Lightweight Preemptive User-Level Threads

Shumpei Shiina
The University of Tokyo
Tokyo, Japan
shiina@eidos.ic.i.u-tokyo.ac.jp

Shintaro Iwasaki
Argonne National Laboratory
Lemont, Illinois, USA
siwasaki@anl.gov

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

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

Abstract

Many-to-many mapping models for user- to kernel-level threads (or "M:N threads") have been extensively studied for decades as a lightweight substitute for current Pthreads implementations that provide a simple one-to-one mapping ("1:1 threads"). M:N threads derive performance from their ability to allow users to context switch between threads and control their scheduling entirely in user space with no kernel involvement. This same ability, however, causes M:N threads to lose the kernel-provided ability of *implicit OS preemption*—threads have to explicitly yield control for other threads to be scheduled. Hence, programs over nonpreemptive M:N threads can cause core starvation, loss of prioritization, and, sometimes, deadlock unless programs are written to explicitly yield in proper places. This paper explores two techniques for M:N threads to efficiently achieve implicit preemption similar to 1:1 threads: *signal-yield* and *KLT-switching*. Overheads of these techniques, with our optimizations, can be less than 1% compared with nonpreemptive M:N threads. Our evaluation with three applications demonstrates that our preemption techniques for M:N threads improve core utilization and enhance the performance by utilizing lightweight context switching and flexible scheduling of M:N threads.

CCS Concepts: • **Computing methodologies → Shared memory algorithms**.

Keywords: preemption, multithreading, user-level threads, deadlock, priority

PPoPP '21, February 27–March 3, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441610

1 Introduction

Many-to-many thread mapping models ("M:N threads")—where "M" user-level threads (ULTs) that are visible to the application programmer are mapped to "N" kernel-level threads (KLTs) that are managed by the OS—have been an area of active research for the past several decades. Several implementations of M:N threads have been proposed during this time [13, 14, 18, 32, 39, 49, 55]. This is in contrast to the simple one-to-one mapping model that current Pthreads implementations use ("1:1 threads"). The many-to-many mapping model used by M:N threads allows them to perform context switching and scheduling almost entirely in user space with no kernel involvement. This capability is the source of the high performance that most M:N thread implementations can achieve. In fact, modern M:N thread implementations such as Argobots [29, 30, 49], Qthreads [55], and MassiveThreads [39] can achieve several orders of magnitude lower overhead compared with current implementations of Pthreads, thus allowing for more fine-grained parallelism and schedulers optimized for it [8].

As beneficial as the kernel-bypass feature of M:N threads is, however, it has its own shortcomings. In particular, bypassing the kernel for scheduling and context switching causes M:N threads to lose the kernel-provided ability of *implicit OS preemption*. That is, M:N threads have to explicitly yield control for other threads to be scheduled. Since scheduling of M:N threads happens only at explicit *scheduling points*, programs that do not call scheduling operations in a timely manner cause M:N threads to occupy cores for a long time, resulting in load imbalance or loss of thread prioritization. Programs can even deadlock if nonpreemptive M:N threads fall into a busy loop waiting for the progress of other threads. Manually inserting explicit scheduling points in proper places would be a solution but is often impractically burdensome considering today's complex applications built on top of numerous dependent frameworks and libraries. Closed-source software exacerbates this situation since users cannot rewrite their implementations. Lack of implicit OS preemption narrows the applicability of M:N threads.

For simplicity, in the rest of the paper we will refer to ULTs simply as "threads," as is common practice in the literature. Kernel-level threads will be referred to as KLTs, to clearly distinguish them from ULTs. As we mentioned, *ULTs can be implemented as either M:N threads or 1:1 threads.*[1] In fact, major thread specifications, including Pthreads [1] and OpenMP [41], do not require a one-to-one mapping of threads to KLTs.

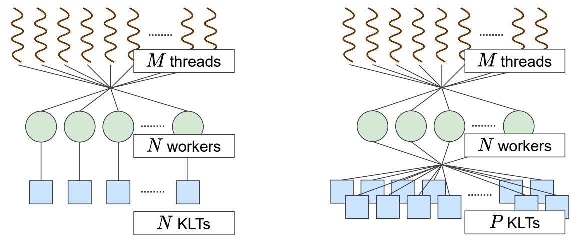
In this paper we investigate two techniques for M:N threads to achieve implicit OS preemption similar to 1:1 threads: *signal-yield* and *KLT-switching*. The first technique—signal-yield—has been previously proposed [4, 9, 38] and also integrated into the Go language [2, 17]. This technique preempts a thread by interrupting the execution with a preset timer signal and context switching from within the signal handler. While this technique is simple, it is not directly applicable to several real-world applications because of two reasons.

1. Timer signals are not scalable and can be expensive on systems with large core counts. In this paper, we present new optimizations to mitigate this overhead by coordinating the signals across different KLTs.
2. Signal-yield requires user functions to be "KLT-independent" and have the ability to execute correctly even if the underlying KLT is switched dynamically during execution. This requirement, unfortunately, is highly restrictive for many applications. For instance, the `malloc()` implementation in Glibc is KLT-dependent because it uses KLT-local data.

Our second technique—KLT-switching—overcomes the KLT-dependence issue in signal-yield by virtualizing the concept of a KLT into what we call a "worker." In other words, "M" threads would be mapped to "N" workers, which in turn would internally be mapped to a pool of "P" KLTs. Like signal-yield, KLT-switching also preempts a thread by interrupting the execution with a signal. Unlike signal-yield, however, instead of context switching the thread from within the signal handler, the entire KLT is suspended, and the worker is remapped to another KLT from the KLT pool. KLT-switching incurs higher overhead than signal-yield does, but it covers a wider range of applications. For consistency, we will use the term "worker" for both signal-yield and KLT-switching. In signal-yield, a worker would be equivalent to a KLT (Figure 1a), while in KLT-switching there is no static mapping between workers and KLTs (Figure 1b).

Both techniques involve kernel calls only for interruption; they require neither kernel modification (e.g., scheduler activations [5]) nor compiler support to insert preemption points [6, 43] because they rely only on OS features provided by today's mainstream OS implementations. Thus,

[1]We note that some literature uses the term "ULT" to denote M:N threads. In order to clearly distinguish KLTs and ULTs, a ULT in this paper denotes a thread visible to user programs. By definition, Pthreads and OpenMP threads are ULTs, and their common implementations adopt 1:1 mapping.



(a) Nonpreemptive/signal-yield **(b)** KLT-switching

Figure 1. Thread mapping to workers and KLTs in M:N threads.

the overhead of our techniques is incurred only at interruption, which is less than 1% when the preemption interval is 1 ms (OS preemption is typically in the millisecond range). We note that our implementation allows combining all three types of M:N threads within the same application: traditional nonpreemptive threads, signal-yield threads, and KLT-switching threads. Users can, therefore, further optimize programs by spawning the most suitable type of threads based on their needs for performance and threading features.

We evaluate three applications to evaluate the benefits of preemptive M:N threads. We first demonstrate the applicability of our preemption techniques with the Cholesky decomposition kernel in SLATE [22], which has nested parallelism and internally uses Intel MKL as a BLAS library. While using nonpreemptive M:N threads, instead of Pthreads, can improve the performance by reducing threading overheads in nested parallelism, they are not reliable and can deadlock in some cases. With our proposed preemptive M:N threads, we can achieve up to 27% performance improvement over Intel OpenMP but without deadlocks. The second evaluation is with HPGMG-FV [3], a high-performance geometric multigrid application, under *thread packing* [15]. Core starvation incurred by thread oversubscription under thread packing is efficiently alleviated by designing a scheduler specialized for thread packing, which outperforms traditional implementations that use 1:1 threads or nonpreemptive M:N threads. Our evaluation of the LAMMPS [44, 47] molecular dynamics application with in situ analysis shows that lightweight preemptive threads can help with thread prioritization.

2 Background

Most traditional M:N thread implementations have not supported preemption. Thus, explicit yielding (also known as "cooperative scheduling") has been considered a fundamental feature of M:N threads. In this section, we first describe the basic idea of M:N threads. We then discuss preemption support for threads in both M:N and 1:1 models.

2.1 Overview of M:N Threads

M:N threads have been proposed as lightweight threads that are implemented mostly with user-space operations. As with

1:1 threads, M:N threads allow each thread to have its own stack. In an M:N thread implementation, when a thread context switches, the running thread's stack pointer and registers are saved in a separate memory location, and the control is handed back to the scheduler. The scheduler then picks another thread to be scheduled (depending on scheduling policy it uses), loads the new thread's stack pointer and registers, and executes it—an approach that costs only about one hundred cycles in total. User-level context switching is key to efficiently implementing several threading operations including fork, join, and yield. Users can develop their own schedulers, thus allowing them to customize their scheduling algorithm for specific workloads. Some parallel runtimes also provide thread-like entities, called run-to-completion threads or "tasks," but these entities do not support generic yield; tasks in Cilk [7, 21] and Intel TBB [46] have no concept of yielding while tasks in LLVM/Intel OpenMP support *stack-based yield*, which has highly restrictive scheduling constraints [48]. This paper does not deal with tasks, but only with threads.

While numerous M:N thread implementations exist, this paper assumes the following threading model, which is common to most such implementations. On initialization, the runtime creates as many workers as the available cores, and each worker has one KLT and one "scheduler thread." The scheduler thread runs an infinite loop that tries to pop a thread from its associated *thread pools* that store ready threads and runs it by context switching. When a thread explicitly yields or finishes its thread function, it switches to the scheduler thread by context switching again.

2.2 Preemption for Threads

1:1 threads adopt both explicit and implicit yielding capabilities. In a 1:1 thread implementation, threads can explicitly call a yield function (such as `sched_yield()`) or use implicit OS preemption if a thread does not yield for a "long time" (an OS scheduler time slice). In either case, control is handed back to the scheduler, which can then pick the next thread to schedule based on priority or fairness metrics. On the other hand, M:N threads are not preemptive. Context switching between threads happens only at explicit scheduling points, namely, when a thread explicitly calls yield or yield-like operations (e.g., fork, join, or barrier) or when it completes. Thus, if threads occupy cores for a long time without issuing explicit scheduling operations, nonpreemptive M:N threads can cause core starvation, loss of prioritization, and deadlock.

Most M:N threads are nonpreemptive since they cannot count on the OS preemption functionality, which utilizes hardware timer interruption, because M:N threads are not visible from the OS. To realize preemption for M:N threads, we have to control interruption to M:N threads from user space, using some kernel interfaces provided by the OS. In the next section, we describe how to interrupt the execution of threads without kernel modification.

3 Designing Preemption for M:N Threads

This section presents the design, implementation, and detailed analysis of two preemption techniques for M:N threads—signal-yield and KLT-switching.

3.1 Preemption Techniques for M:N Threads

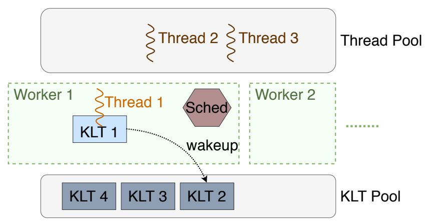
3.1.1 Signal-Yield. The first preemption technique that we study in this paper—signal-yield—was proposed in [2, 4, 9, 38]. The central idea of signal-yield is that a running thread can be interrupted by using OS timer signals, thus emulating OS time slices. Once interrupted, the running thread will context switch to the scheduler from within the signal handler. Since the stack frame of the signal handler is located on top of the running thread's stack frame, the context switch saves contexts for both the signal handler and the running thread. When the thread is resumed, the execution context is still in the signal handler; then the control exits from the signal handler and returns to the thread's context. We note that some systems, including POSIX (by default), do not invoke a signal handler while the previous signal handler associated with the same signal is unresolved. For signal-yield, this would mean that only one thread can be preempted on each worker. To avoid this issue, we unblock the signal for the signal handler right before context switching from within the signal handler. This strategy allows us to preempt as many threads as needed within the same worker.

While signal-yield allows us to achieve OS time-slicing-like preemption capability, it has two shortcomings.

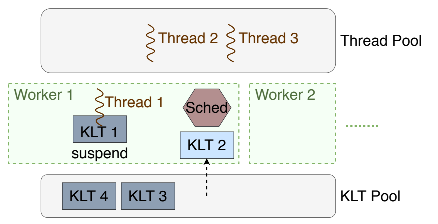
1. Because calling a signal handler involves taking a lock in the kernel, concurrent invocation of signal handlers on multiple cores can cause severe lock contention. Thus, if timer signals are issued at nearly the same time, some threads will spend more time to handle signals. For systems with large core counts this situation can cause significant overhead. In Section 3.2 we will describe optimization techniques to address this issue.

2. The signal-yield technique is relevant only for threads that execute KLT-independent functionality, namely, functions that can execute correctly even if the underlying KLT is switched dynamically during execution. This, essentially, restricts threads from using any KLT-local storage. Unfortunately, using KLT-local storage is common practice in many applications and libraries, making this a significant restriction. For instance, the `malloc()` implementation in Glibc uses KLT-local storage to save flags and cache data. If preemption happens in a KLT-dependent function, the next thread running on the same KLT may modify the KLT-local data by, for example, calling the same function again.

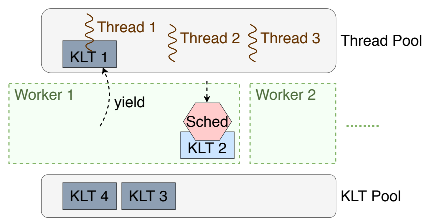
3.1.2 KLT-Switching. Our second preemption technique—KLT-switching—is designed to overcome the KLT-dependence issue in signal-yield. The core idea of this technique is to extend the concept of a worker so there is no static mapping between workers and KLTs. While a thread is not preempted,



(a) When a thread (thread1) is preempted and a signal handler is called on it, it wakes up a KLT (KLT2) in the KLT pool.

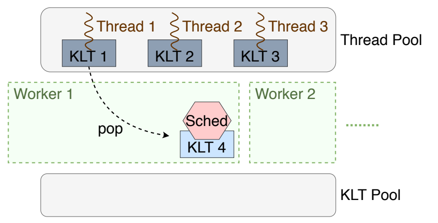


(b) The new KLT (KLT2) wakes up, and the preempted thread (thread1) associates the previous KLT (KLT1) with itself and suspends its execution.

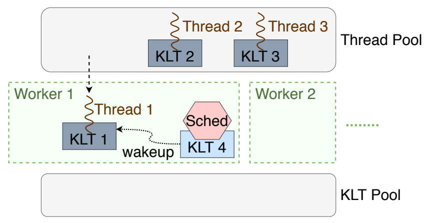


(c) The woken-up KLT (KLT2) runs the scheduler and puts the preempted thread (thread1) into the thread pool as if it had called a yield function.

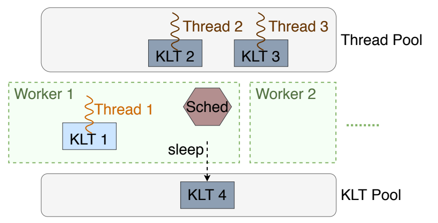
Figure 2. Illustration of KLT-switching when preemption happens.



(a) The scheduler running on a KLT (KLT4) pops a preempted thread (thread1) from the thread pool.



(b) The running KLT (KLT4) wakes up the KLT (KLT1) associated with the preempted thread (thread1).



(c) The preempted thread (thread1) resumes on the same KLT (KLT1). The previous KLT (KLT4) exits from the scheduler and sleeps in the KLT pool.

Figure 3. Illustration of KLT-switching when rescheduling a preempted thread.

it works as a normal M:N thread does. When a thread is preempted, however, the suspended thread temporarily gets 1:1 mapping to the underlying KLT of the worker. The assigned KLT is restricted from executing other threads, so the KLT-local data remains unchanged until the suspended thread resumes. We dynamically allocate additional KLTs during suspension and remap the worker to a different KLT.

The overall working of KLT-switching is illustrated in Figure 2 (showing the suspend path) and Figure 3 (showing the resume path). In this example, worker1 is initially mapped to KLT1 and is executing thread1. When the time slice expires, thread1 receives a signal to suspend itself. At this point, three actions are done: (1) a new KLT (KLT2) is allocated and assigned to worker1; (2) the suspended thread (thread1) and the KLT on which it was executing (KLT1) are both pushed to the thread pool for later execution; and (3) the execution context switches back to the scheduler, which will now execute on the newly allocated KLT2. Similarly, when the thread needs to be resumed, again three steps need to be done: (1) the thread (thread1) and its corresponding KLT (KLT1) are popped back from the thread pool by the scheduler; (2) the execution context switches to thread1; and (3) the KLT on which the scheduler was executing (KLT4) is returned to the KLT pool. We note that even though the KLT pool can have more KLTs than the number of cores, not all of them are active at the same time. In fact, we keep active only as many KLTs as there are cores available.

An interesting detail to note is that KLT-switching needs a special mechanism to create KLTs. To allocate a new KLT upon preemption, the worker first checks the KLT pool to see whether any unused KLTs are available. If no unused KLTs are available, a new KLT will need to be created. Unfortunately, we cannot simply create new KLTs inside a signal handler because most KLT-creation functions (such as `pthread_create()`) are not *async-signal-safe*; in other words, they cannot be called from within signal handlers [1]. To work around this issue, we delegate the creation of KLTs to a dedicated KLT called "KLT creator." When a new KLT needs to be created, we send a request to the KLT creator, which then creates a KLT and adds it to the KLT pool.

Another interesting detail to note is that once a request is sent to the KLT creator, the signal handler cannot wait for the completion of the KLT creation. The reason is that the KLT creation function itself might try to acquire a global lock, which would cause a deadlock if the preempted thread had already taken the lock. Therefore, the running thread must exit from the signal handler and wait for the next signal interruption to check whether a new KLT is available. When the running thread receives its next signal interruption, however, the newly created KLT may have gotten assigned to another worker. In that case, it would have to issue another

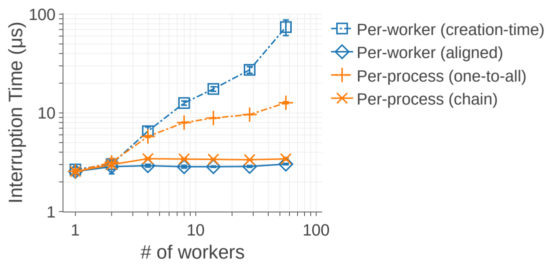


Figure 4. Average time for an OS timer interruption with 1 ms timer interval on Skylake.

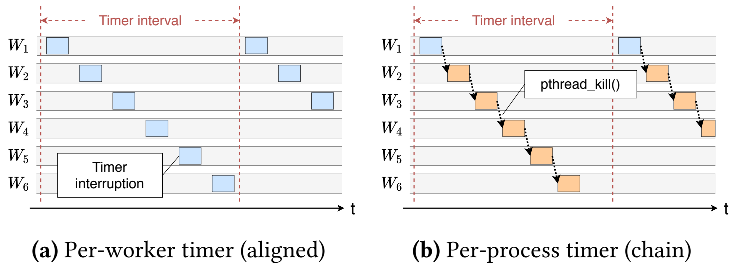


Figure 5. Illustration of optimized preemption timers. A timeline of six workers is shown; the rectangles represent interruptions.

request for a new KLT creation and go through the same cycle again. We note that while this situation may cause delay in some cases, there is no risk of livelock because, in the worst case, we would allocate as many KLTs as threads, thus simply deteriorating to a 1:1 threading model.

3.2 Optimizations for Preemption Timers

Both signal-yield and KLT-switching require a *preemption timer* to periodically send a signal to the running thread in order to force implicit preemption on preemptive threads. This is accomplished by using the `timer_create()` function. We consider two approaches to implement preemption timers: *per-worker* and *per-process*. With a per-worker timer, every worker has its own OS timer. In contrast, with a per-process timer, all workers in the process share one OS timer: one worker receives the signal and forwards it to other workers.

3.2.1 Per-Worker Timer. A naive implementation of a per-worker timer in which the timer of each worker is set independently (e.g., on worker creation) does not scale well on systems with large core counts. Figure 4 shows the average time spent in a timer interrupt when all workers are interrupted every 1 ms. The plot shows an average of 1,000 interrupts with standard deviations. System settings are described in Section 4. The naive implementation (*Per-worker (creation-time)*) shows poor scalability, taking as much as 100 μs for large core counts (in contrast, thread context switching for M:N threads costs tens of nanoseconds). This behavior is because of signal contention. In Linux, calling a signal handler involves taking a lock in the kernel, thus causing lock contention when multiple signals are issued at the same time.

To avoid such signal contention, we propose an approach called "timer alignment," where the timer interrupts across the different workers are explicitly aligned in order not to overlap (see Figure 5a). This ensures that the signal handling cost does not increase with increasing core counts (*Per-worker (aligned)* in Figure 4).

Nevertheless, per-worker timers have two shortcomings.

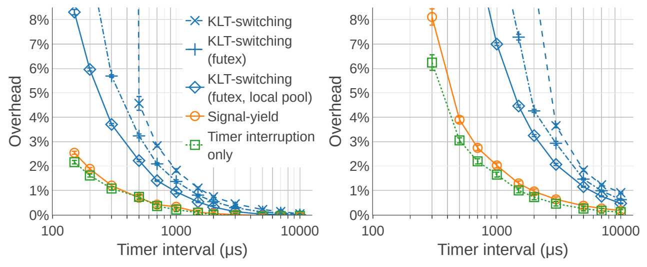
1. They are not portable. The feature of sending periodic timer signals to a specific KLT is not a part of the POSIX specification, although it is implemented in Linux.

2. They do not distinguish between workers that have preemptive threads and workers that do not. Consider an application that has both preemptive and nonpreemptive threads: the per-worker timer would signal all workers, even if none of the currently running threads are preemptive, thus adding overhead with no benefit.

3.2.2 Per-Process Timer. To overcome the shortcomings in per-worker timers, we investigate "per-process timers" as an alternative signaling technique. In this technique, only one worker receives the periodic timer signal from the OS. It then checks whether the running thread on another worker is preemptive, and only if it is preemptive does it send a new signal to that worker by `pthread_kill()`. It repeats this procedure for the remaining workers. If none of the running threads are preemptive, no additional signals are issued.

While per-process timers can significantly reduce signal overhead in cases where there are few preemptive threads, they are not particularly beneficial when most threads are preemptive. In fact, as shown in Figure 4 (*Per-process (one-to-all)*), the average interruption time continues to scale linearly when all threads are preemptive, taking tens of microseconds for large core counts. The reason for this increase is also signal contention because issuing a `pthread_kill()` call is much cheaper than signal handling. Hence, all workers try to process their received signals at nearly the same time, thus causing contention.

To avoid this signal contention, we propose a new optimization to per-process timers, called "chained signals." In this optimization, the worker that received the timer interrupt handles the signal and then issues a signal to at most one other worker, depending on which worker has a preemptive thread running. That worker in turn handles the received signal and again forwards the signal to at most one other worker. Thus, the workers are interrupted one by one like a chain reaction (see Figure 5b). As shown in Figure 4 (*Per-process (chain)*), this optimization can significantly improve the average interruption time. We note that the performance of *Per-process (chain)* is slightly worse than that of *Per-worker*



(a) Skylake **(b)** KNL

Figure 6. Relative overhead of preemptive M:N threads compared with nonpreemptive M:N threads over a compute-intensive benchmark. A per-worker timer is used.

(aligned) because of the additional `pthread_kill()` calls, indicating that per-worker timers are preferable when it is known that most threads are preemptive.

3.3 Optimizations for KLT-Switching

While signal-yield is a low overhead preemption technique, as described in Section 3.1.1, it is relevant only for threads that execute KLT-independent functionality. KLT-switching, on the other hand, is a more generally applicable technique, although it can be more expensive than signal-yield. In this section, we analyze the performance of KLT-switching and present several optimizations to minimize its overhead.

Figure 6 compares the overhead of KLT-switching with that of signal-yield with a compute-intensive microbenchmark in which each of 56 workers runs ten threads that just consume CPU cycles in a loop. In this experiment we use the optimized per-worker timer. Experimental environments of Skylake and KNL are summarized in Table 2 in Section 4. Based on the experimental data, we note that the overhead of signal-yield is virtually identical to that of a pure timer interrupt (*Timer interruption only*). In other words, signal-yield does not add any additional overhead compared with that of the basic timer signal. We also note that a naive implementation of KLT-switching can add significant overhead compared with that of signal-yield.

3.3.1 Suspending and Resuming KLTs. Since we perform KLT-switching in a signal handler, suspending and resuming KLTs require using *async-signal-safe* functionality, which restricts usable functions in practice. A portable implementation of such functionality could use `sigsuspend()` to suspend a KLT at preemption and `pthread_kill()` to resume a KLT because both calls are *async-signal-safe*. While correct and portable, `sigsuspend()` involves additional signal handling, which leads to high overhead.

In this optimization, we replace the `sigsuspend()` and `pthread_kill()` calls with a futex-based suspend/resume implementation. Specifically, once a running thread receives a signal, it simply suspends its underlying KLT on a futex variable. When the KLT needs to be resumed, the resuming thread would simply wake up the suspended KLT with a `FUTEX_WAKE` operation. We note that futex is Linux-specific, but other OSs provide similar functionality. As shown in Figure 6, *KLT-switching (futex)* can reduce the overhead of KLT-switching somewhat, but the overhead is still non-negligible.

Table 1. Overhead of preemption

	1:1 threads (Pthreads)	Signal-yield	KLT-switching
Skylake	2.8 μs	3.5 μs	9.9 μs
KNL	15 μs	18 μs	62 μs

3.3.2 Worker-local KLT Pool. Section 3.1.2 describes a model where KLTs are allocated and cached in a global pool. While this model reduces the average cost of KLT allocation, using a global pool can hurt data locality when the KLTs get resumed on different cores. In addition, in cases where we enable binding of workers to specific cores for performance, whenever a KLT is mapped to a different worker, that KLT's affinity needs to be reset. This can be expensive. To alleviate these overheads, we introduce "worker-local KLT pools," where, together with the global pool, each worker also maintains its own local pool. This optimization further reduces the overhead of KLT-switching, as shown in Figure 6.

Our two optimizations together achieve approximately two times performance improvement, bringing KLT-switching's overheads within a range comparable to that of signal-yield. When the timer interval is short, KLT-switching can still add significant overhead; but once the timer interval is in the millisecond range, its overhead is less than 1% on Skylake. On KNL the overhead of preemption is relatively large because of its less powerful CPU architecture; to make the overheads less than 1%, we need to set about 10 ms as an interval.

We note that even with all the optimizations above, implicit preemption of M:N threads is costlier than that of 1:1 threads. Table 1 shows the median of preemption overheads obtained by repeating preemptions 1,000 times. We set the preemption interval to 10 ms, which is almost the same as the OS preemption interval in our system. The result shows that, on both Skylake and KNL, overheads of signal-yield and KLT-switching are approximately 1.2x and 4x higher than that of 1:1 threads, respectively. The merit of preemptive M:N threads is a combination of implicit preemption, scheduling flexibility, and other lightweight threading operations such as fork-join operations and synchronization primitives. Section 4 shows how preemptive M:N threads can take advantage of the preemption capability.

3.4 Choice of Thread Types

As mentioned earlier, our implementation allows three types of M:N threads to coexist within a single application: traditional nonpreemptive threads, signal-yield threads, and KLT-switching threads. We have a few recommendations on which thread type to choose. First, if a thread function requires no preemption or has explicit yield calls, we recommend using nonpreemptive threads because they are the most efficient. If preemption is needed and users know that the target thread function never calls a KLT-dependent function, signal-yield is recommended because, as shown in Figure 6, the overhead of signal-yield is smaller than that of KLT-switching. If preemption is necessary and the thread function contains a KLT-dependent function, KLT-switching should be chosen. If users do not know what a thread function does (e.g., when replacing Pthreads with M:N threads in an existing large program), we recommend KLT-switching for correct execution.

3.5 Safe Use of Preemptive M:N Threads

Although preemptive M:N threads work similarly to 1:1 threads, they have some programming constraints.

3.5.1 System Calls and Signals. Unlike interruptions generated by a kernel, some system calls fail if a signal handler is triggered while executing them. We can resolve this problem by automatically restarting system calls by setting the `SA_RESTART` flag for the timer signal in `sigaction()`. But some system calls cannot be restarted automatically, and appropriate error handling is required. Moreover, users need to be aware that too short a timer interval would cause many restarts of system calls, which would affect the performance of blocking system calls that take a long time, such as I/O.

3.5.2 Replacement of 1:1 Threads with Preemptive M:N Threads. Implementing a complete substitute for existing 1:1 threads implementations (e.g., widely used Pthreads implementations) with preemptive M:N threads would significantly ease programmer effort in running existing applications over M:N threads. In reality, however, people use threading features beyond the pure specification of standardized threads, so developing a wrapper for a certain thread package is insufficient. Arguably, our preemptive M:N threads lack several nontrivial threading features. For example, on some architectures, programs utilize a register value that is unique to a thread (e.g., fs register on x86/64), so this value must be properly maintained. Without compiler modification, initialization of thread-local storage (TLS) would be problematic since a compiler assumes TLS of KLT and thus often relies on the initialization mechanism on KLT creation. As discussed in Section 3.5.1, our signal-based preemption affects some system calls, so this impact must be evaluated. Investigating a complete replacement of current 1:1 thread implementations is our future work.

3.5.3 Coexistence of Preemptive and Nonpreemptive Threads. Users must pay attention to how they manage their locks when both preemptive and nonpreemptive threads coexist in their application. If a preemptive thread is interrupted while holding a lock and a nonpreemptive thread scheduled later tries to take the same lock, a deadlock can occur. Users should be particularly careful when third-party functions use locks that are out of their control. For example, Glibc `malloc()` can take a global lock. Hence, preemptive threads and nonpreemptive threads should not be mixed unless one is sure that no lock is shared by both types of threads. When in doubt, it is safer to rely on preemptive threads instead of mixing thread types.

Table 2. Experimental environment.

Name	Skylake	KNL
CPU Model	Intel Xeon Platinum 8180M	Intel Xeon Phi 7250
Frequency	2.5 GHz	1.4 GHz
# of Sockets	2	1
# of Cores	56	68
# of HWTs	112 (56 × 2)	272 (68 × 4)
L1 Data Cache	32 KB/core	32 KB/core
L2 Cache	1 MB/core	1 MB/2cores
L3 Cache	38.5 MB/socket	-
Memory	396 GB	204 GB
OS (Kernel)	Red Hat Linux 7.5 (3.10.0-862.14.4.el7)	
Compiler	Intel C/C++ compiler 19.0.4.243 (with the –O3 flag)	

4 Evaluation and Analysis

In this section, we experimentally evaluate and analyze the performance of preemption techniques for M:N threads with three applications. Table 2 shows the experimental environment used in this paper. In the following, we use only Skylake for evaluation. Although our preemption techniques are generic and applicable to other implementation of M:N threads, we chose Argobots (v1.0rc1) [49] for evaluation primarily because it has an OpenMP wrapper called BOLT (v1.0rc3) [31]. For comparison we used Intel MPI for inter-process communication and Intel OpenMP as a 1:1 thread implementation. To maximize the performance of Intel OpenMP, we tweaked its settings as follows. When more threads than cores are created, we set 0 to `KMP_BLOCKTIME` and disabled thread affinity; otherwise, `KMP_BLOCKTIME` and `OMP_PROC_BIND` were set to 200 ms (the default value) and `true`, respectively. We enabled nested hot teams [53] when parallel regions are nested. We plotted the mean and the standard deviation of 10 runs in the following evaluations. In all experiments workers in Argobots were pinned to cores.

4.1 Deadlock Prevention in Cholesky Decomposition

OpenMP [41] is one of the most popular multithreading programming models. Although most production runtimes such as GCC, LLVM, and Intel OpenMP adopt 1:1 threads for OpenMP threads, numerous papers have reported substantial performance benefits from mapping OpenMP threads to lightweight M:N threads [10, 12, 31, 36, 52]. ABI compatibility of such M:N thread-based implementations with major OpenMP runtimes has also been studied [10, 12, 31]. Thus, in theory, such M:N thread-based OpenMP libraries can run existing OpenMP applications without recompilation. In practice, however, a deadlock can occur when programs run on nonpreemptive M:N thread-based OpenMP systems since numerous OpenMP programs assume preemptive threads. Preemptive M:N threads are expected to enhance the performance without deadlocking.

To evaluate the performance, we focus on the Cholesky decomposition kernel extracted from SLATE [22], a modernized linear algebra library. This kernel uses an OpenMP backend with nested parallelism. The outer parallelism uses OpenMP tasks with data dependencies to decompose its computation into operations of submatrices (*tiles*). Computation of each tile calls BLAS routines including DGEMM, TRSM, HERK, and POTRF in an external BLAS library, Intel MKL in our case, in which the inner parallelism exists. OpenMP-parallel Intel MKL, however, assumes implicit preemption during thread synchronization by having threads busy-loop on a memory flag, which causes a deadlock when running on nonpreemptive M:N threads. In our experiment, we fixed the tile size to 1000×1000 and changed the number of tiles. We used BOLT, an Argobots-based OpenMP library that is fully ABI compatible with LLVM OpenMP 9.0 [31]. We modified BOLT to work together with KLT-switching and per-worker timers for preemption. Our thread scheduler is based on BOLT's default work stealing [8] scheduler, in which each worker prioritizes execution of threads in its own FIFO queue and steals a thread from a randomly chosen remote queue when its local queue is empty. Upon preemption, the scheduler pushes the preempted thread into its local FIFO queue and pops the next thread. This preemption mechanism guarantees that all threads are scheduled within a finite time period, preventing a deadlock caused by busy loops.

As a point of comparison, we reverse engineered the closed-source Intel MKL library, by looking through its generated assembly code, to create a version of the code that explicitly yields the thread while waiting for the flag to be set, so it would work with nonpreemptive M:N threads too. Obviously, such reverse engineering is a hack and is only meant to demonstrate the best possible performance that is achievable with nonpreemptive M:N threads. Such modification might not be possible for other applications and does not cover all architectures or all MKL routines, but it is sufficient for our experiment.

Figure 7 shows the results. *IOMP (flat)* shows the performance of Intel OpenMP when inner parallelism is disabled and the outer parallelism is set to the number of cores (56). In all other cases, inner parallelism is enabled, and both inner and outer parallelism are set to 8. In these settings, nested parallel versions, including BOLT and Intel OpenMP, perform better than the flat version because the outer parallelism is not sufficient to make all cores active. In almost all cases, BOLT with preemptive M:N threads outperforms Intel OpenMP thanks to the lightweight threading operations

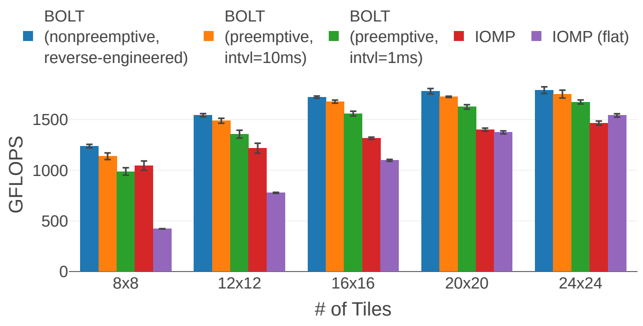


Figure 7. Performance of Cholesky decomposition.

in M:N threads [31]. In this application, larger timer intervals achieve better performance because short preemption intervals incur non-negligible cache misses.

4.2 Thread Packing with HPGMG-FV

OS schedulers designed for general purposes, such as completely fair scheduler (CFS) in Linux, do not perform well for some workloads. We found performance degradation caused by OS schedulers in *thread packing* [15, 16, 42], where threads are dynamically packed into fewer cores for power capping or resource partitioning. With 1:1 threads, CPU affinity masks of threads are dynamically changed so that they are executed on a limited set of cores. This causes significant core starvation because of poor load balancing of OS schedulers for parallel workloads [25, 28, 35]. Nevertheless, users often do not have permission to change the scheduling algorithm of OS schedulers on large-scale computing platforms. Preemptive M:N threads are promising since their user-level scheduling and preemption capability give chances to schedule other threads for better load balancing at the preemption interval.

In this evaluation, we demonstrate that preemptive M:N threads with a user-level scheduler can minimize the performance degradation caused by thread packing. To dynamically change the number of active workers, our preemptive M:N thread-based runtime dynamically wakes up workers by signals or suspends workers upon preemption. Threads that were executed by suspended workers will be executed by the remaining active workers. Let us assume typical HPC workloads where a fixed number of threads as many as cores are created and the computation load is equally balanced among the threads. Under thread packing, the key to good load balancing is how to schedule extra threads that suspended workers have in addition to local threads originally assigned to currently active workers. The idea of our proposed scheduler is to equally balance the load of extra threads across all the active workers by prioritizing extra threads. Preemption allows our runtime to schedule extra threads in a round-robin fashion among active workers, slicing extra threads by a preemption interval.

Algorithm 1 Scheduler specialized for thread packing

```
1:  $N_{total} \leftarrow$ # of workers that are initially created
2:  $pools \leftarrow$ List of $N_{total}$ thread pools
3:  $rank \leftarrow$ Unique ID of this worker within the range $[0, N_{total})$
4:  while true do
5:      $N_{active} \leftarrow$ # of current active workers
6:      $N_{private} \leftarrow N_{active} \times \lfloor N_{total}/N_{active} \rfloor$
7:      for $i \leftarrow rank$ to $N_{private}$ step $N_{active}$ do
8:          if $thread \leftarrow pop(pools[i])$ then
9:              run($thread$)
10:             break
11:     for $i \leftarrow N_{private} + 1$ to $N_{total}$ step 1 do
12:         if $thread \leftarrow pop(pools[i])$ then
13:             run($thread$)
14:             break
```

Algorithm 1 shows the pseudocode of the scheduler. Each worker has a unique integer ID, called *rank*, within the range $[0, N_{total})$, where N_{total} denotes the initial number of workers. Upon thread packing, workers with larger ranks are suspended. The scheduling algorithm consists of two phases: scheduling of threads in *private* pools (lines 7–10) and *shared* pools (lines 11–14). The private pools are exclusively assigned to a worker while the shared pools are shared by all the active workers. Initially only the local pool of each worker is private to the worker while no pools are shared. Under thread packing, the shared pools include local pools of suspended workers. Each worker repeats executing a thread in one of its private pools (lines 7–10) and then a thread in one of its shared pools (lines 11–14) alternately at the preemption interval. Since the preemption interval is the same among all workers, threads in the shared pools are scheduled in a round-robin fashion among all active workers, as active workers peek the shared pools in turn. This strategy, however, hurts data locality when fewer workers are active because most threads are in the shared pools and scheduled by different workers. To avoid this, when the number of active workers (N_{active}) becomes half of N_{total} or less, we exclusively assign more than one pool ($N_{private}$ pools in line 6) to the private pools of every active worker. Thus the number of shared pools are always less than the number of workers, which makes shared pools prioritized against private pools. This achieves good load balancing for threads in the shared pools while keeping good data locality for threads in the private pools.

For evaluation, we chose the finite volume version of High Performance Geometric Multigrid (HPGMG-FV) [3] (version 0.4), which solves linear equations using a full multigrid method. We set eight as the log base 2 of the dimension of each box on the finest grid and allocate eight boxes per process. We created two MPI processes in a single node and bound each process to a NUMA node for better locality. As an M:N thread-based OpenMP runtime we used BOLT [31],

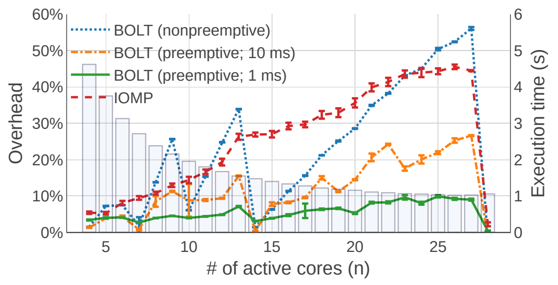


Figure 8. Relative overhead of threading packing in HPGMG-FV. The number of active cores per process is reduced from 28 to n while creating 28 threads. The baseline is BOLT (nonpreemptive) when spawning n threads from the beginning. The baseline's execution times to solve a linear equation are shown as the bar plots.

but we replaced the scheduler with the one we propose (Algorithm 1). We used KLT-switching and per-worker timers for preemption. For comparison to the OS scheduler, we also evaluated the performance of Intel OpenMP (IOMP).

Figure 8 shows performance evaluation of HPGMG-FV when we first create 28 threads per process and reduce the number of active cores from 28 to n (x-axis) in each process. It shows relative overheads compared to a baseline where n threads are created from the beginning, together with the absolute execution time of the baseline to solve a linear equation. We arbitrarily chose the performance of BOLT over nonpreemptive M:N threads as the baseline since Intel OpenMP and BOLT showed almost the same performance. For *IOMP*, we use `taskset` command to bind 28 threads to n cores as done in [16], while for *BOLT* we suspend $28 - n$ workers. The results show that the performance of *IOMP* is far from the ideal performance especially when the number of active cores is close to 28. This is because of inefficient load balancing of CFS in Linux [25, 28, 35]. On the other hand, the performance of preemptive BOLT is close to the ideal one. It is interesting to note that 1 ms preemption interval performs better than 10 ms; this result indicates that 10 ms interval is insufficient to balance the load of extra threads in this case. *BOLT (nonpreemptive)* shows good performance when the number of cores is a divisor of 28, but in other cases the performance is poor because, without preemption, there are not enough scheduling chances for load balancing. This evaluation shows the benefits of user-level scheduling with preemptive M:N threads and configurable preemption intervals.

4.3 In Situ Analysis with LAMMPS

Preemption gives more control over thread scheduling, enabling non-voluntary priority-based scheduling, for example. With preemption, low-priority threads can be immediately evicted in favor of high-priority threads even if low-priority

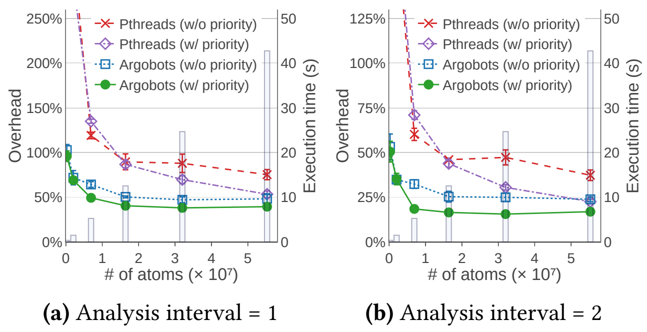


Figure 9. Relative overhead of in situ analysis with LAMMPS compared with simulation-only execution when the problem size is changed on four Skylake nodes. The execution times of simulation-only execution for 100 steps are shown as the bar plots.

threads do not voluntarily yield. As a case study for priority-based scheduling, we picked *in situ analysis in LAMMPS*, where thread prioritization is favored.

LAMMPS [44, 47] is a molecular dynamics simulator developed by Sandia National Laboratories. For our experiments, we used version `stable_7Aug2019` together with the in situ analysis discussed in [27]. We used the Kokkos [11, 20] package for shared-memory parallelism in the simulation code and implemented an Argobots backend in Kokkos which spawns as many simulation threads as the number of workers in every parallel region. The analysis code copies all atoms to a separate buffer and performs analysis on this buffer in parallel, while the simulation is going on, by spawning dedicated analysis threads. In the simulation, we calculate the 3D Lennard-Jones potential for 100 time steps.

Simulation threads have higher priority than analysis threads have because analysis threads are created based on the progress of simulation threads. Too eager scheduling of analysis threads will delay the execution of simulation threads, which can degrade the overall performance. Thus, analysis threads should be executed only when no simulation threads exist (e.g., during MPI communication). The Pthreads version of the code achieves such prioritization by setting a higher "niceness" (lower priority) to the analysis threads than the simulation threads. The Argobots version of the code achieves prioritization by having the scheduler first check whether any simulation threads exist before checking for analysis threads. Only analysis threads are set to be preemptive, and every worker has a LIFO queue for analysis threads in order not to hurt data locality during preemption. We use signal-yield for preemption because the analysis functions are KLT-independent. To minimize the overhead of timer interruption for nonpreemptive simulation threads, we use the per-process timer. The preemption timer interval is fixed to 1 ms. We use four nodes, each of which has one MPI process that consists of 56 workers.

Figure 9 shows the performance comparison of Pthreads and Argobots (with and without prioritization) compared with a baseline of a simulation-only (no analysis) execution. Figure 9a shows the overhead when analysis is executed in every iteration and Figure 9b shows the overhead when it is executed every two iterations. In our experiments, we create one less analysis thread than the available cores because the main thread rarely becomes idle—it is either computing or communicating; the remaining threads, on the other hand, become idle when the simulation is going through sequential portions of the code. For the Pthreads version of the code, simulation threads are created by Intel OpenMP and analysis threads are spawned via the Pthreads interface.

The result shows that Argobots-based execution achieves better performance than Pthreads-based execution because of its lower-overhead threading ability. Especially when the total number of atoms is small, the improvement in performance is significant. Prioritization helps both Pthreads- and Argobots-based executions to reduce core idleness and improve performance when the number of atoms is large. For example, in Figure 9b, when the number of atoms is 5.6×10^7, prioritization reduced 30% of the core idle time in the case of Pthreads and 18% of the core idle time in the case of Argobots. The impact of prioritization is more pronounced when analysis is performed once every two iterations (Figure 9b) compared with when analysis is performed in every iteration (Figure 9a). This is because, when the analysis interval is 2, the amount of analysis work is small enough to complete while no simulation threads exist (e.g., during MPI communication). We note that, because nice values do not impose strict scheduling order, the execution of the Pthreads-based simulation is still uncoordinated. We could have further improved performance by setting strict priority, for example, by using real-time scheduling policies defined in POSIX, such as SCHED_FIFO. However, these policies require root privilege which is often unavailable to users. As a result, Argobots with prioritization using preemptive threads performs the best amongst all of the different approaches—this is the version that is enabled by the work in this paper.

5 Related Work

Research on preemption support for M:N threads is not new. In the early stages of threading support in operating systems, many researchers investigated the interface design between the kernel and user space to implement M:N threads. During this time, researchers investigated using mechanisms to notify kernel events such as blocking and preemption to M:N threads via a special KLT—they called this model *scheduler activation* [5]. Another approach was proposed by Marsh et al. as *first-class user-level threads* [37], in which the kernel notifies events to the user space by using signals. *Nanothreads* passes preemption events from the kernel to the user space via shared memory between them [45]. Solaris OS supported M:N threads; preemption was enabled by using signals [50]. All of these studies required the kernel to be aware of M:N threads. Unfortunately, modern mainstream operating systems have abandoned such capabilities. For example, Solaris OS moved to 1:1 threads from version 9 because of the simple implementation of 1:1 threads [51]. As a result, most modern M:N thread-based parallel systems lack preemption [13, 14, 18, 32, 39, 49, 55]. Our preemption techniques for M:N threads work on today's mainstream operating systems that are unaware of M:N threads.

The *signal-yield* technique that was presented in this paper has been previously investigated [4, 9, 38] and integrated into the Go language [2, 17]. In the context of real-time systems, Anantaraman et al. [4] studied signal-yield on single-core systems, whereas Mollison and Anderson [38] extended it to multicore processors. Mollison and Anderson also noted in their paper that unsafe functions should be migrated to "proxy threads" for safety. This recommendation is impractical since it requires modifying all KLT-dependent functions. Boucher et al. [9] also pointed out this problem and proposed a workaround for safely preempting the execution of external libraries. Their method isolates external shared libraries by loading multiple versions of shared libraries into different linker namespaces and dynamically switching them by modifying the global offset table (GOT). Their workaround, however, imposes additional overheads to every call to external functions. Moreover, their method cannot deal with the KLT-dependence issue when parallelism exists in external libraries (e.g., Intel MKL). Our KLT-switching is the first practical technique that tackles the KLT-dependence issue. We also note that none of the past studies mentioned timer optimizations, which is one of our contributions.

Techniques for virtualizing workers have also been studied, for example, in Concurrent Cilk [56]. These techniques, however, are not designed for implicit preemption but for explicit yield operations on run-to-completion threads or "tasks." This approach simplifies the design somewhat, because yielding no longer needs to be in the context of a signal handler; but it also makes such previous work not applicable to implicit preemption. *User-level processes* proposed by Hori et al. [26] have a similar concept to KLT-switching to deal with the *system-call consistency* issue (i.e., a user-level process has to call system calls from the same KLT throughout its execution), which is especially important for processes. Their approach includes dynamically switching KLTs when system calls are trapped so that system calls are always called from the same KLT. Although this approach is similar to our KLT-switching technique, preemption was not concerned in their work.

Some researchers have proposed compiler-based techniques for preemptive M:N threads. The general idea of these approaches is that the compiler would insert explicit thread scheduling points. This approach has been extensively studied in some Java virtual machine implementations [40, 54]. Unlike timer-based preemption, however, compiler-inserted preemption points cannot be placed at arbitrary points in the code and would depend on the application code structure. Some studies [6, 43] have investigated the optimal frequency of preemption points; frequent preemption points can degrade performance but incur less latency. To achieve more precise timing of preemption at compilation time, Ghosh et al. [23] proposed static estimation for execution time of code blocks by calculating clock cycle latencies of instructions using LLVM [33]. Some work aiming at functional languages, such as Scheme, treats function entrances as scheduling points [19, 24]. It works effectively because function languages usually rely on recursion to perform iterative operations and do not have tight loops in a single function. The Glasgow Haskell Compiler [34] uses an OS timer for preemption, but rather than directly interrupting execution by signals as our approach does, it performs preemption at the explicit scheduling point right after timer expiration for safety. All of these techniques rely on compiler support, and most of them are specific to languages. Our techniques, on the other hand, can be implemented in a threading library and can therefore be easily deployed on most systems.

6 Concluding Remarks

We have investigated two lightweight preemption techniques for M:N threads: *signal-yield*, a technique that is simple but cannot run KLT-dependent functions, and *KLT-switching*, a novel technique that addresses the issue of KLT-dependence. These two techniques have different trade-offs: KLT-switching covers a wider range of programs but has higher overheads than does signal-yield. Our analysis shows that preemption of M:N threads can be achieved with nearly no overhead compared with nonpreemptive M:N threads and with high scheduling flexibility. The results of our evaluation with real-world applications are promising. Preemption can successfully improve the performance of nested OpenMP parallel programs without incurring a deadlock, resolve load imbalance under thread packing, and increase core utilization by efficiently handling thread priority.

Acknowledgments

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] 2018. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (Jan. 2018).

[2] 2020. *Go 1.14 Release Notes.* https://golang.org/doc/go1.14

[3] Mark Adams, Jed Brown, John Shalf, Brian Van Straalen, Erich Strohmaier, and Samuel Williams. 2014. *HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems.* Technical Report LBNL 6630E.

[4] Aravindh V. Anantaraman, Ali El haj Mahmoud, Ravi K. Venkatesan, Yifan Zhu, and Frank Mueller. 2004. EDF-DVS Scheduling on the IBM Embedded PowerPC 405LP. In *Proceedings of the IBM T.J. Watson P=ac^2 Conference* (Yorktown Heights, New York, USA). 63–72.

[5] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1992. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems* 10, 1 (Feb. 1992), 53–79.

[6] Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio Buttazzo. 2011. Optimal Selection of Preemption Points to Minimize Preemption Overhead. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems* (Porto, Portugal) *(ECRTS '11)*. 217–227.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, California, USA) *(PPoPP '95)*. 207–216.

[8] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.

[9] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2020. Lightweight Preemptible Functions. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. 465–477.

[10] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. 2010. ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures. *International Journal of Parallel Programming* 38, 5 (Oct. 2010), 418–439.

[11] H. Carter Edwards and Christian R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *Proceeding of the 2013 Extreme Scaling Workshop* (Boulder, Colorado, USA) *(XSW '13)*. 18–24.

[12] Adrián Castelló, Sangmin Seo, Rafael Mayo, Pavan Balaji, Enrique S. Quintana-Ortí, and Antonio J. Peña. 2017. GLTO: On the Adequacy of Lightweight Thread Approaches for OpenMP Implementations. In *Proceedings of the 46th International Conference on Parallel Processing* (Bristol, UK) *(ICPP '17)*. 60–69.

[13] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications* 21, 3 (Aug. 2007), 291–312.

[14] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, California, USA) *(OOPSLA '05)*. 519–538.

[15] Ryan Cochran, Can Hankendi, Ayse K Coskun, and Sherief Reda. 2011. Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO '11)*. 175–185.

[16] Marco Danelutto, Tiziano De Matteis, Daniele De Sensi, and Massimo Torquati. 2017. Evaluating Concurrency Throttling and Thread Packing on SMT Multicores. In *Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (St. Petersburg, Russia) *(PDP '17)*. 219–223.

[17] Alan A. A. Donovan and Brian W. Kernighan. 2015. *The Go programming language.* Addison-Wesley Professional.

[18] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters* 21, 02 (Mar. 2011), 173–193.

[19] R. Kent Dybvig and Robert Hieb. 1989. Engines from Continuations. *Computer Languages* 14, 2 (Sept. 1989), 109–123.

[20] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns. *J. Parallel and Distrib. Comput.* 74, 12 (Dec. 2014), 3202–3216.

[21] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. 212–223.

[22] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado, USA) *(SC '19)*. Article 26, 18 pages.

[23] Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. 2020. Compiler-Based Timing for Extremely Fine-Grain Preemptive Parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia, USA) *(SC '20)*. Article 53, 15 pages.

[24] Christopher T. Haynes and Daniel P. Friedman. 1987. Abstracting Timed Preemption with Engines. *Computer Languages* 12, 2 (Aug. 1987), 109–121.

[25] Steven Hofmeyr, Costin Iancu, and Filip Blagojević. 2010. Load Balancing on Speed. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Bangalore, India) *(PPoPP '10)*. 147–158.

[26] Atsushi Hori, Balazs Gerofi, and Yutaka Ishikawa. 2020. An Implementation of User-Level Processes using Address Space Sharing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (New Orleans, LA, USA). 976–984.

[27] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. 2018. Process-in-process: Techniques for Practical Address-space Sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona, USA) *(HPDC '18)*. 131–143.

[28] Costin Iancu, Steven Hofmeyr, Filip Blagojević, and Yili Zheng. 2010. Oversubscription on Multicore Processors. In *Proceedings of the 24th IEEE International Symposium on Parallel Distributed Processing* (Atlanta, Georgia, USA) *(IPDPS '10)*. 958–968.

[29] Shintaro Iwasaki, Abdelhalim Amer, Kenjiro Taura, and Pavan Balaji. 2018. 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* (Dallas, Texas, USA) *(SC '18)*. Article 23, 12 pages.

[30] Shintaro Iwasaki, Abdelhalim Amer, Kenjiro Taura, and Pavan Balaji. 2020. Analyzing the Performance Trade-Off in Implementing User-Level Threads. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (Feb. 2020), 1859–1877.

[31] Shintaro Iwasaki, Abdelhalim Amer, Kenjiro Taura, Sangmin Seo, and Pavan Balaji. 2019. BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques* (Seattle, Washington, USA) *(PACT '19)*. 29–42.

[32] Laxmikant V. Kale and Sanjeev Krishnan. 1993. 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 (Washington, D.C., USA) *(OOPSLA '93)*. 91–108.

[33] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (San Jose, CA, USA) *(CGO '04)*.

[34] Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach. 2007. Lightweight Concurrency Primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (Freiburg, Germany) *(Haskell '07)*. 107–118.

[35] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, UK) *(EuroSys '16)*. Article 1, 16 pages.

[36] Gonzàlez Marc, Ayguadé Eduard, Martorell Xavier, Labarta Jesús, Navarro Nacho, and Oliver José. 2000. NanosCompiler: Supporting Flexible Multilevel Parallelism Exploitation in OpenMP. *Concurrency: Practice and Experience* 12, 12 (Oct. 2000), 1205–1218.

[37] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. 1991. First-Class User-level Threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '91)*. 110–121.

[38] Malcolm S. Mollison and James H. Anderson. 2013. Bringing Theory into Practice: A Userspace Library for Multicore Real-Time Scheduling. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (Philadelphia, Pennsylvania, USA) *(RTAS '13)*. 283–292.

[39] Jun Nakashima and Kenjiro Taura. 2014. MassiveThreads: A Thread Library for High Productivity Languages. *Lecture Notes in Computer Science – Concurrent Objects and Beyond* 8665 (Jan. 2014), 222–238.

[40] Anders Nilsson, Torbjörn Ekman, and Klas Nilsson. 2002. Real Java for Real Time - Gain and Pain. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (Grenoble, France) *(CASES '02)*. 304–311.

[41] OpenMP Architecture Review Board. 2018. OpenMP Application Program Interface Version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[42] Jinsu Park, Seongbeom Park, Myeonggyun Han, and Woongki Baek. 2019. POSTER: The Performance Impact of Thread Packing on Synchronization-Intensive Applications. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques* (Seattle, Washington, USA) *(PACT '19)*. 461–462.

[43] Bo Peng, Nathan Fisher, and Marko Bertogna. 2014. Explicit Preemption Placement for Real-Time Conditional Code. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems* (Madrid, Spain) *(ECRTS '14)*. 177–188.

[44] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comput. Phys.* 117, 1 (March 1995), 1–19.

[45] Eleftherios D. Polychronopoulos, Xavier Martorell, Dimitrios S. Nikolopoulos, Jesus Labarta, Theodore S. Papatheodorou, and Nacho Navarro. 1998. Kernel-Level Scheduling for the Nano-Threads Programming Model. In *Proceedings of the 12th International Conference on Supercomputing* (Melbourne, Australia) *(ICS '98)*. 337–344.

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

[47] Sandia National Laboratories. [n.d.]. *LAMMPS Molecular Dynamics Simulator.* http://lammps.sandia.gov

[48] Joseph Schuchart, Keisuke Tsugane, José Gracia, and Mitsuhisa Sato. 2018. The Impact of Taskyield on the Design of Tasks Communicating Through MPI. In *Proceedings of the 14th International Workshop on OpenMP 2018* (Barcelona, Spain) *(IWOMP '18)*. 3–17.

[49] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. 2017. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* PP, 99 (Oct. 2017), 512–526.

[50] Dan Stein and Devang Shah. 1992. Implementing Lightweight Threads. In *Proceedings of the USENIX Summer 1992 Technical Conference* (San Antonio, Texas, USA) *(USENIX '92)*. 1–9.

[51] Sun Microsystems, Inc. 2002. *Multithreading in the Solaris Operating Environment.* https://web.archive.org/web/20090226174929/http://www.sun.com/software/whitepapers/solaris9/multithread.pdf

[52] Yoshizumi Tanaka, Kenjiro Taura, Mitsuhisa Sato, and Akinori Yonezawa. 2000. Performance Evaluation of OpenMP Applications with Nested Parallelism. In *Proceedings of the Fifth International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers* (Rochester, New York, USA) *(LCR '00)*. 100–112.

[53] Xinmin Tian, Jay P. Hoeflinger, Grant Haab, Yen-Kuang Chen, Milind Girkar, and Sanjiv Shah. 2005. A Compiler for Exploiting Nested Parallelism in OpenMP Programs. *Parallel Comput.* 31, 10-12 (Oct. 2005), 960–983.

[54] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. 2004. Preemption-Based Avoidance of Priority Inversion for Java. In *Proceedings of the 33rd International Conference on Parallel Processing* (Montreal, Quebec, Canada) *(ICPP '04, Vol. 1)*. 529–538.

[55] Kyle B. Wheeler, Richard C. Murphyand, and Douglas Thain. 2008. Qthreads: An API for Programming with Millions of Lightweight Threads. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium* (Miami, Florida, USA) *(IPDPS '08)*. 1–8.

[56] Christopher S. Zakian, Timothy A. Zakian, Abhishek Kulkarni, Buddhika Chamith, and Ryan R. Newton. 2016. 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* (Raleigh, North Carolina, USA) *(LCPC '15)*. 73–90.

A Artifact Appendix

A.1 Abstract

The artifact contains the source code of the runtime systems based on preemptive M:N threads (Argobots and BOLT) and benchmarks for evaluation (several microbenchmarks, Cholesky decomposition, HPGMG, and LAMMPS). This appendix section explains how to run the artifact, how to interpret the experimental results, and how to customize the experiments.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Preemption techniques for M:N threads.
- **Program:** The parallel runtime systems used in our experiments were implemented based on Argobots (v1.0rc1) and BOLT (v1.0rc3). Application programs are Cholesky decomposition kernel picked from SLATE (https://bitbucket.org/icl/slate/src), HPGMG (version 0.4), and LAMMPS (stable_7Aug2019). All programs are included in the artifact archive.
- **Compilation:** Intel C/C++ compilers. We used Intel C/C++ compiler 19.0.4.243 (with the −O3 flag) for our experiments.
- **Run-time environment:** Linux should be used for evaluation. We used Red Hat Linux 7.5 (3.10.0-862.14.4.el7) in our experiments. No root access is required.
- **Hardware:** Intel CPUs should be used. We used Intel Xeon Platinum 8180M and Intel Xeon Phi 7250 for our experiments. Multiple nodes are preferable for the evaluation of LAMMPS.
- **Output:** Graph plots will be output as well as raw results such as execution times.
- **Experiments:** See below.
- **How much time is needed to complete experiments (approximately)?:** 2 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** The 2-Clause BSD License. Note that only the pieces of the code of LAMMPS modified for this paper are copyrighted under GNU General Public License version 2.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.4420552

A.3 Description

A.3.1 How to access.
The GitHub repository for the artifact is https://github.com/s417-lama/ppopp21-preemption-artifact. The Zenodo archive with a DOI can be downloaded from https://doi.org/10.5281/zenodo.4420552.

A.3.2 Hardware dependencies.
Intel CPUs should be used for the evaluation of Cholesky decomposition because it has to use Intel MKL. Other experiments should work with different CPU architectures other than Intel CPUs with a little modification, although we did not strictly check if they run properly. We used Intel Xeon Platinum 8180M (two sockets) and Intel Xeon Phi 7250 for our experiments. Multiple nodes are preferable for the evaluation of LAMMPS; we used four nodes for our experiments.

A.3.3 Software dependencies.
Linux should be used as an OS because the artifact uses Linux-specific functionalities. The system should install Intel C/C++ compiler, Intel MKL, Intel OpenMP, and Intel MPI. Although some scripts assume Intel compilers, they should be able to be compiled with other compilers with a little modification of scripts (e.g., check `-use-gcc` option for each script). Singularity or Docker has to be installed to plot graphs.

A.4 Installation

Download the archive from https://doi.org/10.5281/zenodo.4420552 or `git clone` from https://github.com/s417-lama/ppopp21-preemption-artifact.

A.5 Experiment workflow

To configure machine configuration such as the number of cores and the number of sockets in a node, `envs.bash` should be modified. Running `./measure_XXX.bash` (`XXX` is the name of the benchmark to run; see the next section) will compile the runtime systems such as Argobots and BOLT under some specific configuration for the benchmark, compile benchmark programs, store raw results by running benchmarks, and finally generate a plot summarizing the raw results.

The raw results are saved in the `<subdir>/results` directory, where `<subdir>` is the sub directory of each benchmark (e.g. `chol`). The raw results are summarized by plotting scripts that run on a Docker container (Docker image: `s417lama/plotly_ex`), which utilizes `Plotly.js` library (https://plotly.com/javascript/) for visualization. Only this process requires Singularity or Docker be installed.

See `README.md` in the downloaded archive for further information.

A.6 Evaluation and expected results

The correspondence of the measurement scripts (`measure_XXX.bash`) and the experimental results in this paper is summarized as follows.

- `measure_interrupt.bash` outputs a plot (`interrupt_plot.html`) corresponding to Figure 4.
- `measure_overhead.bash` outputs a plot (`overhead_plot.html`) corresponding to Figure 6.
- `measure_overhead_direct.bash` outputs as text context switching times corresponding to Table 1.
- `measure_chol.bash` outputs a plot (`chol_plot.html`) corresponding to Figure 7.
- `measure_hpgmg.bash` outputs a plot (`hpgmg_plot.html`) corresponding to Figure 8.
- `measure_lammps.bash` outputs a plot (`lammps_plot.html`) corresponding to Figure 9.

HTML files of the plots can be opened with a web browser. If the range of axes is not properly set, it is possible to zoom in/out on the graph interactively.

The raw results used in this paper are stored in the `raw_results` directory; the results obtained in other systems (in the `<subdir>/results` directory) can be compared against them.

We note that we set a small number of repeats and a small subset of the data points for several benchmarking scripts to reduce the running time. To reproduce the full results in this paper, please follow the instructions for customizing the benchmarks in `README.md` in the archive.

A.7 Experiment customization

When running `measure_interrupt.bash`, the shell variable `THREADS` in `preemption_benchmarks/measure_jobs/timer_measure.bash` should be modified to set proper numbers of threads to use.

If `numactl` option is not installed in the system, please remove `numactl -interleave=all` and `numactl -iall` in `chol/run.bash` and `lammps/measure.bash`. This may affect the performance results in multi-socket machines; in that case it is recommended to install `numactl` in the system.

For further customization of the experiments, including the number of repeats and the problem sizes, see `README.md` in the downloaded archive.

A.8 Notes

If benchmark programs unexpectedly finish without any output or error, the error message may not be shown in the console. Because all outputs of benchmarks (including standard errors) are stored in the `<subdir>/results` directory. Please check `*.err` files under the result directory to see errors output to stderr.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html