Analyzing the Performance Trade-Off in Implementing User-Level Threads

Shintaro Iwasaki, *Student Member, IEEE,* Abdelhalim Amer, *Member, IEEE,*
Kenjiro Taura, *Member, IEEE,* and Pavan Balaji, *Member, IEEE*

Abstract—
User-level threads have been widely adopted as a means of achieving lightweight concurrent execution without the costs of OS-level threads. Nevertheless, the costs of managing user-level threads represent a performance barrier that dictates how fine grained the concurrency exposed by an application can be without incurring significant overheads; this in turn may translate into insufficient parallelism to exploit highly parallel systems.

This article is a deep dive into the fundamental costs in implementing user-level threads. We first identify that one of the highest sources of fork-join overheads stems from *deviations*, events that incur context switching during the execution of a thread and disrupt a run-to-completion execution. We then conduct an in-depth investigation of a wide spectrum of methods with respect to how they handle deviations while covering both parent- and child-first scheduling policies. Our methodology involves a comprehensive instruction- and cache-level analysis of all methods on several modern CPU architectures. The primary finding of our evaluation is that dynamic promotion methods that assume the absence of deviation and dynamically provide context-switching support offer the best trade-off between performance and capability when the likelihood of deviation is low.

Index Terms—Multithreading, Multitasking, Scheduling, User-Level Threads, Context Switch, Task Parallelism

1 Introduction

Multithreading is the predominant form of parallelism to exploit modern highly parallel multicore and many-core processors. On self-bootable systems, such as traditional servers and the second generation Intel MIC accelerators, the majority of programming systems map their threading abstractions to OS-level threads (e.g., most systems target the POSIX Threads (Pthreads) specification, which itself maps Pthreads to OS-level threads). This approach is known to be too heavyweight to exploit dynamic, irregular, and massive parallelism because of its expensive thread management involving OS kernel operations. Hence, numerous studies have proposed lightweight implementations of threads that bypass the OS layer and rely mostly on user-space operations. In this paper, we call these implementations *user-level threads* (ULTs).[1] Numerous production and research threading libraries including major production OpenMP runtimes (as tasks) [1], [2], [3], Qthreads [4], Nanos++ [5] (used in OmpSs [6]), Converse [7] (used in Charm++ [8]), Filaments [9], MassiveThreads [10], and Argobots [11] have adopted ULTs as an implementation of abstract parallel units.

Despite ULTs being more lightweight than OS-level threads, managing ULTs remains a runtime overhead that should be minimized since it dictates how fine-grained work executed by threads can be. For instance, if we assume that an application can tolerate at most 5% threading overheads and that thread management costs average $1\mu s$, then thread granularity (in terms of time to execute a unit of work) must be at least $20\mu s$. For the application to decompose work into more fine-grained portions (e.g., to express more parallelism), threading overheads must be reduced.[2] There is a lower bound on these overheads that is related to supporting the most basic form of concurrency, which requires managing thread descriptors and allowing basic scheduling primitives, such as fork and join operations. Beyond these, we identified that supporting context switching is the next most significant source of overheads.

A context switch involves a ULT (*current*) jumping to another ULT (*target*) by restoring an execution context of the target often after saving the context of the current ULT. Context switching is necessary in order to support complex control flows such as yielding threads of execution (e.g., `pthread_yield()` provided by Pthreads), intermediate termination (e.g., `pthread_exit()`), efficient synchronization (e.g., `pthread_cond_wait()`), and child-first scheduling [12]. Given the practical importance of context switching, several threading runtimes made it compulsory and thus pay the associated performance overheads regardless of the application control flow [5], [7], [10]. Others are more mindful of the context-switching overheads and thus focus on minimizing them by giving up the context-switching capability altogether [1], [2], [3], [9]. A few runtime systems expose both models as distinct language or runtime abstractions so that users can choose either of them based on application requirements [4], [11].

Our first take at this problem showed that it is possible to dynamically provide the context-switching capability later when needed [13]. We referred to this type of method as a *dynamic promotion* technique. Our prior study was based on a direct correlation between yielding and context switching; that is, a context switch triggered by a yield operation requires thread promotion. This observation was biased by the parent-first scheduling assumption, however, which does

[1] Literature on a *parallel programming model* tends to refer to such a lightweight parallel unit as a "task", while their *underlying implementations* are often discussed as "threads." This paper focuses on implementations and thus uses "threads" and "ULTs".

[2] Example: reducing granularity to $10\mu s$ with the same overhead tolerance of 5% requires lowering threading overheads to $0.5\mu s$

```
thd_desc_t *create_thd(void (*f)(void *), void *arg);
void join_thd(thd_desc_t *thd);
void yield_thd(void);
```

(a) Threading operations that appear in this paper. **RtC** does not support a yield operation (`yield_thd()`) since it requires a context-switching capability (Section 2.1). Other operations such as intermediate termination and synchronization primitives (e.g., a barrier and a mutex) are omitted since they do not appear in our example codes.

```
1  void comp(void *arg) { [...]; }
2  // A parallel version of the following loop:
3  // for (int i = 0; i < n; i++) comp(args[i]);
4  void parallel_loop(void **args, int n) {
5    thd_desc_t *thds[n];
6    for (int i = 0; i < n; i++) // fork ULTs.
7      thds[i] = create_thd(comp, args[i]);
8    for (int i = 0; i < n; i++) // join ULTs.
9      join_thd(thds[i]);
10 }
```

(b) Example code using `create_thd()` and `join_thd()`.

Fig. 1. Basic threading API of a user-level threading library we use in this paper and example code with this API.

not apply to child-first scheduling [12]. To capture causes of thread promotion regardless of the scheduling policy, this paper borrows the notion of *deviation* from Spoonhower et al. [14]. This concept lets us shed light on the fundamental causes of thread promotion in various patterns of execution as well as helps us design thread management methods that reduce the corresponding overheads. We then evaluate the performance characteristics of a large spectrum of user-level threading methods with representative real-world codes and link their execution patterns to probabilities of deviation. Our primary finding is that dynamic promotion techniques exhibit the best trade-off between performance and capabilities when the chances of deviation are low.

Specifically, the contributions of our paper are as follows:

- Identifying deviation as the fundamental cause that incurs context-switching and thus imposes the associated fork-join overheads when implementing ULTs;
- In-depth characterization of the performance vs. capability trade-off with respect to the probability of deviation while covering all feasible methods for building a generic threading library, including a few methods missing from prior literature;
- Identifying stack management as an orthogonal dimension to the problem and demonstrating its applicability to several threading techniques and its tremendous effect on performance and memory usage;
- Highly optimized implementations of all methods within the same threading library (Argobots [11]) for a fair comparison of all the threading techniques;
- Coverage of major hardware architectures in the high-performance computing community—Intel Skylake, Intel Knights Landing (KNL), ARM 64, and IBM POWER8 processors—to highlight the importance of lightweight user-level threading techniques for architectures that employ less powerful cores and have larger thread contexts;
- Evaluating all the threading methods with N-body, machine learning, and graph analytics codes, in which deviations happen during execution. The results indicate that dynamic promotion techniques that defer context management until a deviation happens show the best performance vs. capability trade-off when deviations are unlikely.

This manuscript is an extension of the conference paper previously published by the authors [13]. This paper has more comprehensive coverage by including new threading methods (with respect to stack allocation timing and scheduling policies) and a wider range of modern hardware architectures. In addition, the notion of deviation captures causes of context switching outside a yield operation. Our experiments with the new POWER8 implementation highlight the efficacy of dynamic promotion techniques on different hardware with the corresponding calling conventions. Furthermore, our deeper evaluation with a microbenchmark reveals that the number of created ULTs also affects the performance trade-off. As a result, this paper depicts a comprehensive picture of the performance vs. capability trade-off between user-level threading techniques regarding stack allocation strategies, scheduling policies, CPU architectures, and thread counts, which helps programmers select the best threading methods that fit their hardware architectures and application workloads.

Scope of the Paper

We explore neither a granularity control technique that serializes threads [15], [16] nor a scheduling technique that improves memory locality and alleviates scheduling overheads (e.g., thread pool contention) [17], [18]; our approach tackles the granularity issue from a different aspect by minimizing threading overheads, which can coexist with granularity control strategies and scheduling methods proposed in the previous studies. We do not discuss other parallel programming paradigms (e.g., an event-driven programming model [19]); the focus of our work is a common multithreaded programming model. The target of this work is user-level threading techniques to build a generic threading library without source-to-source translations, compiler modifications (e.g., Cilk [20]), or kernel modifications (e.g., Cilk-M [21]).

2 Basics of a User-Level Threading Library

Our explanation in this work is based on a threading library with a simplified API sketched in Fig. 1, which can be found in most threading libraries. Among functions listed in Fig. 1a, fork and join are the most basic operations; a fork function (`create_thd()`) creates a ULT, and a join function (`join_thd()`) waits for the completion of a given ULT and frees its resource.[3] A *thread pool* is a data structure to keep ready ULTs. A ready ULT is popped from a thread pool and executed by a *scheduler* that runs with its own stack on the corresponding OS-level thread (*worker*). Our following explanation assumes a work-stealing model [23] for load balancing; each worker has its own thread pool and attempts to steal a ready ULT from another worker's pool if needed (e.g., when its local pool is empty).[4] This fork-join mechanism is powerful enough to parallelize several parallel patterns including a parallel loop presented in Fig. 1b.

[3] We do not impose fully strict computation [22] and allow arbitrary synchronization operations (including a barrier and a mutex) between threads in order to maintain flexibility and generality.

[4] This paper does not assume a specific implementation of thread pools and work-stealing algorithms.

Consider the simplest thread implementation that supports only fork and join. The essence of fork and join operations is a schedulable function that can be detached from the current execution context and later invoked. Compared with a function call, minimal additional operations to implement such ULTs are twofold: (1) a thread descriptor that stores completion status, a function pointer, and its argument and (2) a scheduling mechanism that keeps thread descriptors and runs a ready ULT, both of which are fundamental for detaching and deferring the execution of the function. A threading method that satisfies only these requirements is the simplest and most lightweight. This technique, however, abandons all threading features that require a context switch; that is, once scheduled, such a ULT does not stop until completion. Hence we call it a *run-to-completion* thread (**RtC**). We first explain the implementation of **RtC** and show why **RtC** can only run to completion. We then describe how to overcome the limitation of **RtC**.

2.1 Run-to-Completion Thread (RtC)

```
1 void scheduler() {
2   while (true)
3     if (thd_desc_t *thd = pop_pool())
4       thd->f(thd->arg) // schedule thd.
5 }
```

Fig. 2. Pseudocode of **RtC**.

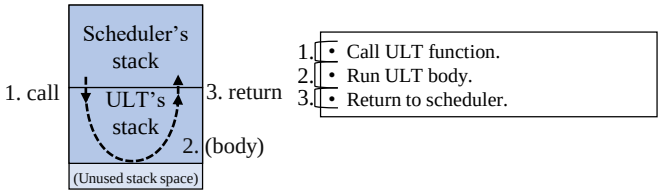


Fig. 3. Flow of fork-join (**RtC**).

We present the pseudocode of **RtC** in Fig. 2 and its execution flow in Fig. 3. **RtC** requires only a thread descriptor and a scheduler; on thread creation, **RtC** allocates a thread descriptor that holds a function pointer and its argument and pushes it to a thread pool. A thread in a pool is pulled by a scheduler running on a worker and simply *called* on top of the scheduler. Compared with an immediate function call natively supported by programming languages, **RtC** incurs overheads of thread descriptor management and scheduling including thread pool operations, both of which are indispensable costs to detach the execution.

Although **RtC** has the smallest fork-join overheads, it lacks threading capabilities that require a context switch because a simple function call welds together a scheduler and an invoked thread; a scheduler that spawns an **RtC** thread cannot be resumed while the invoked thread is running. Consider a yield operation (`yield_thd()` in Fig. 1a) that returns the control from a ULT to a scheduler. In order to restore the context of the scheduler, values of hardware registers (including a stack pointer and an instruction address) must be reinstated. Nevertheless, **RtC** saves none of them explicitly on invocation; thus, although these values are possibly stored somewhere in the call stack of **RtC** as instructed by a compiler, a threading library cannot retrieve these values. Even if registers could be restored, because the invoked **RtC** thread and the scheduler share the same stack region, any stack growth caused by a function call or an invocation of another **RtC** thread would overwrite the call stack of the previous **RtC** thread. This scheduler-thread welding deprives **RtC** of threading features that require an independent invoker's context; unsupported features are not only yielding but also intermediate termination, efficient synchronization, and child-first scheduling. This limitation critically lowers the practicality of **RtC**.

```
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 instruction address to regA // regA is caller-saved.
7   Pop callee-saved registers
8   Jump to *regA
9 }
```

Fig. 4. Pseudo assembly code of user-level context switch.

2.2 Thread with Full Threading Capabilities

RtC lacks a context-switching capability because it bonds contexts of a scheduler and a thread together. If their contexts are maintained independently, however, a ULT can return to a scheduler at any point. A *fully fledged* threading technique creates and maintains a thread context in order to support full threading capabilities. Such a thread allows efficient scheduling, but it suffers from context management overheads. To understand the difference in performance and capabilities between these two opposite threading techniques, we first explain user-level context switch, an essential operation to implement fully fledged threads. Our implementation of user-level context switch follows that of Boost C++ Libraries [24]; similar codes are found in major threading packages, for example, in Qthreads [4], Nanos++ [5], Converse [7], and MassiveThreads [10] as well as Argobots [11]. We note that most ULT implementations do not maintain signal masks and compiler-level thread-local storage for every ULT, so they are shared among ULTs running on the same worker. Figure 4 presents the pseudocode of user-level context switch. This implementation represents a context as a single pointer to the call stack (`ctx_t*` in the figure) since all the other data are saved at the top of the stack. Since a caller of `switch_ctx()` is responsible for saving and restoring *caller-saved* registers before and after calling `switch_ctx()`, `switch_ctx()` itself needs to manage only *callee-saved* registers (lines 2 and 7).[5] This routine first saves all the callee-saved registers including an instruction address on top of the stack (lines 2 and 3) and stores the current stack pointer in `self_ctx` (line 4). Then, `switch_ctx()` updates the stack pointer to the stack address pointed to by `target_ctx` (line 5) and restores the instruction address and the callee-saved register values from the stack of the target in reverse order (lines 6 and 7). The target, which is suspended in `switch_ctx()`, is resumed by jumping to the target instruction address (line 8). We note that all of these operations are executed in the user space.

If a scheduler's context has been saved properly, `switch_ctx()` enables a ULT to save its context and resume a scheduler whenever it needs to return to a scheduler. This method, however, is inappropriate for initiating a ULT because `switch_ctx()` takes `target_ctx` that must have been

[5]Threading libraries must save and restore all callee-saved registers specified by application binary interfaces (ABIs) because, without a special compiler help, libraries are unable to obtain information about which callee-saved registers are read after calling `switch_ctx()`.

```
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 instruction address to regA // regA is caller-saved.
12   Pop callee-saved registers
13   Jump to *regA
14 }
```

Fig. 5. Pseudo assembly code to start and finish thread contexts.

already initialized. This routine always saves the context of the current ULT, but this action is unnecessary when a ULT finishes because that ULT will never be resumed again. To efficiently handle these cases, we split the functionality of `switch_ctx()` and create two methods, `start_ctx()` and `end_ctx()`, to start and finish contexts, respectively. Figure 5 shows the pseudocodes of these functions. Their implementations come from the first and the latter parts of `switch_ctx()`. `start_ctx()` saves the context of the current thread (lines 3–5) but freshly executes a function `f()` on top of `stack` (lines 6 and 7), while `end_ctx()` restores and resumes the target context (lines 10–12) without saving the current context.

Fully fledged threads that support the full threading capabilities are implemented with the three context-switching functions described above. In reality, we can find two implementations of fully fledged threads that have been developed to support different scheduling policies; one is for *parent-first* scheduling, and the other is for *child-first* scheduling.[6] We first explain parent-first fully fledged threads and then child-first threads.

2.2.1 Parent-First Fully Fledged Thread (**Full**)

```
1  thread_local ctx_t *g_sched_ctx // worker-local variable.
2  void scheduler() {
3    while (true)
4      if (thd_desc_t *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         enqueue_pool(thd) // return thd to pool.
12     }
13 }
14 void thd_wrapper(thd_desc_t *thd) {
15   thd->f(thd->arg) // thd->f and thd->arg are given by users.
16   thd->is_finished = true
17   end_ctx(g_sched_ctx)
18 }
```

Fig. 6. Pseudocode of **Full**.

A parent-first scheduling policy is the same as the scheduling order of **RtC**; on `create_thd()`, a parent (i.e., a caller) pushes a child thread to a thread pool and resumes the execution of the parent itself, and later a scheduler executes the child stored in the thread pool. For example, in Fig. 1b, a parent thread that runs `parallel_loop()` first creates all child threads and pushes them to a thread pool in the loop (lines 6–7). On `join_thd()` (line 9), the parent thread checks the completion of each child thread. If the child thread is not completed (e.g., by this worker or other workers), the parent cannot make progress and thus context-switches to a scheduler and runs a ready ULT.

[6]Parent-first scheduling is sometimes called *help-first* scheduling while child-first is called *work-first*.

```
1  thread_local thd_desc_t *g_current_thread // worker-local variable.
2  thd_desc_t *create_thd(...) {
3    thd_desc_t *thd = allocate_thd_desc_t()
4    init_thd_desc(thd, ...)
5    thd->parent = g_current_thread
6    start_ctx(&g_current_thread, thd->stack, thd_wrapper, thd)
7    return thd
8  }
9  void thd_wrapper(thd_desc_t *thd) {
10   push_local_pool(thd->parent)
11   thd->f(thd->arg) // thd->f and thd->arg are given by users.
12   thd->is_finished = true
13   ctx_t *next_ctx = pop_local_pool_or_get_sched_ctx()
14   end_ctx(next_ctx) // child-first scheduling expects next == parent.
15 }
```

Fig. 7. Pseudocode of **C-Full**.

In this paper, we refer to the implementation of a fully fledged threading technique with parent-first scheduling as **Full**. Figure 6 shows the pseudocode of **Full**. The scheduler first pops a ULT (`thd`) from a pool (line 4) and starts it by `start_ctx()` (line 7) if `thd` has not been executed previously; otherwise it resumes `thd` by `switch_ctx()` (line 9) since its context has already been initialized. A user-given thread function is called in a wrapper function `thd_wrapper()` (line 15) so that `end_ctx()` is executed on completion (line 17) because a ULT invoked by `start_ctx()` cannot return to the parent scheduler just by a standard `return` procedure.

Since both `start_ctx()` and `switch_ctx()` save the scheduler's context in `g_sched_ctx`, the scheduler can be resumed by `switch_ctx()` or `end_ctx()` at any time, thus allowing yielding, intermediate termination, and efficient synchronization. Child-first scheduling also requires user-level context switch, as we describe in the next section.

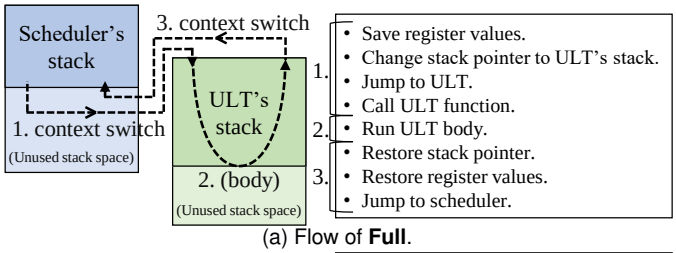
2.2.2 Child-First Fully Fledged Thread (**C-Full**)

Child-first scheduling [12] is a different scheduling policy from that of **RtC** and **Full**; under the child-first scheduling policy, on thread creation, a parent thread yields to a child thread, and the child pushes the parent into a thread pool so that another scheduler can steal the continuation of the parent ULT. After the child completes, it preferably jumps back to the parent thread if the parent is still in the thread pool. For instance, in Fig. 1b, a caller of `parallel_loop()` (i.e., a parent) pushes its continuation to a thread pool and executes a child thread first on `create_thd()` (line 7). Parallelization is achieved by exposing a continuation of a parent thread to other workers. If no work stealing happens, the child returns to the parent context on completion and the parent creates a next child thread in the loop (lines 6–7). Since this child-first scheduling naturally executes threads in sequential order (or depth-first order) if no work stealing happens, it is often adopted to parallelize divide-and-conquer recursive algorithms for better locality [20], [22]. Such a child-first fully fledged thread, which we call **C-Full** in this paper, can also be implemented with the context-switching functions shown in Fig. 4 and Fig. 5.

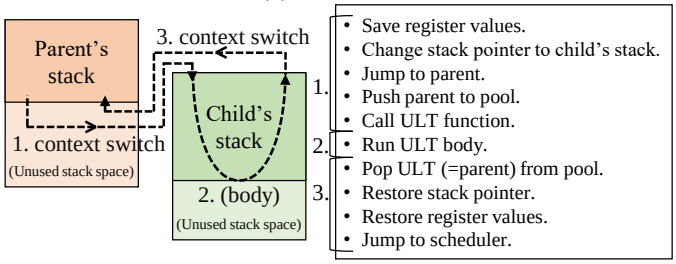
The pseudocode of **C-Full** is presented in Fig. 7. **C-Full** performs a context switch in a thread creation function (`create_thd()`); after allocating and initializing a thread descriptor, a parent thread saves its context and jumps to a child thread by `start_ctx()` (line 6). The child pushes the parent to a local thread pool (line 10) before running a user-given thread function (line 11) to expose concurrency. On completion, the child thread checks the next thread in the pool, which is ideally the parent thread so that execution order is depth-first. However, the child does not always succeed in taking the parent because it might have been

either stolen by another scheduler or resumed by threading operations (e.g., `yield_thd()`). If this is the case, the child thread jumps to another thread if it exists; otherwise, the child thread returns to the scheduler (`scheduler()` in Fig. 6). We note that **C-Full** has also the full threading capabilities since all parent and child threads and schedulers maintain their contexts independently.

2.3 Performance Comparison



(a) Flow of **Full**.



(b) Flow of **C-Full**.

Fig. 8. Flow of fork-join when no deviation happens.

In both the parent- and child-first cases, the management of call stacks and callee-saved registers plays a key role in supporting full threading capabilities and child-first scheduling. In real applications, however, the ULT is often executed as if it were just *called* by following a normal function-call procedure; **Full** threads can finish without any context switch during execution, and **C-Full** threads can just run to completion and return to the parent thread.

To analyze the performance difference, we use a notion of *deviation* appearing in [14][7]. A deviation in the work by Spoonhower et al. [14] is defined as an event that prevents a ULT from sequential execution (i.e., execution order where all thread creations are inlined). Since sequential execution is often most efficient in terms of memory locality [14], [25], the number of deviations has been used as a metric that represents how far the resulting parallel execution differs from sequential execution. This idea works well for child-first scheduling; the number of deviations can be zero if neither work stealing nor yielding happens. If we follow the original definition, however, all fork and join operations of parent-first threads incur deviations because, unlike child-first order, parent-first order is different from sequential execution order even when a parent-first ULT is executed in a run-to-completion manner. This paper, therefore, generalizes the notion of deviation by defining it as an event causing an execution that is different from sequential execution *except* such an event on forking and joining parent-first threads. With this definition, the number of deviations under parent-first scheduling can be zero in a case where no context switch happens during the execution of a thread. Deviation includes any threading operations

[7]We note that this is called differently in other literature; for example, such an event is called "drifted" in [25].

```
1  void kernel(void *yield_flag) {
2    if (yield_flag != NULL)
3      yield();
4  }
5  void microbenchmark(int n) {
6    void *yield_flags[N];
7    thd_desc_t *thds[N];
8    // n yield_flags are set to non-NULL, while 0 <= n <= N.
9    set_yield_flags(yield_flags, n);
10   for (int i = 0; i < N; i++) // fork ULTs.
11     thds[i] = create_thd(kernel, yield_flags[i]);
12   for (int i = 0; i < N; i++) // join and free ULTs.
13     join_thd(thds[i]);
14 }
```

Fig. 9. Microbenchmark that forks and joins N ULTs while random n out of N ULTs encounter deviations invoked by `yield_thd()`.

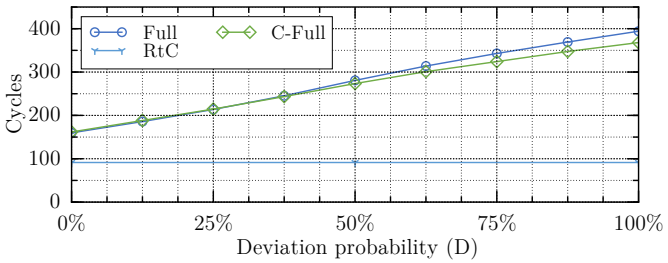


Fig. 10. Fork-join overheads on an Intel Skylake machine using a microbenchmark presented in Fig. 9 (N = 4,096). **RtC** shows the performance at $D = 0\%$ because **RtC** does not allow any deviation.

requiring context switch during execution (e.g., a yield operation, intermediate termination, and synchronization) and, in the child-first case, an event where a parent thread is stolen by another scheduler. We note that no deviation is allowed with **RtC**.

Figure 8 illustrates the execution paths of **Full** and **C-Full** without deviation. Comparison of Fig. 8 with Fig. 3 shows that both **Full** and **C-Full** incur the following additional overheads compared with **RtC**, lowering the performance of **Full** and **C-Full** even when no deviation happens.

1. Save callee-saved registers on ULT invocation (`start_ctx()`).
2. Restore callee-saved registers on ULT completion (`end_ctx()`).
3. Manage call stacks for `thd->stack`.

To quantify the performance difference, we created a microbenchmark that controls the chances of deviation by adding a yield operation. Specifically, we ran a microbenchmark that creates and joins N empty ULTs as shown in Fig. 9. In this benchmark, n randomly chosen ULTs yield once, so $n/N\%$ of ULTs encounter deviations. We define a deviation possibility D as n/N and changed D by controlling n while fixing N to 4,096. We ran this microbenchmark on a single core of an Intel Skylake processor (see Section 4 for details).

Figure 10 shows the fork-join overheads regarding the deviation probability (D). Our result is the arithmetic mean of all iterations. Because **RtC** does not allow deviation, we draw a horizontal line which has the value at $D = 0\%$ (i.e., no deviation). The result shows that even when no deviation takes place, the overhead of **Full** is 1.7x higher than that of **RtC** because of the context management. We note that although their scheduling policies are different, **Full** and **C-Full** perform similarly because the expensive operations including register and stack management are common.

Ideally, **Full** would perform as well as **RtC** when no deviation occurs, and so would **C-Full**. However, it has been an open question whether this performance gap is inevitable

when employing full threading capabilities or whether other threading techniques offer different performance and capability trade-offs. In the next section we analyze the performance discrepancy and investigate threading techniques that exist between these two opposite directions, that are more efficient than **Full** and **C-Full** when no deviation happens, and that keep full threading capabilities.

3 Lightweight User-Level Threading Techniques with Dynamic Promotion

In this section we analyze the performance gap between fully fledged techniques (**Full** and **C-Full**) and **RtC** and explore intermediate threading techniques. The analysis uses the same microbenchmark presented in the preceding section (Fig. 9) and progressively cuts down the overheads of **Full** and **C-Full** at $D = 0\%$. Each step of the analysis finds a lightweight threading technique that has a different trade-off between performance with and without deviation and programming constraints.

3.1 Parent-First Scheduling

We first look at parent-first threading techniques. Our analysis reduces the overhead of **Full** toward that of **RtC** while keeping the capabilities of **Full**. Instruction breakdowns and performance data of all the parent-first methods are summarized in Fig. 16 and Fig. 17, respectively.

3.1.1 Removing Context Switch on Completion (*RoC*)

```
1  void start_ctx_RoC(ctx_t **self_ctx, void *stack, void (*f)(void *),
2                     void *arg) {
3    Push callee-saved registers
4    Push an instruction address
5    *self_ctx = stack_pointer
6    stack_pointer = stack
7    f(arg) // a user function is directly called.
8    return
9  }
10 void end_ctx_to_sched(void) {
11   end_ctx(g_sched_ctx)
12 }
```

Fig. 11. Pseudo assembly code of context switch in **RoC**.

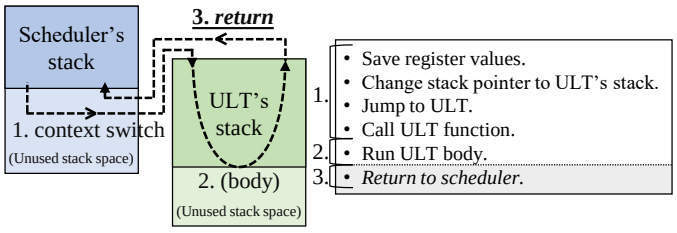


Fig. 12. Flow of **RoC** when no deviation happens. The difference from **Full** (Fig. 8a) is written in italic.

Our instruction analysis shows a large difference in instruction counts between **Full** and **RtC** at $D = 0\%$; even if no deviation occurs, **Full** performs context switch twice, imposing as many as 50 instructions. The first context switch from a scheduler to a ULT is necessary in order to make a scheduler resumable at any point. If no deviation occurs, however, the second manipulation of callee-saved registers is unnecessary since the register values of the scheduler are restored by a user-given thread function (`thd->f()`). A *return-on-completion* technique (**RoC**) exploits the fact that the first context switch is inevitable but the last one can be omitted if no deviation takes place during execution; **RoC** replaces the second context switch by a standard return procedure to reduce the context-switching cost on completion.

Figure 11 presents the pseudocode of a function that invokes **RoC**, and Fig. 12 illustrates its execution flow without deviation. `start_ctx_RoC()` first saves callee-saved registers (lines 3 and 4), changes a stack (lines 5 and 6) as `start_ctx()` does (Fig. 5), and directly calls a thread function `f()` (line 7). If a created thread has not encountered a deviation, the parent scheduler has never been resumed, so the **RoC** thread can simply return to a scheduler without restoring callee-saved registers saved at lines 3 and 4 because they were restored by `f()`. Thus, `start_ctx_RoC()` can return to a scheduler by a `return` instruction.[8] However, the scheduler cannot simply be resumed by a return procedure if deviations have happened because a deviation staled the callee-saved registers saved in `f()`. In order to address this issue without extra overheads, the return address stored in the call stack of `start_ctx_RoC()` is updated to `end_ctx_to_sched()` when a first deviation happens; if the **RoC** thread has confronted a deviation, `start_ctx_RoC()` does not directly return to the scheduler but jumps to `end_ctx_to_sched()` by `return` so that the context of the scheduler can be properly restored by `end_ctx()` (line 11). We note that only the first deviation needs to modify the return address, so succeeding deviation events do not incur any overhead.

When D is 0%, **RoC** omits one context switch per fork-join and successfully saves 24 instructions compared with **Full**, achieving 16% less overheads than does **Full**. However, **RoC** degrades performance when D is large (4% worse at $D = 100\%$) because when a deviation happens, **RoC** performs the same number of context switches but complicates the control flow.

3.1.2 Removing Context Switch on Invocation (*SS*)

```
1  void start_ctx_SS(ctx_t **self_ctx, void *stack, void (*f)(void *),
2                    void *arg) {
3    *self_ctx = stack_pointer
4    stack_pointer = stack
5    f(arg) // a user function is directly called.
6    return
7  }
8  void end_ctx_invoke_sched(void) {
9    stack_pointer = scheduler's stack_top
10   scheduler()
11 }
```

Fig. 13. Pseudo assembly code of context switch in **SS**.

Although **RoC** successfully skips register manipulations on completion at $D = 0\%$, saving a context on invocation makes **RoC** slower than **RtC**. We save the scheduler's context in order to resume it later, but if the scheduler does not need to preserve its state including local variables and its progress, we can freshly start a new one. We call this property of a scheduler *statelessness*. We propose a new threading technique *stack separation* (**SS**) that separates stacks but does not save a context of the scheduler on invocation, while this technique requires a stateless scheduler.

[8] We assume a return mechanism similar to that of the x86/64 ABI [26]; a `return` instruction pops an instruction address from the call stack and jumps to that address. Unlike the x86/64 instruction set [27], however, several architectures including ARM [28] and POWER [29] do not have such a multifunctional return instruction. Nevertheless, their ABIs [30], [31] adopt similar calling conventions, which save an instruction address at a predefined location at a stack frame boundary. Thus we can implement the same algorithm on these architectures by combining multiple instructions as most compiler-generated codes do in a function epilogue.

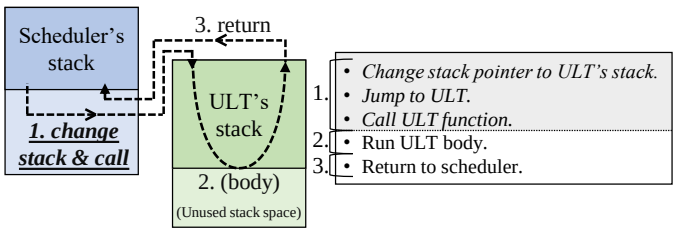


Fig. 14. Flow of **SS** when no deviation happens. The difference from **RoC** (Fig. 12) is written in italic.

Figure 13 shows the pseudocode of **SS**. After changing a stack pointer (lines 3 and 4), **SS** directly calls a thread function (line 5). As illustrated in Fig. 14, if no deviation happens, it returns to the scheduler with a standard `return` (line 6) as **RoC** does. If a deviation occurs, the return address in the call stack of the **SS** thread is updated so that **SS** jumps to a function without restoring the outdated scheduler's context. However, **SS** cannot resume the scheduler by `end_ctx_to_sched()` in Fig. 11 because **SS** does not save the scheduler's callee-saved registers. Instead, **SS** calls `scheduler()` on the stack of the original scheduler, which flushes all the local variables in the call stack and the progress stored in an instruction address.

SS further reduces 14 instructions compared with **RoC** when D is 0%, achieving 14% higher performance than **RoC** does. However, **SS** lowers performance if a deviation happens (7% slower than **Full** at $D = 100\%$) because **SS** essentially needs to rerun a scheduler from the beginning of the function, which is unnecessary if the scheduler context is properly saved.

Although **SS** performs better than **Full** and **RoC** at $D = 0\%$, **SS** imposes a programming constraint that requires a stateless scheduler, narrowing the applicability of **SS**. A random work-stealing scheduler [23] can be implemented as stateless, but we note that not all schedulers are trivially stateless; for example, a scheduler is not stateless if it saves counters in local variables to select a victim of work stealing or if it sleeps when work stealing fails continuously.

3.1.3 Lazy Stack Allocation (*Full-L*, *RoC-L*, and *SS-L*)

SS remains slower than **RtC**. We observe that **RtC** incurs fewer L1 and L2 cache misses than do the other techniques at $D = 0\%$ because **RtC** accesses only the scheduler's stack while each invocation of **Full**, **RoC**, and **SS** touches an independent call stack that is preallocated on creation. Such an eager stack allocation strategy is common in practice to facilitate management of a thread descriptor and a stack; it allows a runtime to reduce memory management operations by allocating together thread descriptors and their corresponding stacks (i.e., use part of a stack region as a descriptor). Nevertheless, this practice increases the memory accesses since each ULT invocation accesses a different stack area that is unlikely in caches. As a result, **Full**, **RoC**, and **SS** increase L1 and L2 cache misses at $D = 0\%$.

However, not all the ready ULTs need to have independent stacks; only simultaneously active ULTs require independent stacks. To reduce the memory footprint, we introduce a *lazy stack allocation* method (LSA) that decouples the management of thread descriptors and stacks and assigns a stack at invocation time. Since most ULTs are forked and joined sequentially when D is small, a call stack can be reused across thread invocation. **Full**, **RoC**, and **SS** can adopt LSA without changing their context-switching algorithms. We refer to these techniques by adding a suffix **-L** to their names.

We observe that **Full-L**, **RoC-L**, and **SS-L** achieve slightly higher performance than do the original techniques by successfully reducing L1 and L2 cache misses at $D = 0\%$; their numbers of L1 and L2 cache misses are almost the same as those of **RtC**. However, LSA adds 11 instructions to manage a stack and a thread descriptor independently, so LSA possibly degrades performance on different machines that have different instruction, memory, and cache costs. The results also show that the advantage of LSA becomes negligible as D gets higher since more ULTs need independent stacks; as a result, the additional allocation operations incurred by LSA lower the performance. We note that LSA promotes stack reuse and thus can reduce the memory footprint when D is small, which is evaluated in Section 4.1.4.

3.1.4 Removing Stack Change (*SC*)

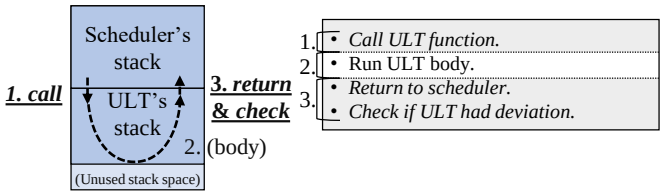
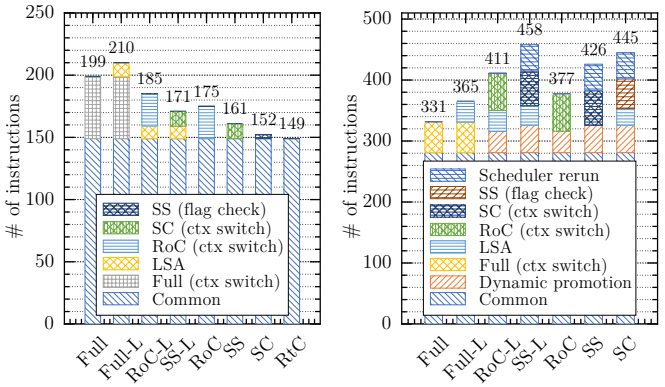


Fig. 15. Flow of **SC**. The difference from **SS** (Fig. 14) is written in italic.

Threading overheads still exist in the stack management, which fundamentally makes **SS-L** slower than **RtC**. If a scheduler is stateless, however, a scheduler can be spawned on top of the newly allocated stack, which eliminates management of both callee-saved registers and stacks. The technique that newly creates a scheduler has been adopted by some runtimes [32], [33], [34]. We refer to this technique as *scheduler creation* (**SC**). When an **SC** thread encounters a deviation for the first time, it spawns a ULT with a new stack and starts a scheduler on top of it. At the same time, the original scheduler currently running the **SC** thread is invalidated by updating a flag in order to keep the number of active schedulers. On completion, a scheduler checks the invalidation flag; if invalidated, it jumps to an active scheduler using `g_sched_ctx`.

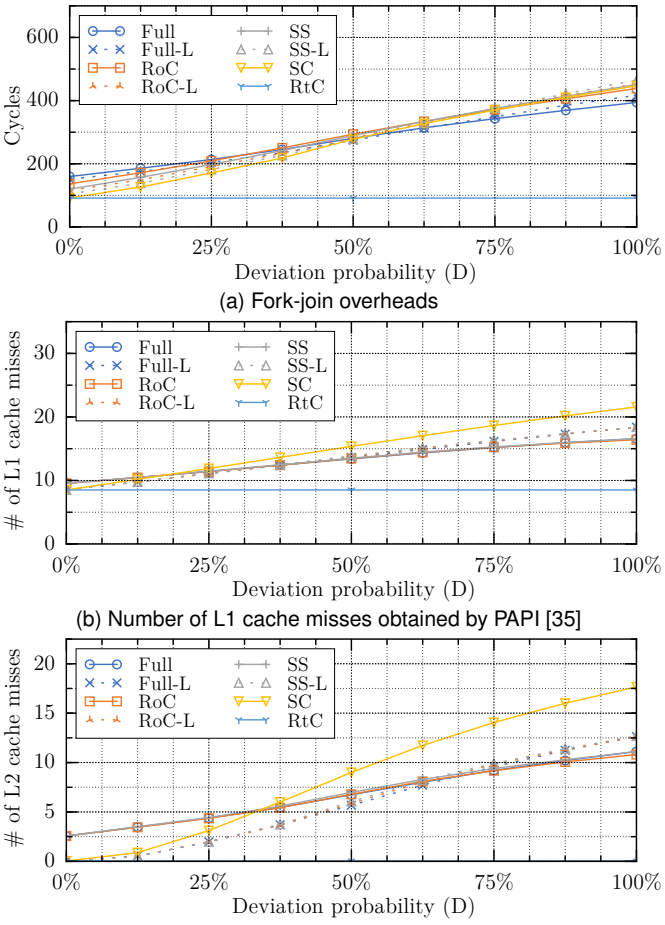
In addition to the requirement of a stateless scheduler as **SS** and **SS-L** have, **SC** imposes a new constraint on the stack size; because stacks are shared with **SC** threads and schedulers, the stack size of all **SC** threads must be the same as that of the scheduler, forcing users to adopt the largest stack size that fits all threads in a program. This constraint is significant when one application contains multiple types of threads each of which requires a different stack size.

Figure 16 summarizes the instruction breakdowns with and without deviation and Fig. 17 shows the performance and cache misses of eight parent-first threading techniques. Fig. 16a shows that at $D = 0\%$ **SC** adds only three instructions to check the invalidation flag. As a result, the overhead of **SC** is as small as that of **RtC** at $D = 0\%$ while **SC** supports all the threading capabilities that may cause deviations. However, restarting a scheduler on a new stack is expensive in terms of the number of instructions and memory accesses; hence, **SC** shows the worst performance among the seven methods at $D = 100\%$.



(a) $D = 0\%$ (b) $D = 100\%$

Fig. 16. Instruction breakdown of fork and join operations on Skylake (parent-first methods).



(a) Fork-join overheads

(b) Number of L1 cache misses obtained by PAPI [35]

(c) Number of L2 cache misses obtained by PAPI [35]

Fig. 17. Performance of the parent-first threading methods on Skylake. Almost no L3 cache miss happens in this experiment because each ULT accesses a small portion of a call stack; we therefore omit the data.

3.2 Child-First Scheduling

In this section, we apply the same analysis methodology to child-first techniques. We use the same microbenchmark to evaluate their overheads. Their instruction breakdowns and performance are summarized in Fig. 20 and Fig. 21.

3.2.1 Eager Stack Release (*C-Full-E*)

C-Full suffers from large L1 and L2 cache misses because each ULT has an independent stack. LSA, which allocates stack on invocation, seems promising to reduce cache misses incurred by stack accesses. However, LSA itself is not applicable to child-first techniques because creation and invocation are done in the same function (i.e., `create_thd()`). To promote stack reuse, we devise an *eager stack release* method (ESR) that frees stacks not when threads are joined (i.e., `join_thread()`) but on completion of ULTs. ESR allows consecutively spawned ULTs to reuse the same stack region if no deviation happens; however, ESR needs to decouple the management of thread descriptors and stacks, adding extra overheads to handle them separately. We call this technique **C-Full-E**.

Although ESR imposes 11 instructions for independent management of thread descriptors and stacks, **C-Full-E** successfully eliminates L2 cache misses and reduces L1 cache misses, achieving an overall performance improvement of 13% when no deviation takes place. In addition, as will be evaluated in Section 4.1.4, this stack reuse can dramatically reduce the memory footprint when the deviation probability is low. However, ESR fails to effectively reuse stacks and degrades performance by additional stack and thread descriptor management as the deviation probability increases.

3.2.2 Removing Context Switch on Completion (*C-RoC and C-RoC-E*)

```
1  thd_desc_t *create_thd(...) {
2    thd_desc_t *thd = allocate_thd_desc_t()
3    init_thd_desc(thd, ...)
4    thd->parent = g_current_thread
5    start_ctx_RoC(&g_current_thread, thd->stack, thd_wrapper, thd)
6    if (stolen_by_another_worker)
7      *(thd->stack + RETURN_ADDRESS_OFFSET) = end_ctx_to_sched
8    return thd
9  }
10 void thd_wrapper(thd_desc_t *thd) {
11   push_local_pool(thd->parent)
12   thd->f(thd->arg) // thd->f and thd->arg are given by users.
13   thd->is_finished = true
14   ctx_t *next_ctx = pop_local_pool_or_get_sched_ctx()
15   if (next_ctx == thd->parent->ctx)
16     return
17   else
18     end_ctx(next_ctx)
19 }
```

Fig. 18. Pseudo assembly code of context switch in **C-RoC**.

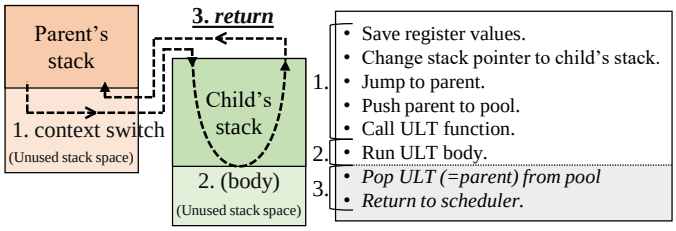
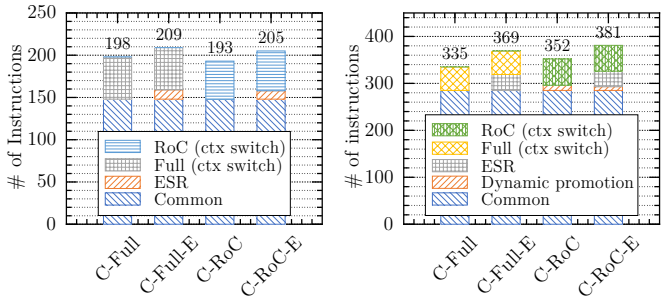


Fig. 19. Flow of **C-RoC** when no deviation occurs. We emphasize the difference from **C-Full** (Fig. 8b) by italicizing it.

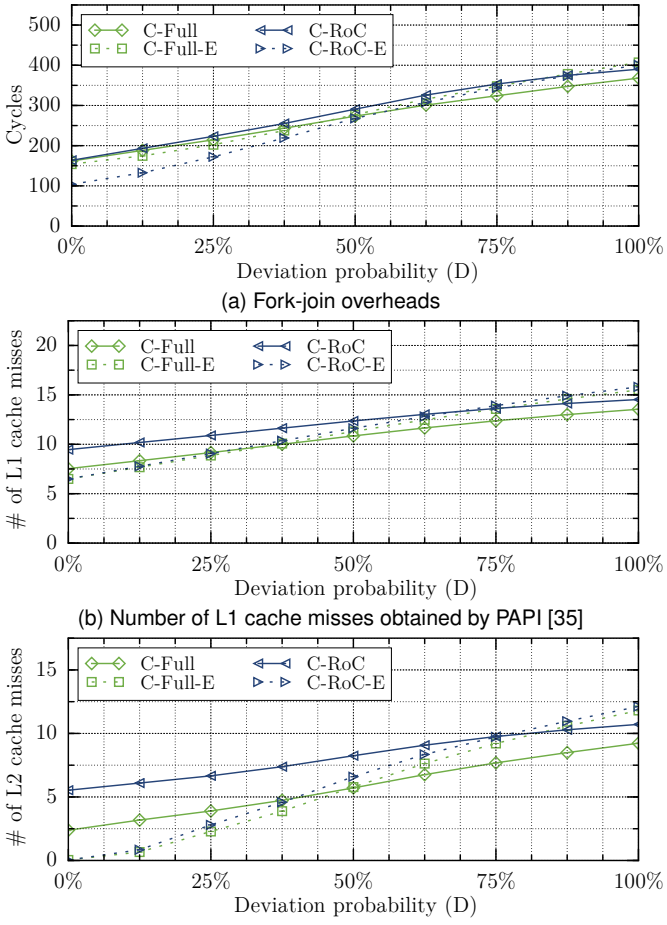
As **Full-L** does, **C-Full-E** manipulates callee-saved registers on both invocation and completion. Unlike parent-first scheduling, child-first scheduling must preserve the context of the parent ULT since it is controlled by the users. Hence, child-first scheduling needs to maintain an independent stack and manage callee-saved registers on invocation. If no deviation happens, however, the parent thread can be resumed by a return function using a *return-on-completion* technique. This technique is applicable to both **C-Full** and **C-Full-E**; we call them **C-RoC** and **C-RoC-E**, respectively. Child-first return-on-completion techniques, however, need to deal with a deviation caused by work stealing to the



(a) $D = 0\%$ (b) $D = 100\%$

Fig. 20. Instruction breakdown of fork and join operations on Skylake (child-first methods).

parent thread.[9] Therefore, **C-RoC** and **C-RoC-E** need an algorithm that allows a thief to update the return address in the call stack of the child ULT.



(a) Fork-join overheads

(b) Number of L1 cache misses obtained by PAPI [35]

(c) Number of L2 cache misses obtained by PAPI [35]

Fig. 21. Performance of the child-first methods on Skylake. Higher levels of caches do not suffer from cache misses in this experiment because each ULT accesses a small portion of a call stack; we therefore omit the data.

We first look at the algorithm of **C-RoC**. The pseudocode of **C-RoC** is shown in Fig. 18. The first context switch uses `start_ctx_RoC()` presented in Fig. 11; if no deviation happens, the child ULT can return to the parent (line 16). As **RoC** does, the first occurrence of any threading operation that causes a deviation updates the return address stored in the stack. In the case of **C-RoC**, however, deviations can be caused by work stealing; even if no context-switching operation is performed by the child ULT, the child may not simply return to the parent ULT if the parent has been stolen. To handle this case, the thief updates the return address of the child (lines 6 and 7), which does not exist in **RoC**. We note that there is no data race between an update by a thief (line 7) and reading a return address by a child (line 16) because the child ULT performs `return` only after taking the parent ULT (line 14). As illustrated by Fig. 19, **C-RoC** successfully removes callee-saved register management on completion when no deviation takes place.

In addition to changing the stack management, **C-RoC-E** requires a small modification to **C-RoC** because at line 7 in Fig. 18 a parent may update a child stack that has been already freed under the ESR policy. A thread descriptor of a child is, however, always available in `create_thd()` since the descriptor has not yet been returned to the caller of `create_thd()`. **C-RoC-E**, therefore, does not access a return address in the call stack but reads a member variable in a thread descriptor. This change makes an update by a thief safe but imposes additional overheads for reference in comparison with directly manipulating values in a call stack at $D = 0\%$.

The instruction breakdowns and performance of the four child-first threading methods are summarized in Fig. 20 and Fig. 21. Figure 20 shows that at $D = 0\%$ **C-RoC** and **C-RoC-E** in total reduce 5 and 4 instructions compared with **C-Full** and **C-Full-E**, respectively. However, **C-RoC** increases the memory footprint because the complicated operations in `thd_wrapper()` require larger stack space, which increases L1 and L2 cache misses. Overall, **C-RoC** is 10% slower than **C-Full** even if no deviation happens. **C-RoC-E** overcomes this issue of memory footprint by reusing stacks, which successfully trims down the overhead by 29% compared with **C-Full** at $D = 0\%$ while worsening performance by 7% at $D = 100\%$.

3.3 Summary

Table 1 shows the trade-off regarding performance and programmability. **Full**, **Full-L**, **RoC**, **RoC-L**, and all the child-first threading techniques have no programming constraints because they save a parent context, while **SS**, **SS-L**, and **SC** require stateless schedulers. **SC** has an additional constraint on stack size, which further narrows its applicability. **RtC** supports no threading capability that requires a context switch such as yielding, intermediate termination, and efficient synchronization. However, highly constrained threading techniques perform better if no deviation happens; in the case of parent-first scheduling, **RtC** and **SC** perform better than the others. We also note that in both parent- and child-first cases, threading techniques that show better performance at low D tend to perform worse at large D because if deviation happens, dynamic promotion methods that lazily manage the stack and callee-saved registers incur extra overheads than do eager methods. We note that these techniques can co-exist in a single threading library without impacting other techniques. We will discuss how to choose the best technique in Section 4.5.

3.4 Coverage of Our Techniques

Table 2 summarizes the coverage of our analysis. An area labeled with (*) in the table denotes an absence of practical

[9] It does not happen with parent-first scheduling since a scheduler, which corresponding to a parent in child-first scheduling, is never stolen by another worker.

TABLE 1
Summary of the twelve threading techniques

			$D = 0\%$			$D = 100\%$	Constraints
		LSA/ESR	Change stack?	# of Register Managements	Overheads	Overheads	
Parent-First	Full (*Fully Fledged Thread*)	No	Yes	2	Highest	Lowest	No
	Full-L (*Fully Fledged Thread (LSA)*)	Yes	Yes	2			No
	RoC (*Return on Completion*)	No	Yes	1			No
	RoC-L (*Return on Completion (LSA)*)	Yes	Yes	1			No
	SS (*Stack Separation*)	No	Yes	0			Scheduler must be stateless.
	SS-L (*Stack Separation (LSA)*)	Yes	Yes	0			Scheduler must be stateless.
	SC (*Scheduler Creation*)	Yes	No	0		Highest	Scheduler must be stateless. Stack size is shared.
	RtC (*Run to Completion*)	-	No	0	Lowest	-	No deviation is allowed.
Child-First	C-Full (*Fully Fledged Thread*)	No	Yes	2	Highest	Lowest	No
	C-Full-E (*Fully Fledged Thread (ESR)*)	Yes	Yes	2			No
	C-RoC (*Return on Completion*)	No	Yes	1			No
	C-RoC-E (*Return on Completion (ESR)*)	Yes	Yes	1	Lowest	Highest	No

TABLE 2
Coverage of our analysis

Change Stack?	# of Register Managements	LSA/ESR	Parent-First	Child-First
Yes	2	No	Full	C-Full
		Yes	Full-L	C-Full-E
	1	No	RoC	C-RoC
		Yes	RoC-L	C-RoC-E
	0	No	SS	(*2)
		Yes	SS-L	
No	2	No	(*1)	
		Yes		
	1	No		
		Yes		
	0	No	(*3)	
		Yes	SC/RtC	

techniques. In the following, we explain reasons why these techniques are infeasible for general threading libraries.

Saving Registers (*1): Our analysis does not include threading techniques that do not change a stack but explicitly maintain callee-saved registers on invocation. Intuitively, if a stack is shared between a parent and a scheduler, resuming a scheduler is prohibitive since a scheduler can potentially overwrite the invoked stack. On the other hand, if we totally rerun a new scheduler as **SC** does, storing callee-saved registers is pointless. In the past, however, such techniques have been proposed for child-first scheduling [36], [37], [38], [39]. We note that these techniques are not suitable for building threading libraries because compiler modifications are required. We discuss their techniques in Section 5.

Restarting Parent ULTs (*2): With parent-first scheduling, **SS**, **SS-L**, and **SC** restart a stateless scheduler on deviation. This technique is not applicable to child-first threads, however, since parent ULTs are in most cases not stateless; rerunning a parent ULT loses not only the result computed by the parent ULT but also a child thread descriptor if the parent is in the midst of the thread creation function. This is an impractical restriction as a thread, so we do not show child-first techniques that require stateless parent ULTs.

Eager Stack Management for SC (*3): From the viewpoint of stack management, **SC** follows the LSA policy; **SC** allocates a stack for a scheduler not on creation but on deviation. One might suggest allocating a stack and a thread descriptor together on creation for **SC**, but such an eager stack allocation strategy does not work for **SC**. **Full**, **RoC**, and **SS** always keep the same pair of a stack and a thread descriptor, while **SC** needs to decouple the management of the stack and thread descriptor since a stack required on deviation is assigned to a new scheduler, not a thread associated with a thread descriptor.

4 Evaluation

In this section, we evaluate the performance of all the threading techniques presented in the preceding sections with a microbenchmark and three fine-grained parallel applications. All the parent- and child-first threading techniques were implemented in Argobots [11], a highly optimized user-level threading library. Our experimental environments are described in Table 3. All the programs were compiled with `-O3`. We set the same stack size (64 KB for ExaFMM and 16 KB for the others) to both ULTs and schedulers to the advantage of **SC**. All results reported in this paper are the arithmetic mean. The error bars in the charts indicate the 95% confidence intervals.

4.1 Fork-Join Microbenchmark

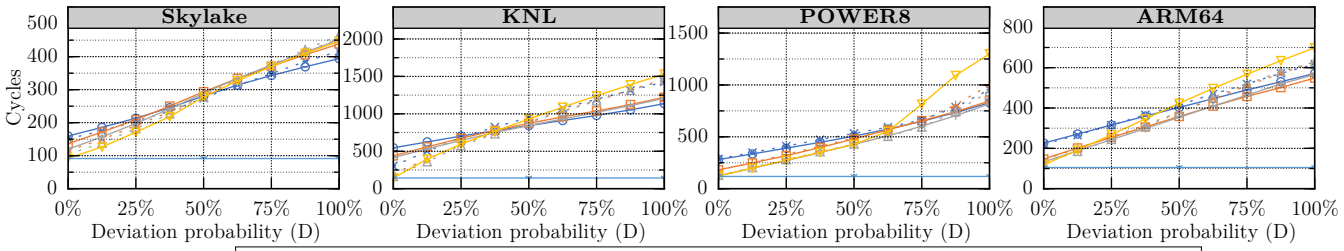
We first evaluate the threading overheads with the fork-join microbenchmark used in the preceding sections; the code is presented in Fig. 9. This microbenchmark repeats creating and joining N ULTs on a single worker. Deviations are artificially introduced by `yield_thd()`; the deviation probability D is calculated by n/N, where n ULTs uniformly selected out of N ULTs yield once. We used a lightweight private pool [11] to minimize overheads of pool operations. In the microbenchmark, the set of N fork-join operations was repeated $2^{19}/N$ times and obtained the average of the execution time. The result of **RtC** is at $n = 0$ (i.e., $D = 0\%$) because **RtC** cannot yield. We ran this microbenchmark 50 times on Skylake, KNL, POWER8, and ARM64.

4.1.1 Performance with Different Deviation Probabilities

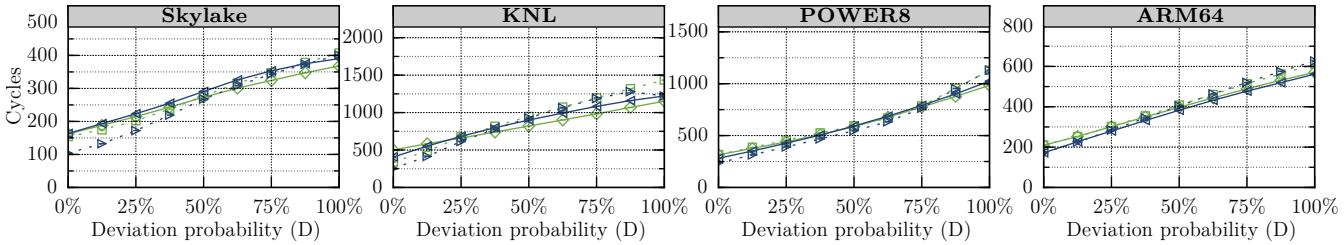
Figure 22 shows the results with various D values where N is fixed to 4,096 in order to see how the deviation probability affects the threading overheads. For better visibility, we separate results by scheduling type; Figure 22a shows the parent-first techniques while Fig. 22b plots only the child-first ones. First, all the results indicate the same performance trend: **SC**, **SS**, and **RoC** outperformed **Full** at $D = 0\%$ because these dynamic promotion techniques alleviate the context management overheads when no deviation takes place. In the case of child-first scheduling, at $D = 0\%$ **C-RoC** outperformed **C-Full** on KNL, POWER8, and ARM64 by reducing context-switching overheads, while it degraded performance on Skylake because of complex control as we discussed in Section 3.2.2. LSA and ESR (**-L** and **-E**)

TABLE 3
Experimental environments used in the paper.

Name	Skylake	KNL	POWER8	ARM64
Processor	Intel Xeon Platinum 8180M	Intel Xeon Phi 7210	IBM S822LC (10 cores)	AMD Opteron A1120
Architecture	Skylake	Knights Landing	POWER8 LE	ARMv8-A
Frequency	2.5 GHz	1.3 GHz	2.9 GHz	1.7 GHz
# of sockets	2	1	2	1
# of cores	56	64	20	4
# of HWTs	112	256	160	4
Memory	396 GB	198 GB	130 GB	8 GB
OS	Red Hat 7.5	Red Hat 7.5	Red Hat 7.6	openSUSE 42.2
Compilers	Intel Compiler 17.2.174	Intel Compiler 17.2.174	IBM XL Compilers 16.1.1	GNU Compilers 4.8.5



(a) Parent-first scheduling



(b) Child-first scheduling

Fig. 22. Cycles per fork-join with various D values ($N = 4{,}096$).

mitigated cache misses at the cost of additional stack management overheads, with elevated performance overall. The performance of **SS-L** and **SC** was close to that of **RtC**, but these threading techniques have programming constraints as discussed in Section 3.

On the other hand, at a larger D, **SC**, **SS**, **RoC**, and **C-RoC** were slower than the traditional fully fledged techniques (**Full** and **C-Full**) because the dynamic promotion techniques lose their advantages and become extra overheads. LSA and ESR (**-L** and **-E**) further degraded the performance since they no longer promote stack reuse and result in additional overheads to manage thread descriptors and stacks independently.

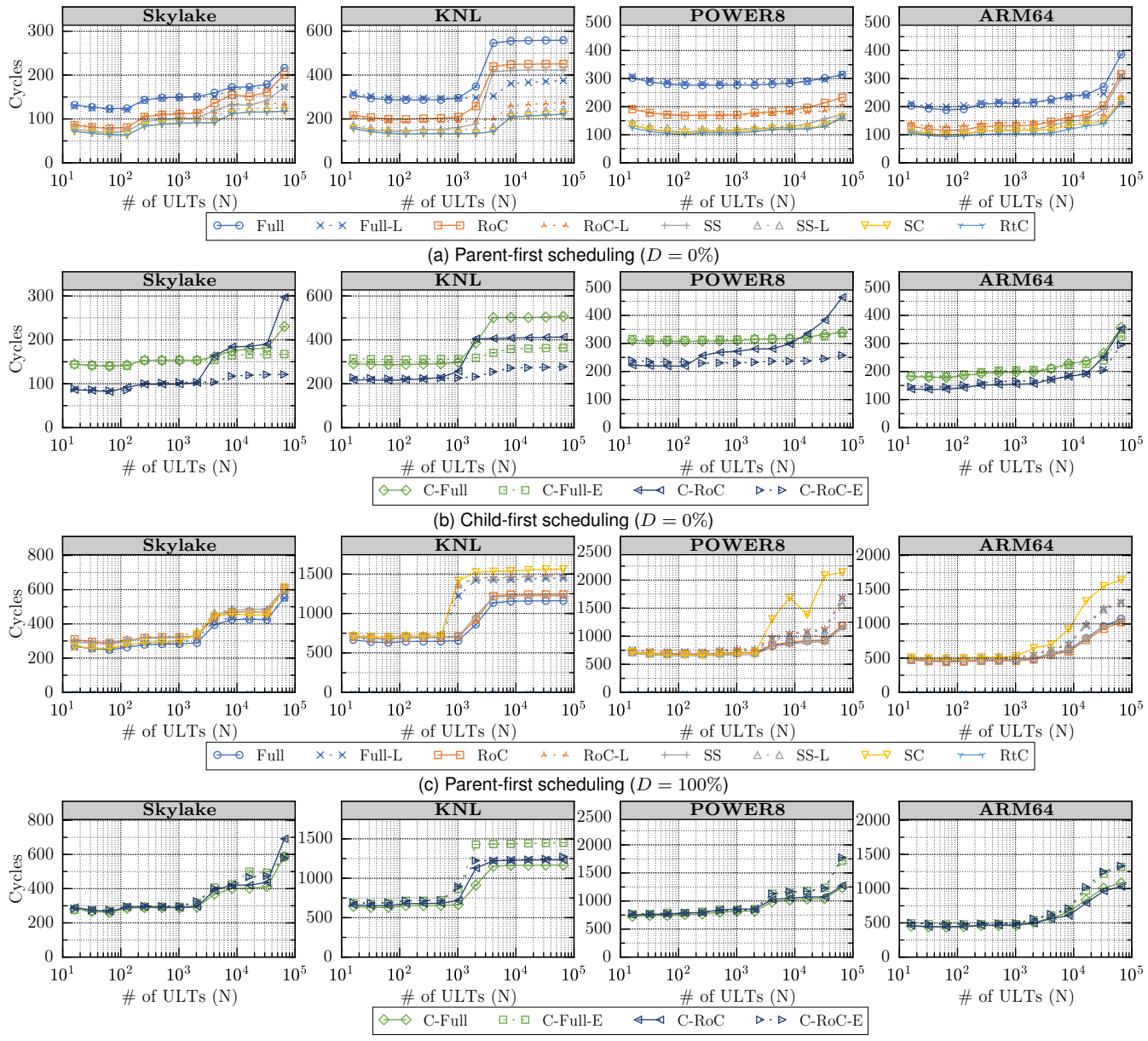
Although there is a fundamental scheduling difference between parent- and child-first techniques, the results of corresponding techniques (e.g., **RoC** and **C-RoC**) are similar because the expensive context-switching operations are common; in terms of context-switching overheads, there is no significant performance difference between parent- and child-first scheduling, while the applicability of dynamic promotion techniques is limited in the child-first cases. We note that, as pointed out by vast literature (e.g., [40], [41], and [42]), the scheduling policies have been known to affect the application performance. KMeans and ExaFMM in our evaluation showcase the difference in application-level performance, while in both cases the dynamic promotion methods enhance performance by reducing threading overheads.

4.1.2 Performance with Different Numbers of ULTs

With the same microbenchmark, we examined the effect of the dynamic promotion techniques by varying ULT counts (N) while fixing D to 0% and 100%. Figure 23 shows fork-join overheads with different N. Overall the performance trend is the same; at $D = 0\%$ the dynamic promotion techniques (**SC**, **SS**, **RoC**, and **C-RoC**) are faster than fully fledged techniques (**Full** and **C-Full**). With smaller N, however, LSA and ESR (**-L** and **-E**) are insignificant because stack accesses hit caches without LSA and ESR; on the contrary, decoupling the management of thread descriptors and stacks negatively affects the performance even at $D = 0\%$. The results indicate that if only few ULTs are used in the runtime, LSA and ESR do not contribute to performance improvement and possibly just impose additional overheads.

4.1.3 Performance on Different Architectures

The effectiveness of the dynamic promotion techniques varies on different architectures. Figure 22a and Figure 22b indicate that KNL shows the largest performance difference at $D = 0\%$; the gaps between **Full** and **RtC** are 1.7x, 3.8x, 2.4x, and 2.2x while speedups of **C-RoC-E** over **C-Full** are 1.4x, 2.0x, 1.3x, and 1.3x on Skylake, KNL, POWER8, and ARM64, respectively. This difference comes from the design of KNL. In comparison with Skylake, which is a general-purpose Intel CPU, in spite of the identical calling convention, KNL showed a larger gap because of its throughput-oriented architecture; KNL poorly performs pointer-based operations with many branches and noncontiguous memory accesses, both of which highly impact



(a) Parent-first scheduling ($D = 0\%$)

(b) Child-first scheduling ($D = 0\%$)

(c) Parent-first scheduling ($D = 100\%$)

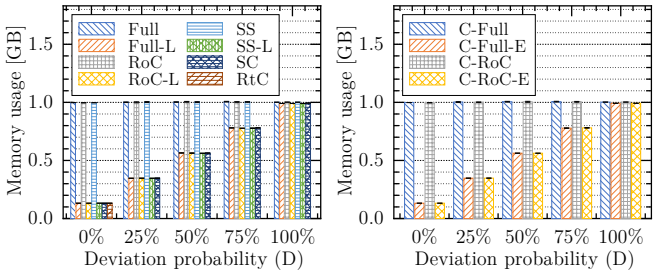
(d) Child-first scheduling ($D = 100\%$)

Fig. 23. Cycles per fork-join with various numbers of ULTs (D is fixed to either 0% or 100%).

the context-switching performance. The context-switching overhead on POWER8 is also high. For example, at $D = 0\%$ and $N = 128$, the performance gap between **Full** and **RtC** is 2.7x while the gaps of Skylake, KNL, and ARM64 are between 1.9x and 2.2x. Context switching on POWER8 is costly because more instructions are required to save its larger context; the context size of POWER8 is as large as 528 bytes because its ABI marks more registers as callee-saved [31]. In contrast, the size of x86/64 and ARMv8-A is only 64 bytes and 176 bytes, respectively [26], [30]. We observe that the dynamic promotion techniques are more effective on architectures that are throughput oriented or have a large context.

4.1.4 Memory Usage with Different Deviation Probabilities

We measured the memory usage of each threading technique to evaluate the impact of stack reuse. We ran the same benchmark with various D values where N is fixed to 65,536. Figure 24 shows the maximum memory consumption on Skylake obtained with `ru_maxrss` of `getrusage()`.



(a) Parent-first scheduling (b) Child-first scheduling

Fig. 24. Total memory consumption on Skylake where $N = 65{,}536$.

Results of the other machines were almost identical and hence are omitted. Figure 24 shows that at smaller D threading techniques with LSA and ESR (**Full-L**, **RoC-L**, **SS-L**, **C-Full-E**, and **C-RoC-E**) significantly reduced the memory footprint by reusing a stack, as do **RtC** and **SC**. On the other hand, methods without stack reuse always consumed about 1 GB (equal to the stack size (16 KB) multiplied by $N = 65{,}536$). Although the modern hardware has abundant

memory resources, this difference can be more significant when the stack size is set to a large value or more ULTs are created. In such a case, dynamic promotion techniques with LSA and ESR would be beneficial.

We evaluated the overheads of the threading techniques with a microbenchmark. The following evaluates the benefits of the dynamic promotion techniques with three fine-grained parallel applications: KMeans, ExaFMM, and Graph500. These applications utilize a context switch to efficiently schedule other ready work when currently running ULTs need to wait for locks, completions of other ULTs, or communications.

4.2 OpenMP-Parallelized KMeans

OpenMP is one of the most widely used parallel programing systems for multithreading. OpenMP offers threads and tasks as yieldable parallel units (i.e., `barrier` and `taskyield`[10]), so they were created as fully fledged threads in ULT-based OpenMP systems [6], [46], [47]. However, not all OpenMP threads and tasks encounter deviations in real programs (e.g., no task scheduling during execution). The dynamic promotion techniques are expected to improve performance when deviations rarely happen. We used OpenMP-parallelized KMeans for evaluation.

KMeans is a machine learning algorithm that partitions N data points into K clusters. Our benchmark is based on the KMeans implementation found in NU-MineBench [48]. In the KMeans algorithm, a point is considered belonging to a cluster with the nearest center. The algorithm first randomly distributes each center of K clusters and repeats updating the cluster centers to the centroids of their points until the positions of the centers get stable enough. The computation of the new centroids is parallelized by a simple method adopted by Chabbi et al. [49]; in our benchmark, each of N ULTs is associated with a data point and updates the partial sum of the centroid of the nearest cluster. At the end of an iteration a master ULT sums up the partial results. The partial sums are shared among workers, so the updates are protected by locks to avoid data race.

To control the lock granularity, we artificially change the number of replications per cluster, which we denote by r. When $r = 1$, each cluster has one partial sum protected by a corresponding lock, so any attempt to update the partial sum of the same cluster incurs lock contention. Creating multiple partial sums increases the reduction cost at the end of iterations but alleviates contention. When $r > 1$, every cluster has r partial sums each of which is accessed by only r/W workers, where W is the number of workers. Accordingly, no contention occurs if $r = W$.

The kernel of KMeans was parallelized with OpenMP by a nested parallel loop; the outer loop creates W OpenMP threads each of which spawns N/W tasks in the inner loop. An Argobots-based OpenMP runtime library called BOLT [50] maps OpenMP threads and tasks to ULTs. We used KNL for the evaluation, so W was set to 64 in this benchmark. We built the program with Intel compilers,[11] while we needed to apply manual vectorization to the compute kernel to exploit SIMD units in KNL. We used the first 10% data of KDD Cup 1999 [51]; our experiment classifies $N = 5.0 \times 10^5$ points, each of which has 41 floating-point features,[12] into $K = 24$ clusters as instructed by the original problem statement. We changed r from 1 to 64 and measured the performance with different ULT types.

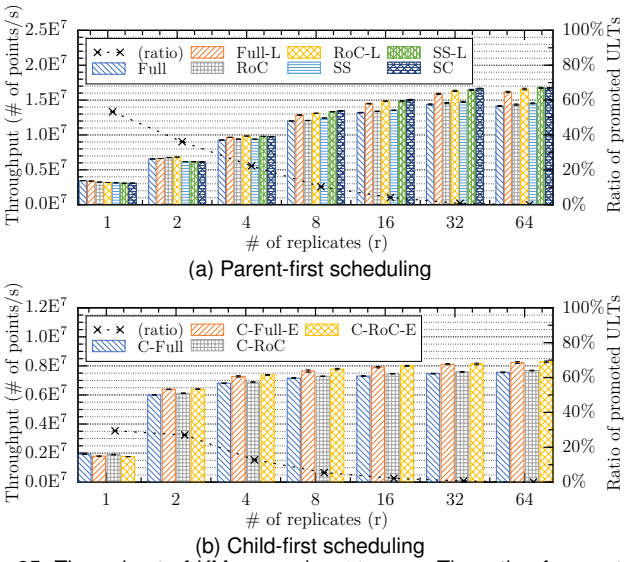


Fig. 25. Throughput of KMeans using 64 cores. The ratio of promoted ULTs is calculated by dividing the number of promoted ULTs by the number of created ULTs during execution. We obtain these results with **SC** for parent-first scheduling and **C-RoC-E** for child-first scheduling. Other dynamic promotion techniques show similar results.

To exploit better locality, we set the OpenMP's `close` affinity for the parent-first threading techniques. The affinity of ULTs can be implemented by limiting the access of a specific pool; BOLT implements the OpenMP's affinity by limiting ULTs associated with OpenMP threads to be scheduled by a specific worker associated with a specific core [50]. Although this strategy works well in a parent-first case, such an affinity setting inhibits dynamic load balancing in a child-first case. Consider a case where r is 64 and no deviation happens in innermost ULTs (an inner OpenMP tasks). Under the child-first scheduling policy, not a child ULT but a parent ULT (i.e., an outer OpenMP thread) is made stealable. Because of the affinity setting, however, a parent ULT cannot be scheduled other than by a specific worker, disabling dynamic load balancing across workers. Thus, we disabled the affinity setting for the child-first threading techniques.

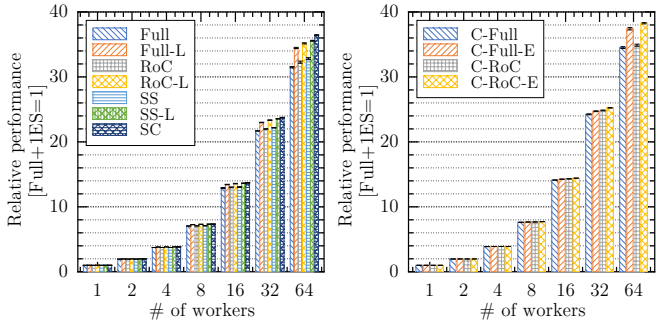
Figure 25 shows the average throughputs of 64 executions each of which repeats the KMeans algorithm five times after a warm-up (one execution). An increase in replicates alleviates lock contention and reduces the deviation probability, elevating overall performance. At a larger r, LSA and ESR (**-L** and **-E**) enhances performance. Reducing context-switching overheads (**SC**, **SS-L**, **RoC-L**, and **C-RoC-E**) further improves throughputs. With fewer replicates, fully fledged techniques perform better, but the absolute performance is worse because of significant lock contention. The results show that the dynamic promotion techniques speed up programs if deviations happen infrequently, whereas the threading overheads often get negligible when deviations are frequent because the causes of deviation become the performance bottleneck. We also note that although the

[10]The specification allows no operation for `taskyield` [43], while several studies pointed out the usefulness of yieldable tasks [44], [45]. In this benchmark, we assume tasks yield at `taskyield`.

[11]BOLT is a runtime library compatible with LLVM and Intel OpenMP compilers, so the compiler modification is unnecessary.

[12]We arbitrarily map string-typed values to floating-point values.

dynamic promotion methods improve performance with both parent- and child-first scheduling policies, the KMeans algorithm prefers parent-first scheduling because it suits the OpenMP's affinity setting.



(a) Parent-first scheduling (b) Child-first scheduling

Fig. 26. Relative performance of ExaFMM on KNL. The baseline is the performance of **Full** with a single worker.

4.3 ExaFMM

ExaFMM [52] is a highly optimized $O(N)$ N-Body solver using a fast multiple method. In the kernel, a tree is traversed in a divide-and-conquer manner, and leaf nodes calculate the actual forces. Recursive divide-and-conquer task parallelism has been known to efficiently parallelize the ExaFMM kernel [53]. Deviations can happen while waiting for completion of child ULTs, but they never occur in leaf nodes because they just perform computation without synchronization. The most efficient solution seems mapping leaf nodes to **RtC** and internal nodes to non-**RtC** ULTs. However, this optimization requires identifying leaf ULTs on creation, which is not only cumbersome but also expensive if the leaf condition is complicated. The dynamic promotion techniques alleviate the programmers' burden without hurting performance.

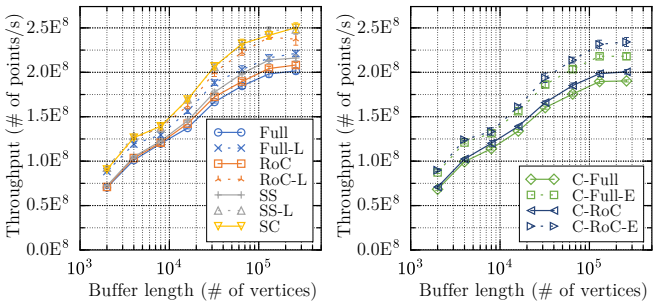
We ran ExaFMM ten times on KNL with `--ncrit 16 -t 0.15 -P 4 --dual -n 524288` as arguments. As the number of workers changed, we adjusted `--nspawn` to keep the number of created ULTs per worker constant (within 5% of error) while `--nspawn 256` was set with 64 workers. We measured the performance of the tree traversal where the program spends more than 90% of the total execution time. We manually vectorized the compute kernels to efficiently utilize SIMD units in KNL. To reduce internal nodes, we changed the way of work decomposition and collapsed internal nodes in the traverse tree while keeping the computation of leaf nodes.

Figure 26 presents the performance of ExaFMM with different numbers of workers. Since the deviation probability is low (regardless of the number of workers, approximately 1.5% of ULTs are dynamically promoted), the dynamic promotion techniques achieved better performance. LSA and ESR (**-L** and **-E**) improved performance with 64 workers. Reduction of context-switching overheads contributes to a 9% speedup for parent-first (**SC** over **Full-L**) and 2% for child-first scheduling (**C-RoC-E** over **C-Full-E**). This ExaFMM showcases the merit of child-first scheduling; the child-first methods overall perform better than the parent-first ones because child-first scheduling can efficiently exploit locality when parallelism is deep and narrow [12].

We note that the dynamic promotion techniques are suitable for divide-and-conquer recursive parallelism; in a k-ary tree approximately $1 - \frac{1}{k}\%$ of ULTs are leaves;[13] and therefore if no deviation happens in leaf ULTs (e.g., no yielding), D is approximately $\frac{1}{k}\%$. The results in Fig. 22 show that the dynamic promotion techniques perform better than the fully fledged threads when D is less than 30 to 50%. Because D is at most 50% ($k = 2$), the dynamic promotion techniques are beneficial in most cases.

TABLE 4
Experimental environment of Graph500

Processor	Intel Xeon Phi 7230	Architecture	Knights Landing
Frequency	1.3 GHz	# of sockets	1
# of cores	64	# of HWTs	128
Memory	99 GB	OS	CentOS 7.6
Compilers	Intel Compiler 17.0.4	Interconnect	Intel Omni-Path



(a) Parent-first scheduling (b) Child-first scheduling

Fig. 27. Traversed edges per second of Graph500 using 1,024 cores.

4.4 Distributed Graph500

Graph500 [54] is a well-known benchmark that traverses a distributed graph in a breadth-first manner. Our Graph500 is based on the reference implementation of MPI+Thread found in [55]. Since each process has only a part of the whole graph, interprocess communication is necessary in order to visit vertices in remote subgraphs. Specifically, in each iteration, every process repeats visiting adjacent vertices. A process can update a vertex if locally owned, while it needs to send a message to another process if the vertex exists in a remote node. In order to avoid the finest communication, message aggregation is commonly adopted. Each process has buffers associated with all the target ranks to store visit messages; messages are sent only when a buffer gets full and thus needs to be flushed. Because the bottleneck of Graph500 is communication, hybrid parallelism is often used to reduce the intranode communication overheads. MPI+Thread, where ULT is used as Thread, has been studied to exploit fine-grained communication on a distributed system [56], [57], because a ULT can efficiently switch to another ULT when an MPI function blocks. We note that the current state-of-the-art MPI+Thread implementation [58] sometimes acquires a lock even in nonblocking MPI calls (e.g., `MPI_Isend()` and `MPI_Test()`) since MPI functions need to periodically handle global progress for active messages and nonblocking collectives, which is typically protected by a global lock. The dynamic promotion techniques are expected to reduce overheads in cases where ULTs do not call an MPI function or happen to avoid lock contentions in the MPI runtime.

[13] Denote the number of internal nodes in a task tree N and the number of leaf nodes n. When $N = 1$, (N, n) is $(1, k)$. Because 1 leaf can be replaced with 1 internal node and k leaves, (N, n) becomes $(N, N(k-1)+1)$. Hence, the ratio of leaf ULTs is calculated as $\frac{n}{N+n} \approx 1 - \frac{1}{k}$ with a larger N.

We parallelized the Graph500 implementation in [55] with Argobots and evaluated the performance over Argobots-aware MPICH [58]. We associated one visit with a ULT to focus on threading overheads. A buffer length B is a parameter to control the communication granularity; with a larger B, more memory is consumed, but more messages are aggregated. We set a scale factor to 26, so the whole graph over nodes consists of 2^{26} vertices. We executed this benchmark on 16 KNLs described in Table 4. We spawned a single MPI process per node, each of which ran 64 workers, so 1,024 workers were used in total. Figure 27 shows the averages of ten times execution. In this setting, the ratio of promoted ULTs is less than 1.0%, highlighting the efficacy of the dynamic promotion techniques. The fully fledged (**Full** and **C-Full**) techniques should perform better with extremely small B and higher deviation probability, while the communication overheads would mask their benefit.

4.5 How to Choose the Best Threading Technique

This paper investigated several user-level threading techniques that have different performance characteristics and programming constraints. As all the threading techniques can coexist in a single library, users and developers can choose the suitable techniques from Table 1 for their assuming workloads. Practically, the first decision should be a choice of either parent- or child-first scheduling. As demonstrated in our evaluation, parent-first scheduling tends to perform better for shallow parallelism (e.g., loop parallelism in KMeans) while a child-first scheduling is preferred when the parallelism is deep and nested (e.g., divide-and-conquer parallelism in ExaFMM). More sophisticated strategy would be a mixed scheduling policy [59], which is out of scope of this work. The optimal technique under a specific scheduling policy should be chosen based on scenarios regarding the number of created ULTs, typical deviation probability, and required threading capabilities, all of which depend on their algorithms, machines, and inputs.

If the user has no idea about the application behavior, we recommend **RoC-L** for parent-first scheduling or **C-RoC-E** for child-first scheduling; they perform well at low deviation probability with minimum memory footprint while retaining all the threading capabilities. Nevertheless, among fully capable threads, these two do not always perform the best. When deviation probability is high, they are slower than **Full** and **C-Full**. As we have seen in the evaluation, however, when deviations are frequent, threading overheads often become negligible because events that cause deviations (e.g., lock contention and blocking communication) hide the benefit of lightweight threads. **RoC** and **C-RoC** also outperform **RoC-L** and **C-RoC-E** when deviations rarely happen, and fewer ULTs are used because LSA and ESR are not effective when stack accesses hit caches. Although our microbenchmark uses an empty function for a thread function, real thread functions are likely to require larger function stacks for computation, rendering LSA and ESR more beneficial. We recommend **RoC-L** or **C-RoC-E** in general, but the most promising approach is the automatic selection of the best threading techniques, which is one direction of our future work.

5 Related Work

Although numerous parallel systems have adopted ULTs as an implementation of lightweight parallel units, the focus of the past papers on ULT-based systems is not a threading technique but other components such as programmability, usability, portability, abstraction, and other performance optimizations such as scheduling and thread pool implementations. The performance comparison between these parallel systems measured only the overall performance [60], [61] and fundamentally lacked a detailed performance analysis of threading techniques. This section describes notable work out of countless studies on ULTs from the aspect of user-level threading techniques.

Fully Fledged Threads: Fully fledged threads are widely used to implement ULTs with full threading capabilities. For example, Qthreads [4], Nanos++ [5], Converse [7], MassiveThreads [10], and Argobots [11] are well-known threading libraries that use fully fledged threads. Their stack management policies are different, however. For example, Converse 6.9.0, MassiveThreads 1.00, and Argobots 1.0rc2 employ a parent-first fully fledged ULT without LSA while Qthreads 1.15 and Nanos++ 0.15 implement it with LSA. MassiveThreads 1.00 also supports a child-first thread, which is implemented with ESR. The trade-off disclosed by this paper would be helpful for these runtimes to choose the best thread implementation based on their assuming workloads. Evaluating the performance impact of choosing the optimal technique in these libraries is lifted as our future work.

Saving Registers: We did not evaluate techniques that do not change stacks but only save registers, but some previous studies including LazyThreads [36], StackThreads/MP [37], [38], and Fibril [39] proposed such techniques for child-first scheduling. Cilk 1.0-3 over Tapir/LLVM [62] is an actively developed multitasking framework that adopts this idea. Their approaches assume the following premises:

1. All local variables in the stack are addressed by a frame pointer instead of a stack pointer.
2. The call stack is not dynamically grown after a function prologue.
3. All threads are joined in a function that creates them (i.e., fully strict computation [22]).

Under these premises, a parent can call a child function on top of the parent stack after saving (or clobbering) callee-saved registers. The algorithm works as follows. If no work stealing happens, the child just returns to the parent. When another worker steals the parent ULT, the thief worker restores the original registers while setting a newly allocated stack to a stack pointer and resumes the parent on top of the new stack. Premise 1 guarantees that spaces for all local variables have already been allocated or reserved before the child invocation and these locations are referenced by a frame pointer. Premise 2 assures no stack growth, so a parent thread will not erode the stack used by the child thread. We note that premise 2 allows function calls because the stack address of a new function is based on a stack pointer. Premise 3, which narrows the expressiveness of parallelization, is required in order to prevent the caller of the parent ULT from overwriting the stack of the child thread prior to the completion of a child.

However, premises 1 and 2 require compiler modifications, and therefore StackThreads/MP [38],

LazyThreads [36], and Cilk [20] modified a compiler. Yang and Mellor-Crummey [39] tried to avoid compiler modifications by adding a GCC compiler flag, `-fno-omit-frame-pointer`, but it does not guarantee premise 1. Unfortunately, the current popular C compilers do not provide a flag that guarantees premises 1 and 2. These techniques are not evaluated in this paper because our work targets threading techniques without compiler modifications.

Stack Separation: Some studies have proposed methods that omit register manipulations but change only the stacks. Their approaches are different from ours in that a thread invocation function adopts a special calling convention that only marks registers for a stack pointer and an instruction address as callee-saved (e.g., Intel CilkPlus [63]). This approach can be seen as a technique that utilizes a calling convention to save all the necessary registers (including registers marked as callee-saved in widely adopted ABIs). This approach requires patching a compiler to recognize a custom calling convention, whereas neither **SS** nor **SS-L** requires compiler modification.

Scheduler Creation: A few parallel systems have adopted the scheduler creation technique. Chores [32] and Wool [33] are parent-first threading libraries that utilized this method to reduce threading overheads. Concurrent Cilk [34] is a child-first threading library that adopted this technique to implement a yield feature in Intel CilkPlus [64]. The past work, however, solely implemented the scheduler creation technique and thus lacked performance comparison and analysis of programming constraints. We also note that their approaches specially handle ULTs that encountered deviations, so promoted ULTs are differently scheduled from unpromoted ULTs. Our techniques uniformly schedule all ULTs including promoted **SC** threads.

Run-to-Completion Threads: Numerous runtime systems including Filaments [9], Qthreads [4],[14] and Argobots [11] support a run-to-completion thread in order to eliminate all the cost associated with user-level context switch. OpenMP *task* implementations found in the popular OpenMP runtimes [1], [2], [3] and `task` in Intel TBB [65] are essentially classified as **RtC** threads but not "run to completion" in a narrow sense because they can wait for the completion of children.

In general, **RtC** is lightweight and easy to implement, but its constraint significantly limits the applicability because it cannot perform a context switch at an arbitrary point. Several papers have argued yieldable threads in non-yieldable threading packages from performance and programmability perspectives. For example, Zakian et al. [34] showed that a yield operation in Cilk [20] enables efficient blocking communication and synchronization while several papers on OpenMP [44], [45] reported the same benefits of yieldable OpenMP tasks. Graph500 in our evaluation is a good example that **RtC** cannot execute; removing a yield operation from a polling loop in the MPI runtime might cause a deadlock.

Other Threading Techniques: Several threading techniques cannot be classified into the categories above. Sivaramakrishnan et al. [66] proposed MultiMLton, which allows relocation of function stacks. This technique might be applicable to functional languages, but it can hardly support C/C++ programs. Cilk-M [21] enables stack copying by modifying OS to expose the same address space so that a pointer reference to a call stack is valid after copying a stack. This technique requires OS modification. Tascell [67] is a compiler-based technique adopting a lazy task creation policy. This technique invokes threads in a sequential manner and lazily creates logical threads if necessary by backtracking call stacks. Acar et al. [18] propose a threading technique that lazily creates parallel threads at a *heartbeat*. This method requires the cactus stack implementation [36], which breaks the interoperability with precompiled libraries and thus is not suitable for a generic threading library.

[14]Qthreads executes a thread on top of the scheduler's stack when `QTHREAD_SPAWN_SIMPLE` is specified. No blocking operation (i.e., `qthread_yield()`) inside threads is allowed.

6 Concluding Remarks

This work extensively explores user-level threading techniques that are suitable for threading libraries from the viewpoint of threading overheads. Our in-depth instruction- and cache-level analysis of twelve methods revealed their performance characteristics and programming constraints. We found that deviation inhibits the run-to-completion execution of thread and highly impacts fork-join overheads. We implemented all the techniques in the same runtime system and evaluated fork-join overheads on Skylake, KNL, POWER8, and ARM64 architectures. Our evaluation with a microbenchmark and three fine-grained applications indicates that the dynamic promotion techniques that defer the context management show the best trade-off between fork-join overheads and programming constraints when the chances of deviation are low.

Our quest is a comprehensive understanding of the design and implementation for lightweight threading libraries. This work solely investigates fork-join performance. Arguably, other factors including schedulers and thread pools are known to highly affect the overall performance. Investigating their design and performance is our future work.

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, in particular, its subproject on Scaling OpenMP with LLVm for Exascale performance and portability (SOLLVE). We gratefully acknowledge the computing resources provided and operated by LCRC and JLSE at Argonne National Laboratory. This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

References

[1] D. Novillo, "OpenMP and automatic parallelization in GCC," in *Proceedings of the GCC Developers' Summit*, June 2006, pp. 23–24.
[2] OpenMP®: Support for the OpenMP Language. https://openmp.llvm.org/.
[3] Intel® OpenMP* Runtime Library. https://www.openmprtl.org/.
[4] 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.

[5] 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.

[6] 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, Mar. 2011.

[7] L. V. Kalé, J. Yelon, and T. Knauff, "Threads for interoperable parallel programming," in *Proceedings of the Ninth International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '96, Aug. 1996, pp. 534–552.

[8] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93, Sept. 1993, pp. 91–108.

[9] 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, Mar. 1998.

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

[11] 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. 512–526, Oct. 2017.

[12] 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.

[13] S. Iwasaki, A. Amer, K. Taura, and P. Balaji, "Lessons learned from analyzing dynamic promotion for user-level threading," in *Proceedings of the 2018 IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18, Nov. 2018, pp. 23:1–23:12.

[14] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper, "Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures," in *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09, Aug. 2009, pp. 91–100.

[15] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *Proceedings of the 2008 IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '08, Nov. 2008, pp. 36:1–36:11.

[16] U. A. Acar, V. Aksenov, A. Charguéraud, and M. Rainey, "Provably and practically efficient granularity control," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19, Feb. 2019, pp. 214–228.

[17] A. Tzannes, G. C. Caragea, U. Vishkin, and R. Barua, "Lazy scheduling: A runtime adaptive scheduler for declarative parallelism," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 3, pp. 10:1–10:51, Sept. 2014.

[18] U. A. Acar, A. Charguéraud, A. Guatto, M. Rainey, and F. Sieczkowski, "Heartbeat scheduling: Provable efficiency for nested parallelism," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '18, June 2018, pp. 769–782.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Boost C++ Libraries. https://www.boost.org/.

[25] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '00, July 2000, pp. 1–12.

[26] 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.

[27] "Intel 64 and IA-32 Architectures Software Developers Manual Volume 2," Sept. 2016.

[28] "ARM Cortex-A Series Version: 1.0 Programmer's Guide for ARMv8-A," Mar. 2015.

[29] "Power ISA™ Version 2.07 B," Jan. 2018.

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

[31] "64-Bit ELF V2 ABI Specification Power Architecture Workgroup Specification Revision 1.4," http://openpowerfoundation.org/wp-content/uploads/resources/leabi/leabi-20170510.pdf, May 2017.

[32] 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.

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

[34] 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.

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

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

[37] 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, June 1997, pp. 320–333.

[38] 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.

[39] 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.

[40] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, ser. IPDPS '09, May 2009, pp. 1–12.

[41] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *International Journal of High Performance Computing Applications*, vol. 26, no. 2, pp. 110–124, May 2012.

[42] A. Huynh, D. Thain, M. Pericàs, and K. Taura, "DAGViz: A DAG visualization tool for analyzing task-parallel program traces," in *Proceedings of the Second Workshop on Visual Performance Analysis*, ser. VPA 15, Nov. 2015.

[43] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 5.0," Nov. 2018.

[44] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, Mar. 2009.

[45] J. Schuchart, K. Tsugane, J. Gracia, and M. Sato, "The impact of taskyield on the design of tasks communicating through MPI," in *Proceedings of the 13th International Workshop on OpenMP*, ser. IWOMP '18, Sept. 2018, pp. 3–17.

[46] P. E. Hadjidoukas and V. V. Dimakopoulos, "Support and efficiency of nested parallelism in OpenMP implementations," *Concurrent and Parallel Computing: Theory, Implementation and Applications*, pp. 185–204, 2008.

[47] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "ForestGOMP: An efficient OpenMP environment for NUMA architectures," *International Journal of Parallel Programming*, vol. 38, no. 5, pp. 418–439, Oct. 2010.
[48] 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.
[49] 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.
[50] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji, "BOLT: Optimizing OpenMP parallel regions with user-level threads," in *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '19, Sept. 2019, pp. 29–42.
[51] KDD Cup 1999 Data. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.
[52] R. Yokota and L. A. Barba, "A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems," *Int. J. High Perform. Comput. Appl.*, vol. 26, no. 4, pp. 337–346, Nov. 2012.
[53] K. Taura, J. Nakashima, R. Yokota, and N. Maruyama, "A task parallelism meets fast multipole methods," in *Proceedings of the Third Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '12, Nov. 2012, pp. 617–625.
[54] J. Ang, B. Barrett, K. Wheeler, and R. Murphy, "Introducing the Graph 500," Cray User Group (CUG), May 2010.
[55] 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.
[56] 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.
[57] T. Fukuoka, W. Endo, and K. Taura, "An efficient inter-node communication system with lightweight-thread scheduling," in *Proceedings of the 21st International Conference on High Performance Computing and Communications*, ser. HPCC '19, Aug. 2019.
[58] MPICH — High-Performance Portable MPI. https://www.mpich.org/.
[59] Q. Chen, M. Guo, and Z. Huang, "CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12, June 2012, p. 163172.
[60] 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.
[61] 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.
[62] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into LLVMs intermediate representation," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 17, Jan. 2017, p. 249265.
[63] "Intel® Cilk™ Plus Application Binary Interface Specification," https://www.cilkplus.org/sites/default/files/open_specifications/CilkPlusABI_1.1.pdf, Dec. 2011.
[64] C. E. Leiserson, "The Cilk++ concurrency platform," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09, July 2009, pp. 522–527.
[65] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, July 2007.
[66] 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.
[67] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa, "Backtracking-based load balancing," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09, Feb. 2009, pp. 55–64.

Shintaro Iwasaki is a Ph.D. candidate at the University of Tokyo. He is also a predoctoral appointee at Argonne National Laboratory. He received his B.S. and M.S. degrees from the University of Tokyo in 2015 and 2017, respectively. His research interests include parallel programming languages, compiler optimizations, parallel runtime systems, and scheduling techniques.

Abdelhalim Amer is an assistant computer scientist in the Mathematics and Computer Science Division at Argonne National Laboratory. His research falls generally under the parallel and distributed computing landscape. More specifically, his focus is on communication runtimes and their interaction with threading models on massively parallel systems.

Kenjiro Taura is a professor at the Department of Information and Communication Engineering, University of Tokyo. He received his B.S., M.S., and D.Sc. degrees from the University of Tokyo in 1992, 1994, and 1997, respectively. His major research interests are centered on parallel/distributed computing and programming languages. His expertise includes efficient dynamic load balancing, parallel and distributed garbage collection, and parallel/distributed workflow systems.

Pavan Balaji holds appointments as a computer scientist and group lead at Argonne National Laboratory, where he leads the Programming Models and Runtime Systems group. His research interests include parallel programming models and runtime systems for communication and I/O on extreme-scale supercomputing systems, modern system architecture, cloud computing systems, data-intensive computing, and big-data sciences.