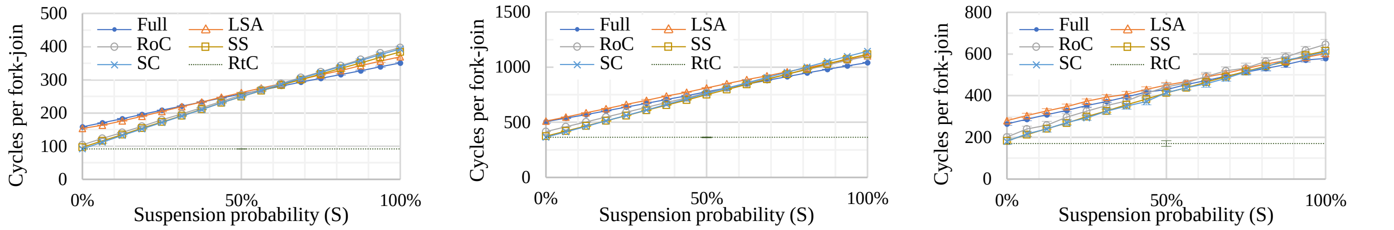
E. Discussion

Table I summarizes the six threading techniques. We can observe a performance trade-off with respect to suspension probability. *Full* is slowest when the suspension probability is low but achieves the highest performance when a thread yields. In contrast, *SC* minimizes the fork-join costs but incurs the highest suspension cost. *LSA* slightly reduces the fork-join cost of *Full* to the detriment of a higher suspension cost. The remaining dynamic promotion techniques perform closely to *RtC*. From a programming perspective, *Full*, *RoC* and *LSA* incur no constraints while the other techniques have various limitations. *SS* and *SC* require stateless schedulers and *SC* additionally puts a constraint on stack size. *RtC* guarantees the lowest fork-join cost but incurs the highest programming constraint by giving up the suspension capability.

From a user perspective, a threading technique can be selected as follows. If the user knows that most threads suspend, *Full* is the best choice. On the other hand, if threads never suspend, *RtC* is preferable. If the user knows in advance that few threads yield, *LSA*, *RoC*, *SS*, or *SC* should be used. When schedulers are not stateless, *RoC* is the best method because *RoC* has minimum fork-join costs among threading techniques with the suspension feature. *SS* and *SC* are also useful with stateless schedulers. Usually, if a program has a single task type, *SC* is the best choice since the user does not need to use different stack sizes. The user can choose *SS* if a program has various types of threads requiring differently sized stacks.

If the user has no idea about the application behavior, *RoC* is recommended. *RoC* is a suspendable threading technique without any constraints and performs well when suspension is less likely. Nevertheless, *Full*, for example, shows higher performance when most ULTs suspend, while threading overheads in such cases tend to be negligible because of time-consuming operations causing suspensions (e.g., high communication contentions). Automatic selection of thread types, through runtime adaptation or compile-time code analysis, is a promising research direction. Given the difficulty of automating the identification of threading capability requirements for the more constraining methods (*SS*, *SC*, and *RtC*), however, a separate study outside the scope of this paper is warranted.

[3] Higher levels of caches do not suffer from cache misses in this experiment because each ULT accesses a small portion of a call stack.



(a) Cycles per fork-join on Haswell

(b) Cycles per fork-join on KNL

(c) Cycles per fork-join on ARM64

Fig. 15: Cycles per fork-join ($N = 128$).

F. Coverage of Our Techniques

Section III assumes parent-first schedulers. A natural question is whether other techniques are possible.

For Parent-First Schedulers. To discuss the coverage of our techniques, we focus on management of registers and stacks, which has a major impact on costs of a user-level context switch. We described that *Full* allocates a stack for each ULT and explicitly handles callee-saved registers twice on fork and join. Reduction of these operations to alleviate this overhead is straightforward. As shown in the second and third columns in Table I, these operations are eliminated one by one from *Full* and *LSA* to *SC*. We note that we miss threading techniques that do not change stacks but explicitly maintain callee-saved registers (i.e., [“No”, “2”] and [“No”, “1”]). Such techniques have been proposed [21], [22], [23]. However, these techniques are not suitable for library approaches because the techniques need compiler modifications [22]. We discuss this problem further in Section V. In terms of the number of these operations, all the techniques to construct a parent-first threading library are considered in our paper, although other parts—including designs of schedulers, thread queues, and memory pools—might leave room for improvement.

For Child-First Schedulers. Our related work includes studies based on a child-first scheduling policy [14]. Not all the methods are directly associated in the child-first case. *SS* and *SC* require stateless parent functions, although in the child-first case parent functions invoking children are user-written functions, not a scheduler. Losing the context of parents often loses the computational results, so the constraints of those two techniques are tighter and less practical. Since a created thread with the new stack is immediately executed in the child-first scheduling, there is no distinction between *LSA* and *Full*. We note that, as with the parent-first case, threading techniques that do not alter stacks but instead push and pop registers need compiler help [21], [22], [23]. *RoC* is applicable and promising. Its evaluation is listed as one of our future projects.

IV. Evaluation and Analysis

In this section, we evaluate the performance of the methods with a microbenchmark and three fine-grained parallel applications. All the methods were implemented on Argobots [6], a threading library adopting a parent-first scheduling policy. Table II shows the experimental environment. We compiled all the programs with Intel Compiler 17.2.174 for Haswell and KNL and with GCC 4.8.5 for ARM64, with the `-O3` compilation flag. The stack size of ULTs was set to 16 KB. The scheduler's stack size was set to the same size as well, which is beneficial for *SC*. We used the arithmetic mean and the standard deviation to obtain an average and the error bar, respectively.

TABLE II: Experimental environment

Name	Haswell	KNL	ARM64
Processor	Intel Xeon E5-2699 v3	Intel Xeon Phi 7210	AMD Opteron A1120
Architecture	Haswell	Knights Landing	ARMv8-A
Frequency	2.3GHz	1.3GHz	1.7GHz
# of sockets	2	1	1
# of cores	36	64	4
# of HWTs	72	256	4
Memory	123GB	198GB	8GB
OS	Red Hat 7.4	Red Hat 7.4	openSUSE 42.2

A. Overheads of Fork-Join

The fork-join microbenchmark that we used repeats creating N threads and joining all N threads $5,000$ times on a single worker. At each iteration, exactly n threads randomly distributed in N threads suspend once. The suspension probability S is calculated by n/N. Because *RtC* cannot suspend, the result of *RtC* is valid only at $n = 0$. We ran the benchmark 50 times on Haswell, KNL, and ARM64 with different n while fixing N to 128. Lightweight private pools [6] were used so that we could evaluate only the threading overheads.

Figure 15 shows the results on Haswell, KNL, and ARM64. Even though we use the number of cycles as a metric, the absolute performance on KNL is worse because KNL is a less powerful superscalar processor. However, Haswell, KNL, and ARM64 show similar performance except for *LSA*; *Full* is slowest and *SC* fastest when S is low, while *Full* shows the highest performance when S is high. *LSA* is not faster than *Full* on KNL and ARM64 even at $S = 0\%$. As discussed in Section III-A, *LSA* increases the number of instructions to reduce cache misses, so *LSA* lowers performance on KNL and ARM64, which have different instruction and memory costs from those of Haswell.

We evaluate three fine-grained parallel applications, reflecting three situations where fine-grained ULTs are useful.

KMeans Parallel programs expected to execute other ULTs if running ULTs fail to acquire locks.

ExaFMM Recursive parallel programs in which most ULTs do not suspend because they are leaves, while nodes need suspension to wait for other ULTs.

Graph500 Parallel programs expected to run other ULTs if the current ULTs need to wait for communications (although they are less likely to happen).

B. KMeans over OpenMP

OpenMP is the most popular parallel programming system for multithreading because of its rich APIs and portability. Several implementations, including OmpSs [24] and Intel OpenMP [25], utilize ULTs in OpenMP, especially as tasks. Since OpenMP's tasks and threads are suspendable (i.e., `taskyield` and `barrier`), they have been created as *Full*-type ULTs. As we pointed out in this paper, however, not all parallel units require suspension capability. The dynamic promotion techniques can exploit the benefits from the nonsuspension case as well as *RtC*. We used KMeans for evaluation.

KMeans is a well-known machine learning algorithm for partitioning N points into K groups. Our benchmark is based on a simple KMeans implementation in NU-MineBench [26]. This algorithm works as follows. Initially, a randomly distributed center is assigned to each of K clusters. A point is considered belonging to the cluster whose center is nearest to that point. The algorithm repeats updating cluster centers to centroids of their points until the positions of the centroids get stable. In our program, we create as many as N threads each of which is associated with a point and updates a centroid of the nearest cluster. To parallelize calculation of new centroids, we follow a simple technique with critical sections used by Chabbi et al. [27]. Each thread updates the partial sums of centroids shared among workers, and a master thread sums up the partial results at the end of each iteration. The partial sums are protected by locks to avoid conflicts.

We artificially change the number of locks L to protect partial sums in order to control the lock granularity. When $L = 1$, there is only a single global lock for all clusters, so only one ULT can update a partial sum at the same time. When $L = K$, locks are prepared for every cluster, so no contentions occur unless multiple ULTs try to update a partial sum of the same cluster. L can be set to more than K by making replicates of partial sums per cluster, although doing so increases the reduction cost at the end of iterations. When $L = K \cdot W$, where W is the number of workers, since each worker has a replicate of all clusters, ULTs can update the partial sums without contention. When $1 < L < K$, we assign a lock to K/L partial sums. When $K < L < K \cdot W$, we create L/K replicates of partial sums per cluster and make $K \cdot W/L$ workers share the replicates and their locks.

We parallelized the original KMeans with OpenMP by a doubly nested parallel loop; the outer loop creates 64 OpenMP threads, and the inner loop spawns tasks in a finest-grained manner. We customized BOLT [28], an OpenMP runtime system over Argobots, to map OpenMP threads and tasks to ULTs based on the five suspendable threading techniques.[4] The kernel was rearranged to exploit vector units in KNL. We used the first 10% data of KDD Cup 1999 [29]. Our dataset contains $N = 5.0 \times 10^5$ points each of which consists of 41 floating-point features.[5] The number of clusters K is given as 24 in the original problem statement.

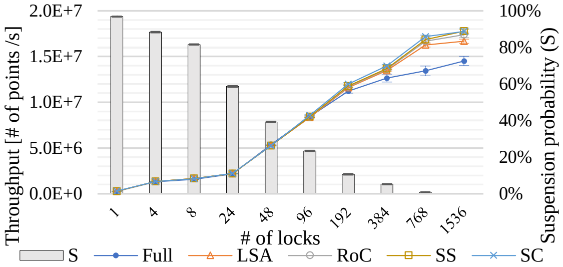


Fig. 16: Throughput and suspension probability (S) of *SC* of KMeans using 64 cores.

Figure 16 shows the average throughputs of 20 executions with 64 workers. As the number of locks increases, the suspension probability declines; and therefore *SC*, *SS*, *RoC*, and *LSA* achieve better performance than *Full* does. The performance difference among threading methods diminishes where L is less than 100 or S is higher than 40%. Our microbenchmark indicates that *Full* achieves the best performance at higher suspension probability, but the performance improvement becomes negligible as the locking overheads become dominant. This result demonstrates that the dynamic promotion techniques can exploit performance of nonsuspension cases without losing the suspension functionality, while the integration with other parallel systems is easy.

C. ExaFMM

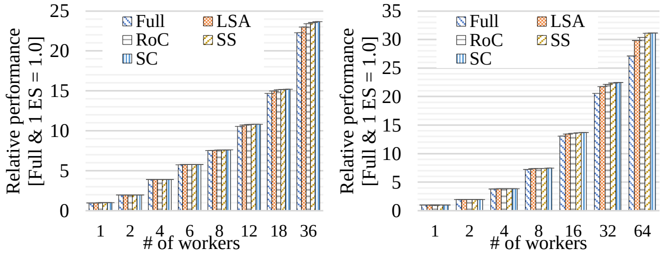
ExaFMM [30] is a highly optimized fast multipole method that computes an N-body problem with $O(N)$ complexity. ExaFMM has a kernel recursively parallelized by ULTs [31] to exploit modern many-core CPUs. A tree is traversed in a divide-and-conquer manner, and leaf threads calculate actual forces. Node threads in a tree need a suspension feature so that they can wait for child threads, but leaf threads never suspend because they just contain computations. The most efficient way seems to be to create leaf threads as *RtC* and nodes as suspendable threads (e.g., *Full*). However, this approach not only burdens programmers but also requires identifying leaf threads on creation, incurring additional overheads. The dynamic promotion techniques explained in the paper can reduce the programmer load and achieve good performance.

We parallelized ExaFMM with our proposed techniques and ran it 10 times on Haswell and KNL. The kernel was vectorized by hand. To reduce nonleaf threads, we collapsed intermediate nodes in the spawn tree; those nodes do not have computation but only decompose work. We give `--ncrit 16 -t 0.15 -P 4 --dual -n 100000` as arguments and change `--nspawn` to keep the number of created ULTs per worker constant (within 3% of error), while `--nspawn 60` is given when 36 and 64 workers are used on Haswell and KNL, respectively. We measured the performance of the tree traversal, which is responsible for more than 90% of the total execution time.

Figure 17 plots the performance with different numbers of workers. The result shows that *SC*, *SS*, *RoC*, and *LSA* achieve better performance than *Full* does on both Haswell and KNL: 6% at maximum with 36 cores on Haswell and 15% on KNL.

[4] The environmental variable is used to change a thread type. Finer-grained thread type control in OpenMP runtime system is part of our future work.

[5] We arbitrarily mapped string-typed values to floating-point values.



(a) Haswell (b) KNL

Fig. 17: Relative performance of ExaFMM. The baseline is the performance of *Full* with a single worker. The suspension probability (S) was up to 5%.

KNL shows larger gaps between the various methods because its wider SIMD units reduce kernel computation time while the absolute overhead of context switch is larger (Figure 15), which accentuates thread management overheads. Performance differences among methods except for *Full* are small; for example, 4% between *LSA* and *SC* on KNL.

We note that nearly $1 - \frac{1}{k}\%$ of threads in a k-ary tree are leaves.[6] The microbenchmark shown in Figure 15 indicates that the dynamic promotion techniques perform better than *Full* does when the suspension probability is less than 50%. Since 50% of threads become leaf nodes even when the shape of the task tree is binary, the dynamic promotion is suitable for parallel recursive algorithms.

D. Distributed Graph500

Graph500 [32] runs a breadth-first graph traversal on distributed computing environments. In this section, we refer to a compute node as an MPI process or just a process since we use an MPI library and assign one process per a compute node. Each process has a part of the whole graph, so interprocess communication is necessary in order to visit vertices in a graph owned by other processes. In each iteration, processes visit adjacent vertices and, if they are not locally stored, send messages to other processes to update them. Simultaneously, local vertex information must be updated upon messages from others. A buffer length B is the number of visit messages per process that are temporarily stored locally to manually aggregate communications. Since the performance of Graph500 is sensitive to communication overheads, hybrid parallelism is often used to reduce the local communications in a compute node. MPI+Thread, where ULT is used as an implementation of "Thread," has been known to achieve good performance for finer-grained parallelism on a distributed system [33] since, when an MPI function blocks, a worker can efficiently switch to another thread and process it. Even nonblocking MPI functions (e.g., `MPI_Isend` and `MPI_Test`) might sometimes block inside a runtime in order to take a lock of communication resources shared among multiple workers, although the behavior is implementation dependent.

TABLE III: Experimental environment of Graph500

Processor	Intel Xeon Phi 7230	Architecture	Knights Landing
Frequency	1.3 GHz	# of cores	64
# of HWTs	128	Memory	128 GB
OS	Red Hat 7.4	Interconnect	Intel Omni-Path

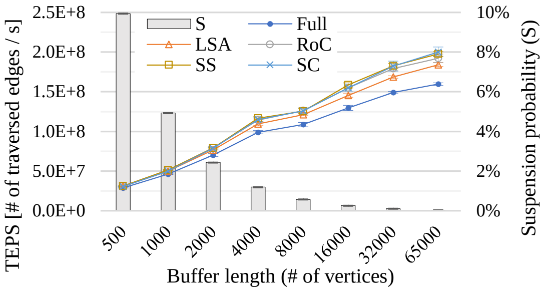


Fig. 18: TEPS and suspension probability (S) of *SC* of Graph500 using 1K cores.

The dynamic promotion can alleviate threading overheads in cases where ULTs do not call MPI functions or where called MPI functions never suspend inside.

We modified the multithreaded Graph500 implementation [34] to use our optimized Argobots and Argobots-aware MPICH [35]. To focus on threading overheads, we associated one visit with a ULT. The buffer length B was changed to control the communication granularity; with a larger B, more memory is consumed, but fewer communications are required. We set a scale factor to 26, so a created graph has 2^{26} vertices. We executed this benchmark on 16 KNLs described in Table III. One compute node is associated with a single MPI process running 64 workers, so 1,024 workers are used in total. We ran the benchmark five times and show the average TEPS and the suspension probability of *SC*. The results indicate that the larger B coarsening the communication granularity improves performance on the whole, while the dynamic promotion techniques outperform *Full* as S approaches 0%. *Full* shows better performance with a smaller B, but it worsens the overall performance and diminishes the benefits of lightweight ULTs.

V. Related Work

Numerous parallel libraries and parallel languages have been developed to efficiently support high-level parallel programming languages or APIs. The explicit suspension feature is not supported, however, although not all of them use *RtC*. For example, Cilk [1], [36], Intel CilkPlus [2], and Intel TBB [37] cannot manually suspend. Nevertheless, several user-level threading libraries, such as Qthreads [3], Nanos++ [4], MassiveThreads [5], and Argobots [6], support suspension. We analyzed the costs and features of lightweight threading techniques that threading libraries can adopt. Most previous papers introducing the libraries and systems we listed above have not focused on threading techniques but have instead focused on other characteristics such as portability, usability, abstraction, and performance improved by other optimizations (e.g., locality-aware scheduling). The performance of various parallel systems has been also intensively evaluated in prior work [38], [39], but such work has measured only overall performance and lacked a cost analysis of different threading

[6] Denote the number of nodes in a tree N and the number of leaves n. (N, n) is $(1, k)$ when $N = 1$ and 1 leaf can be replaced with 1 node and k leaves, so (N, n) is in general $(N, N(k-1)+1)$. The ratio of leaf threads is therefore $\frac{n}{N+n} \approx 1 - \frac{1}{k}$, when N is large.

methods. In this section, we briefly discuss notable studies out of countless work using ULTs in terms of threading techniques.

Register Saving: Some previous studies adopting a child-first scheduling policy, including LazyThreads [21], StackThreads/MP [40], [22] and Fibril [23], proposed techniques that avoid manipulation of stack pointers and just save (or clobber) callee-saved registers. Their approaches are based on the following premises.

1. All local variables in the thread stack are addressed by a frame pointer instead of a stack pointer.
2. The stack is not shrunken dynamically in the midst of a function.
3. All threads are joined in a function creating them.

If these premises are satisfied, a thread can call a child function on top of the parent stack after saving registers. The approach obviously works well if work stealing does not happen. When another worker steals a continuation, a thief allocates a new stack, restores the original registers, and resumes the execution. A frame pointer points to the original stack, but a stack pointer points to a new stack allocated by a thief worker. Premise 1 guarantees that local variables previously used are referenced by a frame pointer so that a thief can access them. A thief can call a new function on top of the newly allocated stack based on a stack pointer without violating the child stack.

In addition to premise 3, narrowing the expressiveness of parallelization, the premises require compiler modifications. Taura et al. [22], for example, discussed necessary modifications to GCC for premise 1 and premise 3. Lazy Threads [21] extensively modified a compiler, which also utilizes this method without any difficulties. Yang and Mellor-Crummey [23] tried to address this problem without compiler modifications by adding a special compiler flag, `-fno-omit-frame-pointer`, provided by GCC. Unfortunately, the current widely used compilers including GCC do not provide a flag that guarantees that all local variables are addressed by the frame pointer. In order to avoid compiler modification, the threading techniques we present in this paper do not use this approach.

Stack Separation: In contrast to the approach saving registers, a few studies have proposed methods to omit register manipulations. However, their approaches are different from ours; they adopt new calling conventions that have no callee-saved registers except a register representing a stack pointer and instruction address (e.g., Intel CilkPlus [41]). Context switches therefore can be done by changing a stack pointer and an instruction address. In contrast, our *SS* does not modify a compiler to change calling convention.

Scheduler Creation: A few papers have mentioned scheduler creation. Chores [8] and Wool [9], both of which are parent-first threading libraries, have adopted this method to reduce overheads according to a low suspension probability scenario. Recently, Concurrent Cilk [10], which is a child-first threading library, utilized this technique to integrate a suspension feature into Intel CilkPlus. Nevertheless, the past work has not offered an in-depth comparison with other threading techniques. Moreover, their approaches handle threads that suspend once in a special manner, so suspended threads are differently scheduled from threads that do not suspend. Our techniques are implemented with unified pools and schedulers so that they are uniformly scheduled.

Run-to-Completion Threads: In order to save the cost of user-level context switch, numerous studies—including Filaments [7], Qthreads [3][7] and Argobots [6]—support a run-to-completion thread. It is efficient in both cost and memory, but its constraint limits the applicable cases. For instance, a run-to-completion thread is not feasible for programs used in our evaluation. In particular, ExaFMM and Graph500 might cause a deadlock by replacing a suspension with a spin loop.

Other Threading Techniques: Several other lightweight threading techniques have also been developed that cannot be classified into the categories above. For example, Sivaramakrishnan et al. [42] proposed MultiMLton, in which function stacks are relocatable. This technique might be applicable to pure function languages, although it cannot support the general C/C++ code. Cilk-M [15] enables stack copying by modifying operating systems to expose the same address space. The approach is not generally applicable, however, since OS modification is required.

VI. Concluding Remarks

This paper investigates the full spectrum of user-level threading techniques between *Full* and *RtC*. Specifically, we analyze six threading techniques and discuss their costs and constraints. *Full* shows the best performance at high suspension probability. *LSA* and *RoC* are techniques with the same threading capability as *Full*, but they can reduce threading overheads up to 5% and 30% on Haswell when suspension probability is low. *SC* is as fast as *RtC* when threads do not suspend, while additional constraints such as stack size limitations and scheduler requirements are imposed. *SS* is a technique to overcome the stack size limitations at a 5% additional cost compared with *SC* on Haswell. We implemented these methods in the same runtime system and measured their performance on three processors that have different architectures. We also evaluated the performance of three fine-grained applications and demonstrated potential performance improvement by exploiting a wide range of trade-offs.

Our goal is a comprehensive understanding of lightweight threading techniques for threading libraries. This work specifically focuses on costs of context switches and stack management, while other factors such as schedulers, memory allocators, and thread pools might be left suboptimal. Investigating their fundamental overheads is a direction of our future work.

Acknowledgment

We gratefully acknowledge the computing resources provided and operated by LCRC and JLSE at ANL. This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357, and by the Exascale Computing Project (17-SC-20-SC).

[7] Qthreads executes a thread on top of the scheduler's stack when `QTHREAD_SPAWN_SIMPLE` is specified.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '95, July 1995, pp. 207–216.

[2] C. E. Leiserson, "The Cilk++ concurrency platform," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09, July 2009, pp. 522–527.

[3] K. B. Wheeler, R. C. Murphyand, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '08, Apr. 2008, pp. 1–8.

[4] X. Martorell, J. Labarta, N. Navarro, and E. Ayguadé, "A library implementation of the nano-threads programming model," in *Proceedings of the second European Conference on Parallel Processing*, ser. EuroPar '96, Aug. 1996, pp. 644–649.

[5] J. Nakashima and K. Taura, "MassiveThreads: A thread library for high productivity languages," in *Concurrent Objects and Beyond*, ser. Lecture Notes in Computer Science, Jan. 2014, vol. 8665, pp. 222–238.

[6] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, Oct. 2017.

[7] D. K. Lowenthal, V. W. Freeh, and G. R. Andrews, "Efficient support for fine-grain parallelism on shared-memory machines," *Concurrency: Practice and Experience*, vol. 10, no. 3, pp. 157–173, March 1998.

[8] D. L. Eager and J. Jahorjan, "Chores: Enhanced run-time support for shared-memory parallel computing," *ACM Transactions on Computer Systems*, vol. 11, no. 1, pp. 1–32, Feb. 1993.

[9] K.-F. Faxén, "Wool - A work stealing library," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 93–100, June 2009.

[10] C. S. Zakian, T. A. Zakian, A. Kulkarni, B. Chamith, and R. R. Newton, "Concurrent Cilk: Lazy promotion from tasks to threads in C/C++," in *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, ser. LCPC '15, Sept. 2016, pp. 73–90.

[11] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, ser. HOTOS '03, 2003.

[12] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management," in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '02, June 2002, pp. 289–302.

[13] J. Wiegert, G. Regnier, and J. Jackson, "Challenges for scalable networking in a virtualized server," in *16th International Conference on Computer Communications and Networks*, Aug 2007, pp. 179–184.

[14] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr., "Lazy task creation: A technique for increasing the granularity of parallel programs," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90, June 1990, pp. 185–197.

[15] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson, "Using memory mapping to support cactus stacks in work-stealing runtime systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, Sept. 2010, pp. 411–420.

[16] Boost C++ Libraries. http://www.boost.org/.

[17] J. Hubička, A. Jaeger, M. Matz, and M. Mitchell, "System V application binary interface AMD64 architecture processor supplement," https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf, Oct. 2013.

[18] "Procedure Call Standard for the ARM 64-Bit Architecture," http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf, May. 2013.

[19] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, Sept. 1999.

[20] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, Aug. 2000.

[21] S. C. Goldstein, K. E. Schauser, and D. E. Culler, "Lazy Threads," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 5–20, Aug. 1996.

[22] K. Taura, K. Tabata, and A. Yonezawa, "StackThreads/MP: Integrating futures into calling standards," in *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '99, May 1999, pp. 60–71.

[23] C. Yang and J. Mellor-Crummey, "A practical solution to the cactus stack problem," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '16, July 2016, pp. 61–70.

[24] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: a proposal for programming heterogeneous multi-core architectures." *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[25] Intel OpenMP runtime library. https://www.openmprtl.org/.

[26] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A benchmark suite for data mining workloads," in *Proceedings of 2006 IEEE International Symposium on Workload Characterization*, ser. IISWC '06, Oct. 2006, pp. 182–188.

[27] M. Chabbi, M. Fagan, and J. Mellor-Crummey, "High performance locks for multi-level NUMA systems," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '15, Feb. 2015, pp. 215–226.

[28] BOLT | BOLT is OpenMP over lightweight threads. http://www.bolt-omp.org/. Accessed: 2018-2-1.

[29] KDD Cup 1999 Data. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. Accessed: 2017-12-04.

[30] ExaFMM, Boston University. http://www.bu.edu/exafmm/. Accessed: 2017-11-03.

[31] K. Taura, J. Nakashima, R. Yokota, and N. Maruyama, "A task parallelism meets fast multipole methods," in *Proceedings of the 3rd Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '12, Nov. 2012.

[32] J. Ang, B. Barrett, K. Wheeler, and R. Murphy, "Introducing the Graph 500," Cray User's Group (CUG), May 2010.

[33] H. Lu, S. Seo, and P. Balaji, "MPI+ULT: Overlapping communication and computation with user-level threads," in *Proceedings of the 17th International Conference on High Performance Computing and Communications*, ser. HPCC '15, Aug. 2015, pp. 444–454.

[34] A. Amer, H. Lu, P. Balaji, and S. Matsuoka, "Characterizing MPI and hybrid MPI+Threads applications at scale: Case study with BFS," in *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '15, May 2015, pp. 1075–1083.

[35] MPICH | High-Performance Portable MPI. https://www.mpich.org/.

[36] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98, June 1998, pp. 212–223.

[37] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[38] A. Podobas, M. Brorsson, and K.-F. Faxén, "A comparative performance study of common and popular task-centric programming frameworks," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 1, pp. 1–28, Jan. 2015.

[39] G. W. Price and D. K. Lowenthal, "A comparative analysis of fine-grain threads packages," *Journal of Parallel and Distributed Computing*, vol. 63, no. 11, pp. 1050–1063, Nov. 2003.

[40] K. Taura and A. Yonezawa, "Fine-grain multithreading with minimal compiler support - a cost effective approach to implementing efficient multithreading languages," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI '97, 1997, pp. 320–333.

[41] "Intel Cilk Plus application binary interface specification," https://www.cilkplus.org/sites/default/files/open_specifications/CilkPlusABI_1.1.pdf, Dec. 2011.

[42] K. Sivaramakrishnan, L. Ziarek, R. Prasad, and S. Jagannathan, "Lightweight asynchrony using parasitic threads," in *Proceedings of the Fifth ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, ser. DAMP '10, Jan. 2010, pp. 63–72.

A. Artifact Description: Lessons Learned from Analyzing Dynamic Promotion for User-Level Threading

A.1 Abstract

We explain how to obtain and install the benchmarks and dependent libraries and run experiments described in the paper.

A.2 Description

The core of the proposal is implemented in Argobots. The paper shows the results of four benchmarks: a microbenchmark to evaluate fork-join costs (*Yield*), k-means clustering parallelized by OpenMP (*KMeans*), a fast multipole method (*ExaFMM*), and a distributed breadth-first graph traversal (*Graph500*). The experiments were performed on four environments: *Haswell*, *KNL*, *ARM64*, and *KNL_Cluster*. Table IV summarizes the environments, and Table V shows the correspondence between benchmarks and environments evaluated in the paper.

A.2.1 Checklist (artifact metainformation):

- **Algorithm**: Lightweight user-level threading techniques described in the paper. Benchmarks include k-means clustering (*KMeans*), a fast multipole method (*ExaFMM*), and a distributed breadth-first graph traversal algorithm (*Graph500*).
- **Program**: C/C++ code.
- **Compilation**: Intel C/C++ Compiler 17.2.174 for *Haswell* and *KNL*, GCC 4.8.5 for *ARM64*, and Intel C/C++ Compiler 17.0.4 for *KNL_Cluster*.
- **Hardware and runtime environment**: See Table IV.
- **Experiment workflow**: Download and compile libraries and benchmarks and run test scripts.
- **Experiment customization**: Change the configuration parameters for compilation, or give different arguments at runtime.

A.2.2 How software can be obtained (if available): All libraries, benchmarks, and scripts are available at http://argobots.org/files/2018/08/sc18_dynamic_promotion_material.tar_.gz.

A.2.3 Hardware dependencies: The current implementation supports only x86/64 and ARM-64 machines adopting the corresponding System V ABIs. The instruction analysis might be difficult on machines where Intel SDE does not work. Argobots-aware MPICH should work in any environments supported by the original MPICH, while our performance was obtained on KNL clusters connected with Intel Omni-Path.

A.2.4 Software dependencies: Some benchmarks and analyses rely on uncommon libraries. All of them are installed just by running `download.sh`, while Intel SDE and Intel C/C++ Compilers must be manually installed. The PSM2 library is required to utilize Intel Omni-Path. Our scripts contain some general commands such as `git`, `cmake`, `automake`, and `python`.

A.2.5 Datasets: KMeans uses KDD Cup 1999, which is publicly available. The download process is also automated.

A.3 Installation

Since our experiments require several libraries and benchmarks, it is cumbersome to download and build them one by one. Except for compilers (i.e., `icc`), Intel SDE and the PSM2 library, all the libraries and datasets are available in the tar ball or obtained by running `download.sh`. The compilation process is automated by scripts; running `build.sh xxx` builds a benchmark of *xxx* and all the necessary libraries. Users should read those scripts to know what parameters are passed. For example, to set up everything on an Intel Xeon Phi cluster, one types the following.

TABLE IV: Experimental environments used in the paper

Name	*Haswell*	*KNL*	*ARM64*	*KNL_Cluster*
Processor	Intel Xeon E5-2699 v3	Intel Xeon Phi 7210	AMD Opteron A1120	Intel Xeon Phi 7230
Architecture	Haswell	Knights Landing	ARM v8 -A	Knights Landing
Frequency	2.3 GHz	1.3 GHz	1.7 GHz	1.3 GHz
# of nodes	1	1	1	16
# of sockets	2	1	1	1 x 16
# of cores	36	64	4	64 x 16
# of HWTs	72	256	4	256 x 16
Memory	123 GB	198 GB	8 GB	128 GB
OS	Red Hat 7.4	Red Hat 7.4	openSUSE 42.2	Red Hat 7.4
Interconnect	-	-	-	Omni-Path

TABLE V: Correspondence between benchmarks and environments described in the paper

	Yield	*Yield* (analysis)	*KMeans*	*ExaFMM*	*Graph500*
Haswell	✓	✓		✓	
KNL	✓		✓	✓	
ARM64	✓				
KNL_Cluster					✓

```
$ compiler=icc
$ arch=intel
$ sh download.sh $compiler $arch
$ sh build.sh all
```

A.4 Experimental workflow

All the four benchmarks are run by `eval_xxx.sh` where *xxx* is a benchmark name. Log files are written to `log/xxx` directories. Those scripts can be executed locally or as a batch job. Embedded parameters are by default configured for the environments discussed in the paper (e.g., the number of threads). We recommend running *Graph500* on multiple nodes, but the others are for a single node. The following commands execute *Yield*, *Yield* with detailed performance analysis, *KMeans*, *ExaFMM*, and *Graph500*, respectively.

```
$ sh eval_yield.sh
$ sh eval_yield_sde.sh
$ sh eval_kmeans.sh
$ sh eval_exafmm.sh
$ nprocs=16
$ sh eval_graph500.sh $nprocs
```

A.5 Evaluation and expected results

Log files in the `log` directory contain all the results.

A.6 Experiment customization

Experiments can be easily customized by changing parameters embedded in compilation scripts and execution scripts.