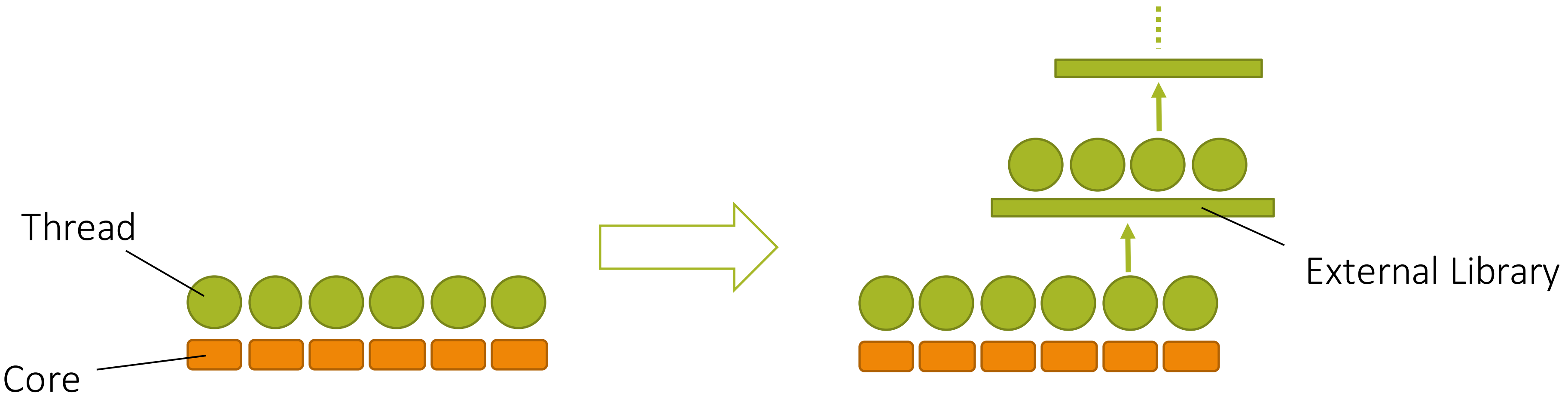
Lightweight Preemptive User-Level Threads

@ PPoPP '21

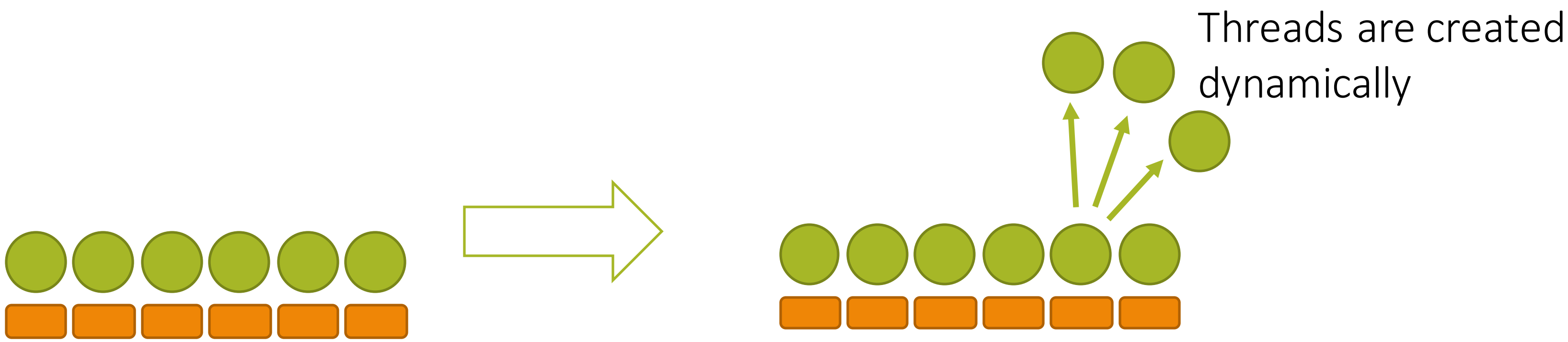
Shumpei Shiina (The University of Tokyo), Shintaro Iwasaki (Argonne National Laboratory), Kenjiro Taura (The University of Tokyo), Pavan Balaji (Argonne National Laboratory)

More Threads than the Number of Cores

Nested Parallelism



Asynchronous Tasks



Thread schedulers should be able to handle many, fine-grained threads

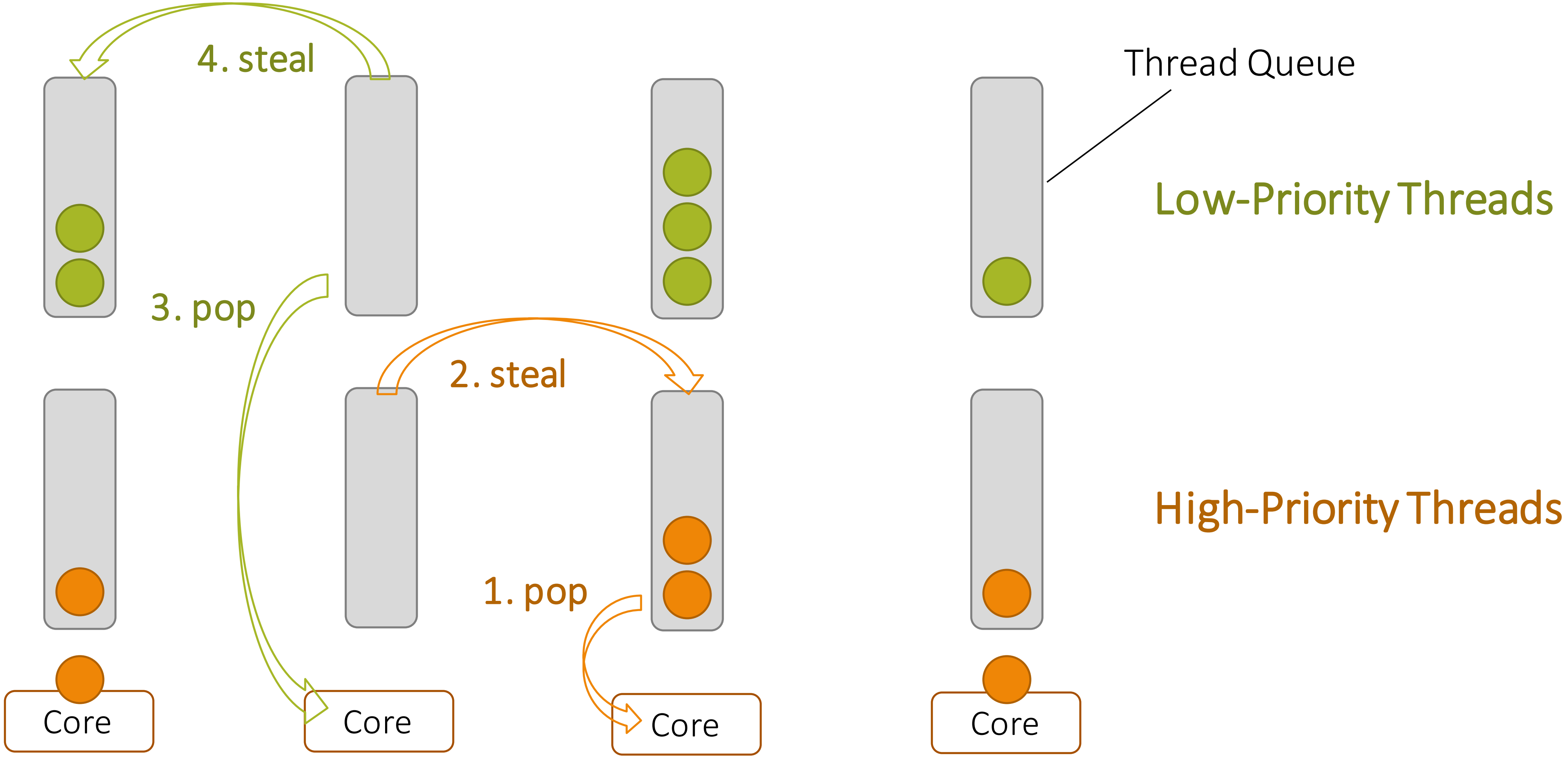
- Complicated software stacks where many parallel libraries are hierarchically composed
- Coexisting various kinds of tasks (e.g., in situ analysis with simulation tasks and analysis tasks)

--> Lightweight threads are needed

Customizable Thread Schedulers for High Performance

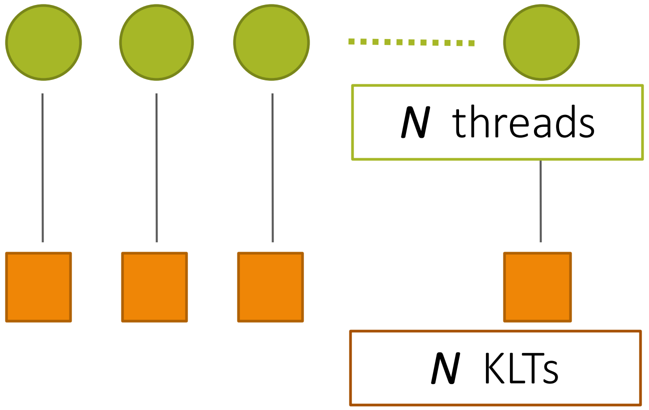
It is often beneficial to customize thread schedulers for specific workloads

The below example: work stealing scheduler for threads with priorities (e.g., for in situ analysis)

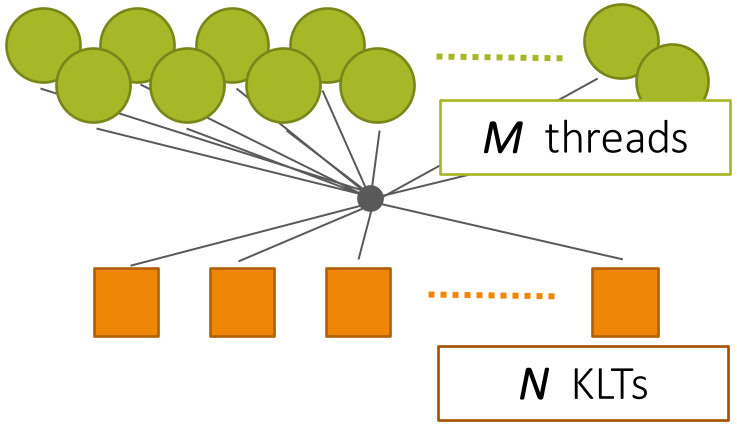


How about M:N Threads (User-Level Threads)?

1:1 Threads
(Kernel-Level Threads; KLTs)



M:N Threads
(User-Level Threads; ULTs)



Heavyweight threading operations
- e.g., thread creation, context switching

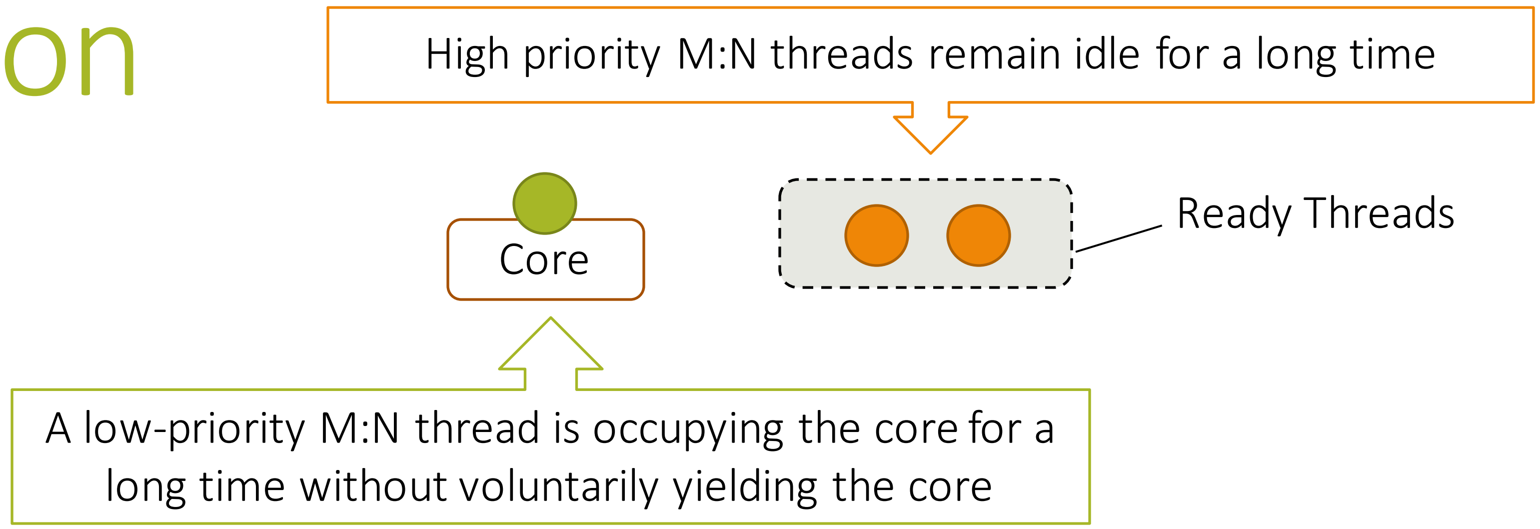
Inflexible scheduling policies
- controlled by the kernel

Lightweight threading operations
- without the involvement of the kernel

Flexible scheduling policies
- which can be defined in user space

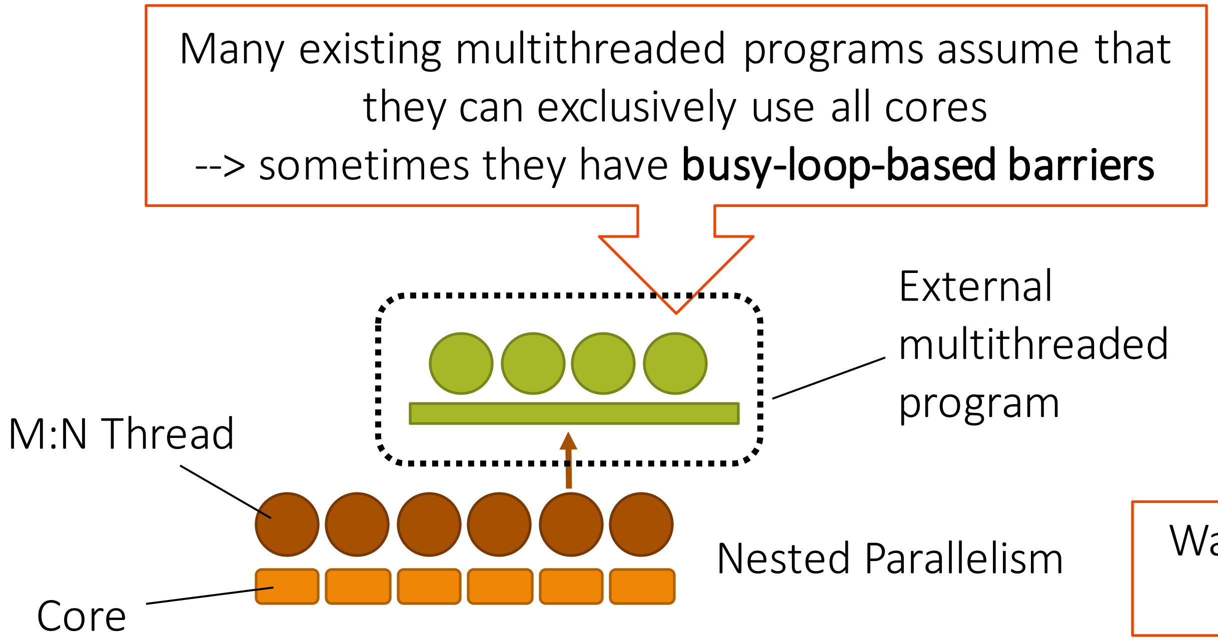
Many M:N thread-based parallel runtimes have been developed:
- Language: Chapel, X10, Charm++, OmpSs, etc.
- Library: Argobots, Qthreads, Massivethreads, etc.

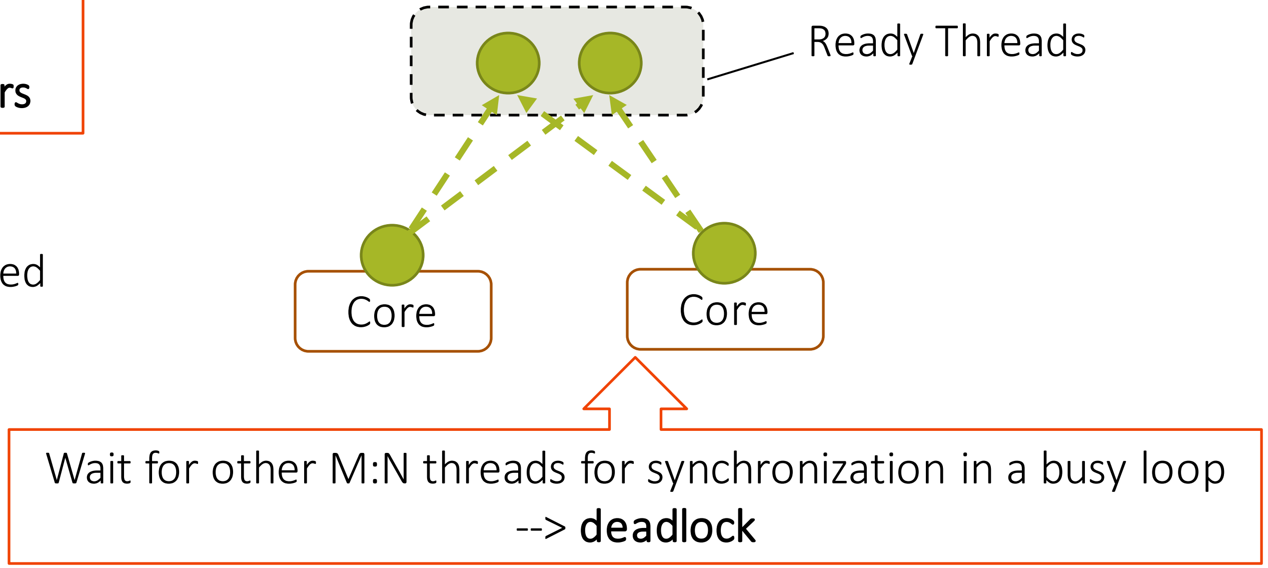
Lack of Preemption in M:N Threads



Loss of Prioritization

Deadlock





Contributions of This Paper

- Investigate **preemption techniques** for user-level M:N threads
  - They should be implemented as a pure **library** (that can be used from C/C++)
  - Two techniques:
    - Signal-Yield: an existing technique
    - KLT-Switching: our new proposal
- Provide optimizations for preemptive M:N threads based on detailed analysis
  - Scalable periodic timer interruption for preemption
  - Efficient implementation of KLT-switching
- Evaluate preemptive user-level threads implemented on Argobots [1]
  - Preemptive M:N threads can be as fast as nonpreemptive M:N threads in practice

[1] S. Seo, et al., "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems,* vol. 29, no. 3, pp. 512-526, 2017.

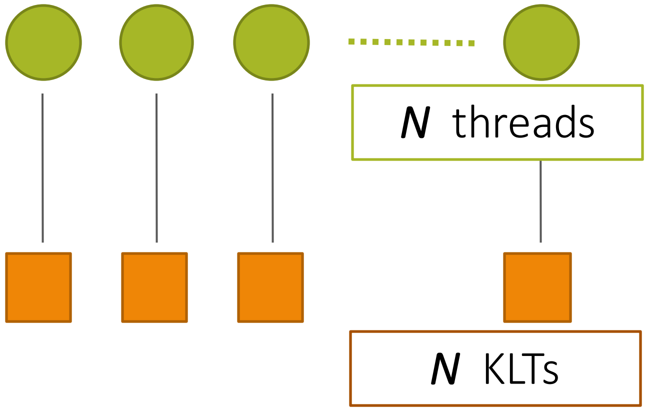
Outline

- 1:1 Threads and M:N Threads
- Design of Preemptive M:N Threads
  - Signal-Yield
  - KLT-Switching
- Optimizations for Preemptive M:N Threads
- Evaluation
  - Overhead of Preemption
  - Deadlock Prevention in Cholesky Decomposition
  - In Situ Analysis with LAMMPS
- Conclusion

Outline

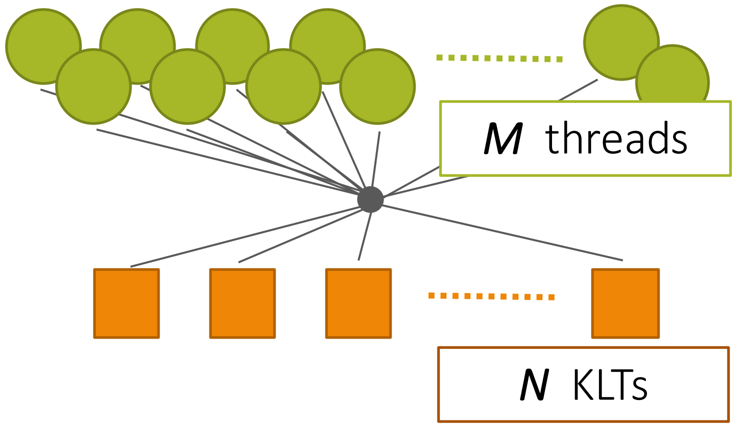
- 1:1 Threads and M:N Threads
- Design of Preemptive M:N Threads
  - Signal-Yield
  - KLT-Switching
- Optimizations for Preemptive M:N Threads
- Evaluation
  - Overhead of Preemption
  - Deadlock Prevention in Cholesky Decomposition
  - In Situ Analysis with LAMMPS
- Conclusion

1:1 Threads



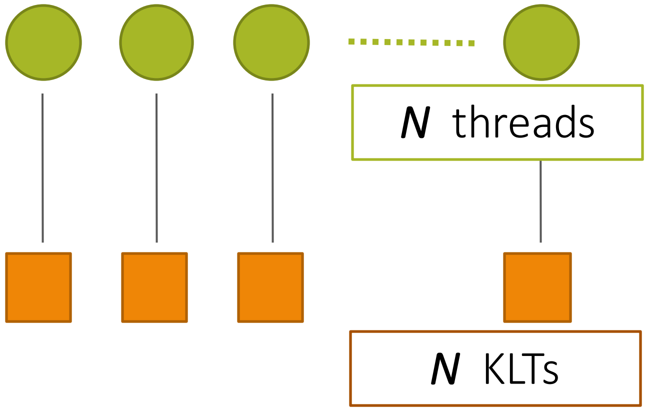
- Threads visible to the user are directly mapped to kernel-level threads (KLTs)
- Most implementations of Pthreads and OpenMP threads

M:N Threads



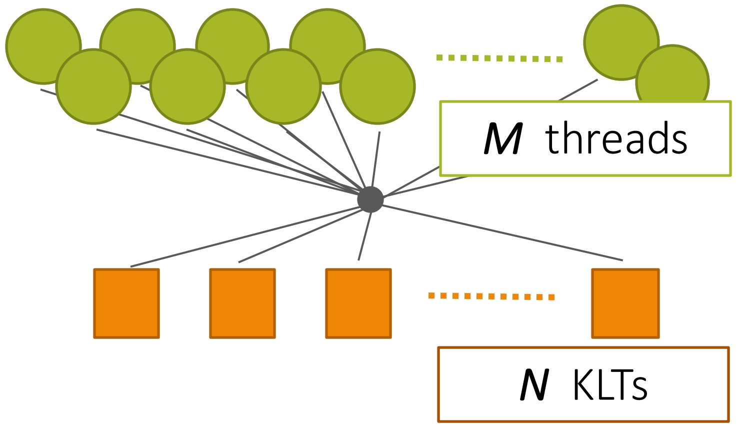
- Many user-visible threads are created and dynamically mapped to KLTs
- As many KLTs as the number of cores are usually created
- Often called "user-level threads (ULTs)"

1:1 Threads



The kernel can interrupt KLTs and schedule others on cores at any time (**preemption**)

M:N Threads



KLTs can be preempted by the kernel, but how to preempt M:N threads?

Outline

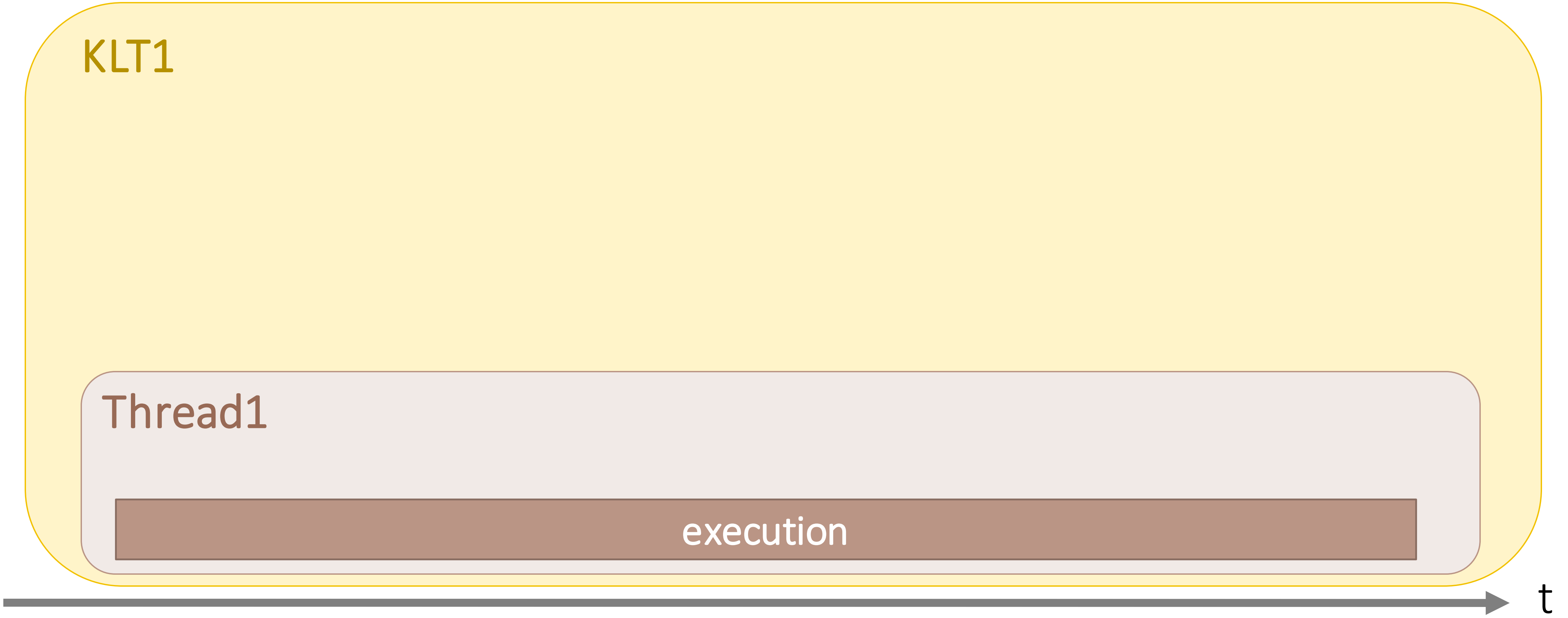
- 1:1 Threads and M:N Threads
- Design of Preemptive M:N Threads
  - Signal-Yield
  - KLT-Switching
- Optimizations for Preemptive M:N Threads
- Evaluation
  - Overhead of Preemption
  - Deadlock Prevention in Cholesky Decomposition
  - In Situ Analysis with LAMMPS
- Conclusion

Signal-Yield

[1] A. Anantaraman et al., "EDF-DVS scheduling on the IBM embedded PowerPC 405LP," *Proceedings of the IBM P= ac2 Conference*, 2004.
[2] M. S. Mollison and J. H. Anderson, "Bringing theory into practice: A userspace library for multicore real-time scheduling," *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.
[3] S. Boucher et al., "Lightweight preemptible functions," *2020 USENIX Annual Technical Conference*, 2020.
[4] Go 1.14 release notes. Available: https://golang.org/doc/go1.14

Idea: **Interrupt** execution of threads by a signal and **yield** in a signal handler

This technique has already been proposed in [1, 2, 3] and integrated to Go from v1.14 [4]

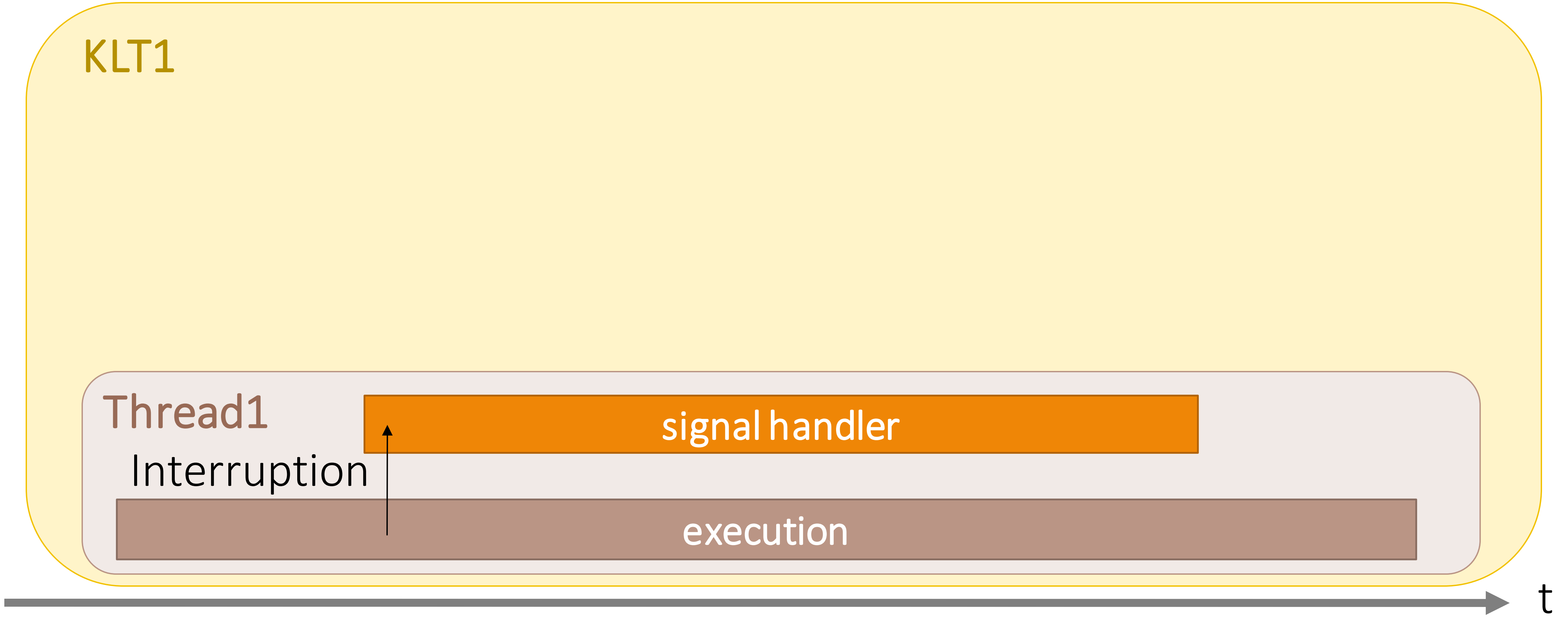


Signal-Yield

[1] A. Anantaraman et al., "EDF-DVS scheduling on the IBM embedded PowerPC 405LP," *Proceedings of the IBM P= ac2 Conference*, 2004.
[2] M. S. Mollison and J. H. Anderson, "Bringing theory into practice: A userspace library for multicore real-time scheduling," *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.
[3] S. Boucher et al., "Lightweight preemptible functions," *2020 USENIX Annual Technical Conference*, 2020.
[4] Go 1.14 release notes. Available: https://golang.org/doc/go1.14

Idea: **Interrupt** execution of threads by a signal and **yield** in a signal handler

This technique has already been proposed in [1, 2, 3] and integrated to Go from v1.14 [4]

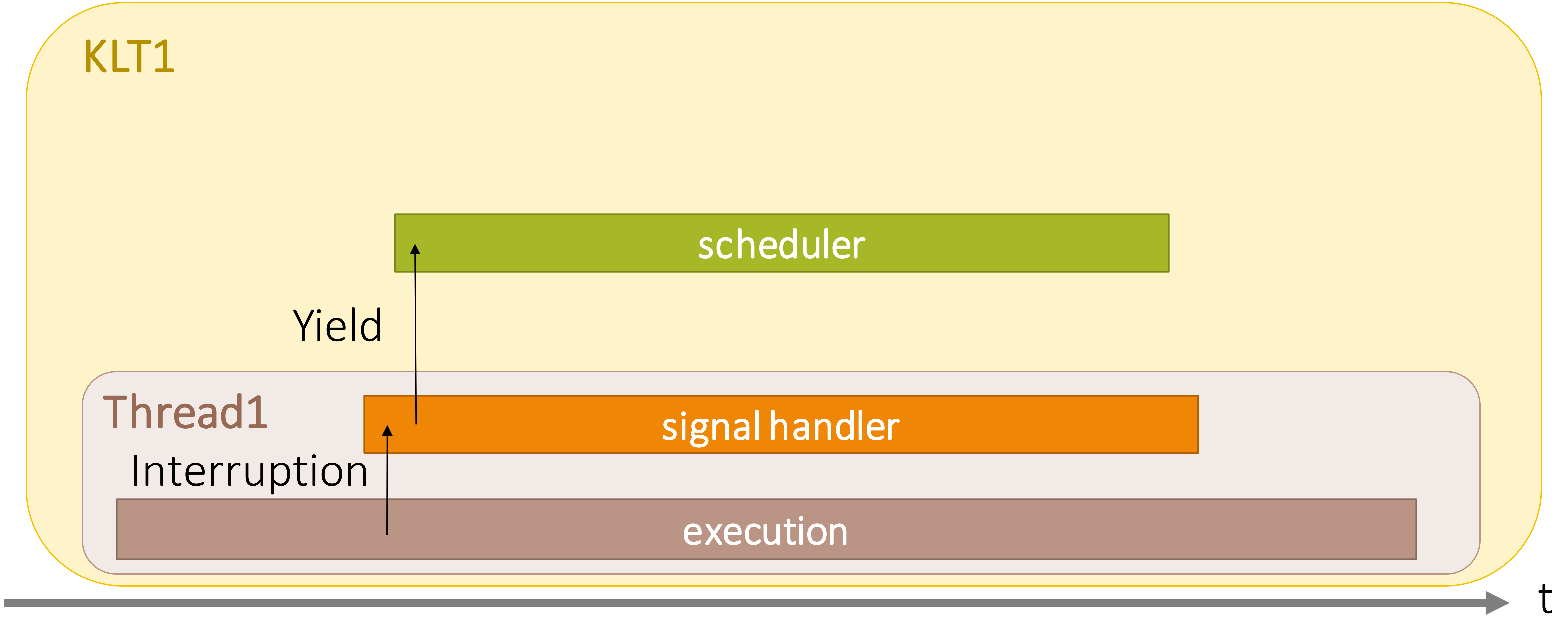


Signal-Yield

[1] A. Anantaraman et al., "EDF-DVS scheduling on the IBM embedded PowerPC 405LP," *Proceedings of the IBM P= ac2 Conference*, 2004.
[2] M. S. Mollison and J. H. Anderson, "Bringing theory into practice: A userspace library for multicore real-time scheduling," *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.
[3] S. Boucher et al., "Lightweight preemptible functions," *2020 USENIX Annual Technical Conference*, 2020.
[4] Go 1.14 release notes. Available: https://golang.org/doc/go1.14

Idea: **Interrupt** execution of threads by a signal and **yield** in a signal handler

This technique has already been proposed in [1, 2, 3] and integrated to Go from v1.14 [4]

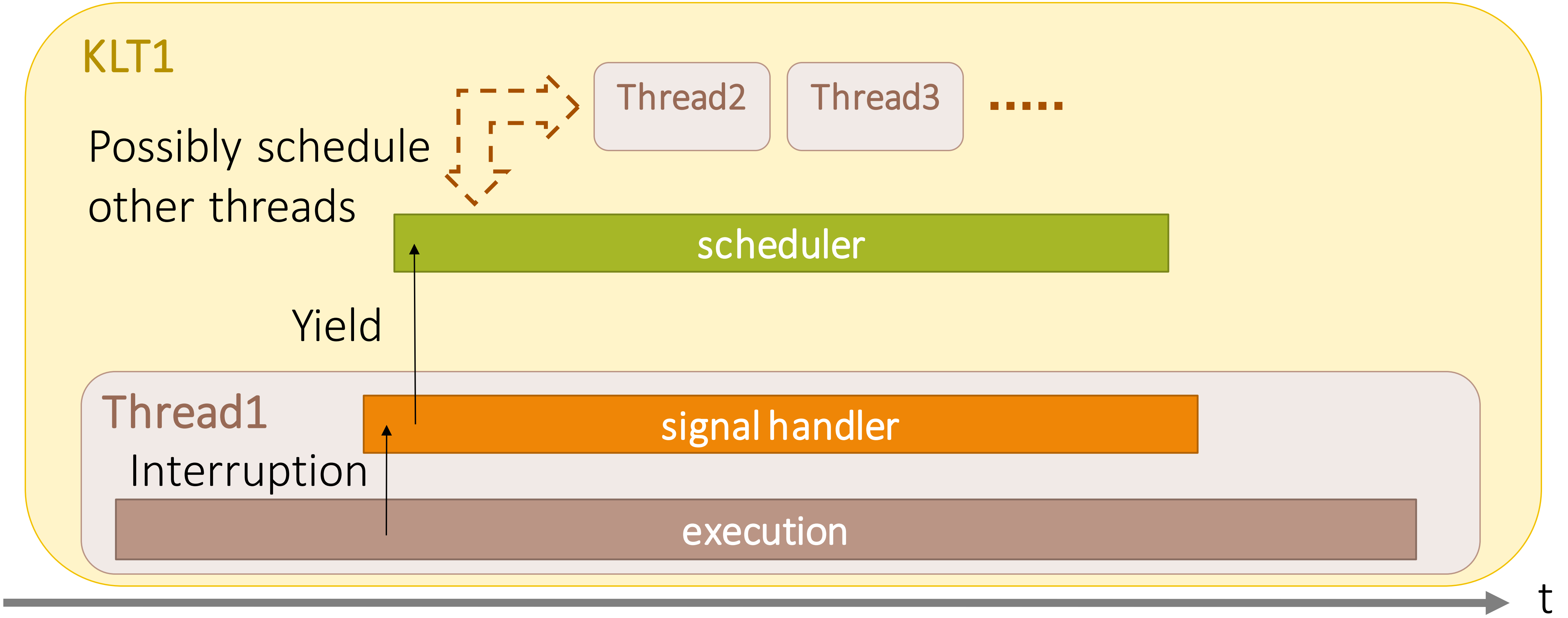


Signal-Yield

[1] A. Anantaraman et al., "EDF-DVS scheduling on the IBM embedded PowerPC 405LP," *Proceedings of the IBM P= ac2 Conference*, 2004.
[2] M. S. Mollison and J. H. Anderson, "Bringing theory into practice: A userspace library for multicore real-time scheduling," *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.
[3] S. Boucher et al., "Lightweight preemptible functions," *2020 USENIX Annual Technical Conference*, 2020.
[4] Go 1.14 release notes. Available: https://golang.org/doc/go1.14

Idea: **Interrupt** execution of threads by a signal and **yield** in a signal handler

This technique has already been proposed in [1, 2, 3] and integrated to Go from v1.14 [4]

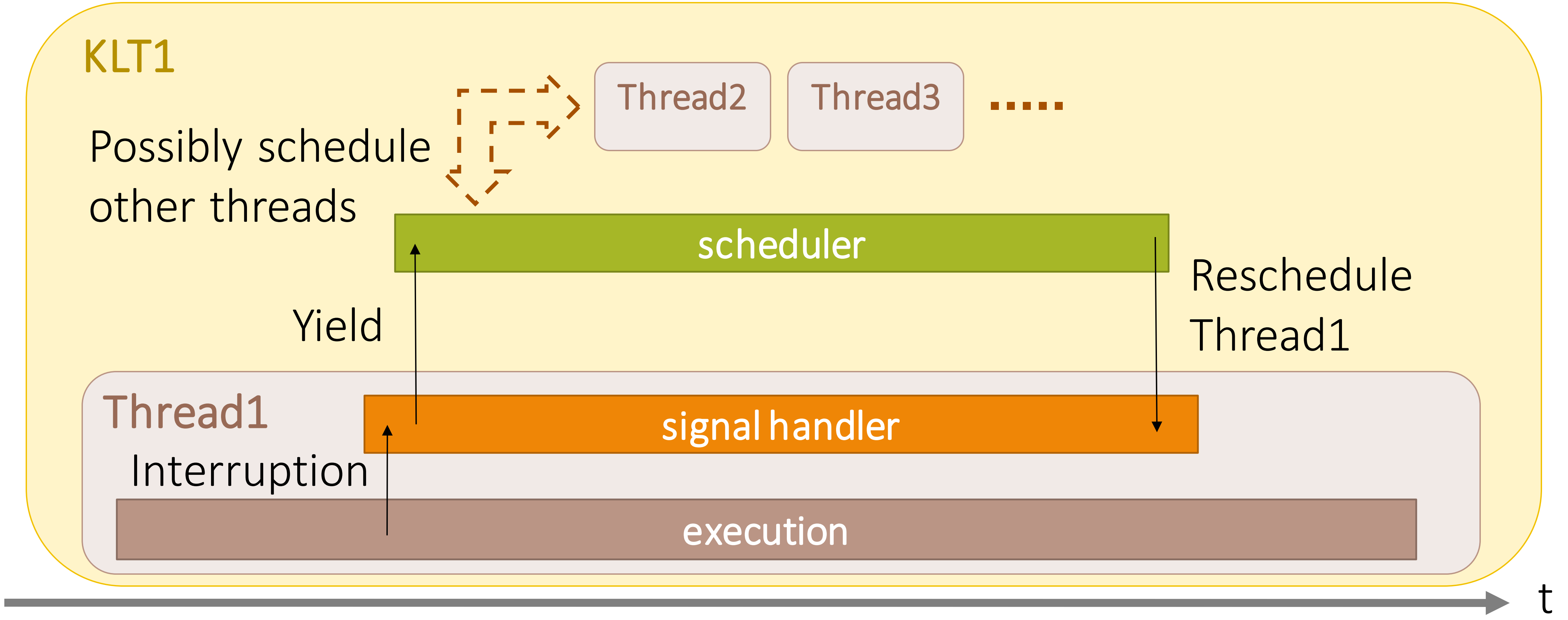


Signal-Yield

[1] A. Anantaraman et al., "EDF-DVS scheduling on the IBM embedded PowerPC 405LP," *Proceedings of the IBM P= ac2 Conference*, 2004.
[2] M. S. Mollison and J. H. Anderson, "Bringing theory into practice: A userspace library for multicore real-time scheduling," *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.
[3] S. Boucher et al., "Lightweight preemptible functions," *2020 USENIX Annual Technical Conference*, 2020.
[4] Go 1.14 release notes. Available: https://golang.org/doc/go1.14

Idea: **Interrupt** execution of threads by a signal and **yield** in a signal handler

This technique has already been proposed in [1, 2, 3] and integrated to Go from v1.14 [4]

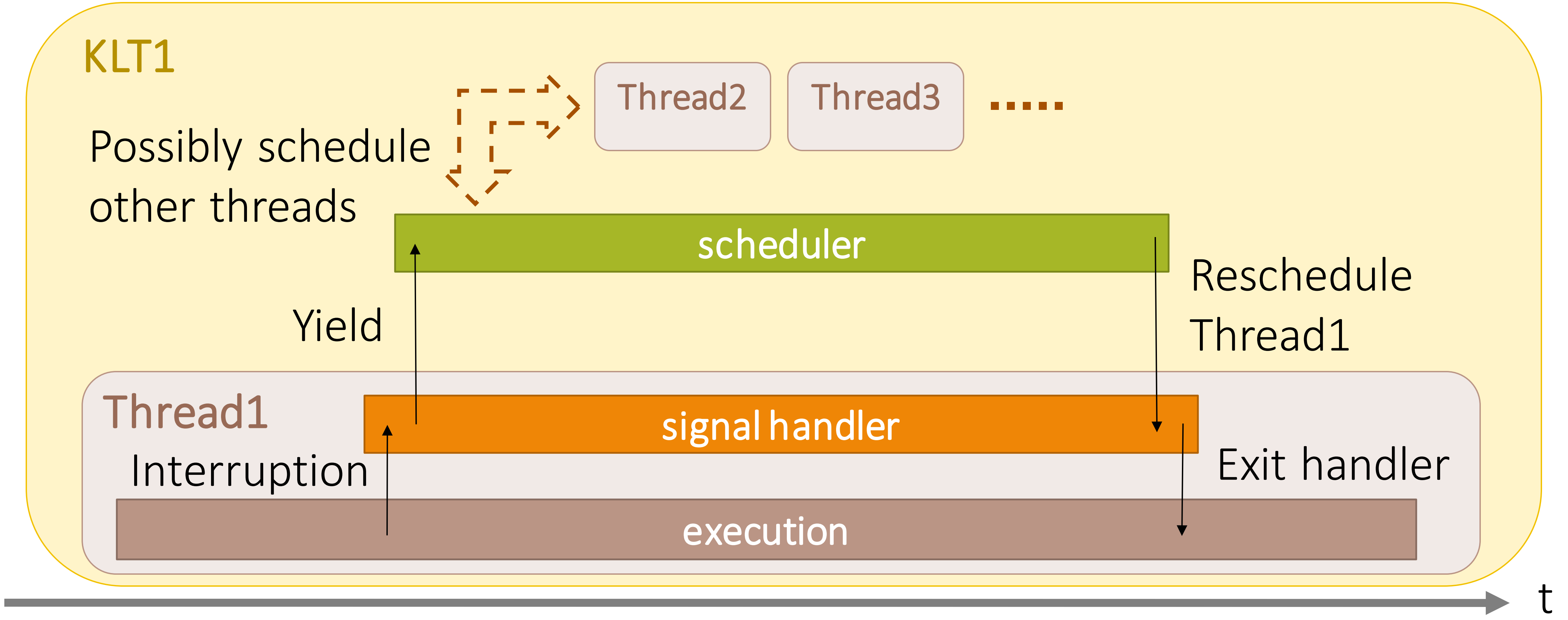


Signal-Yield

[1] A. Anantaraman et al., "EDF-DVS scheduling on the IBM embedded PowerPC 405LP," *Proceedings of the IBM P= ac2 Conference*, 2004.
[2] M. S. Mollison and J. H. Anderson, "Bringing theory into practice: A userspace library for multicore real-time scheduling," *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.
[3] S. Boucher et al., "Lightweight preemptible functions," *2020 USENIX Annual Technical Conference*, 2020.
[4] Go 1.14 release notes. Available: https://golang.org/doc/go1.14

Idea: **Interrupt** execution of threads by a signal and **yield** in a signal handler

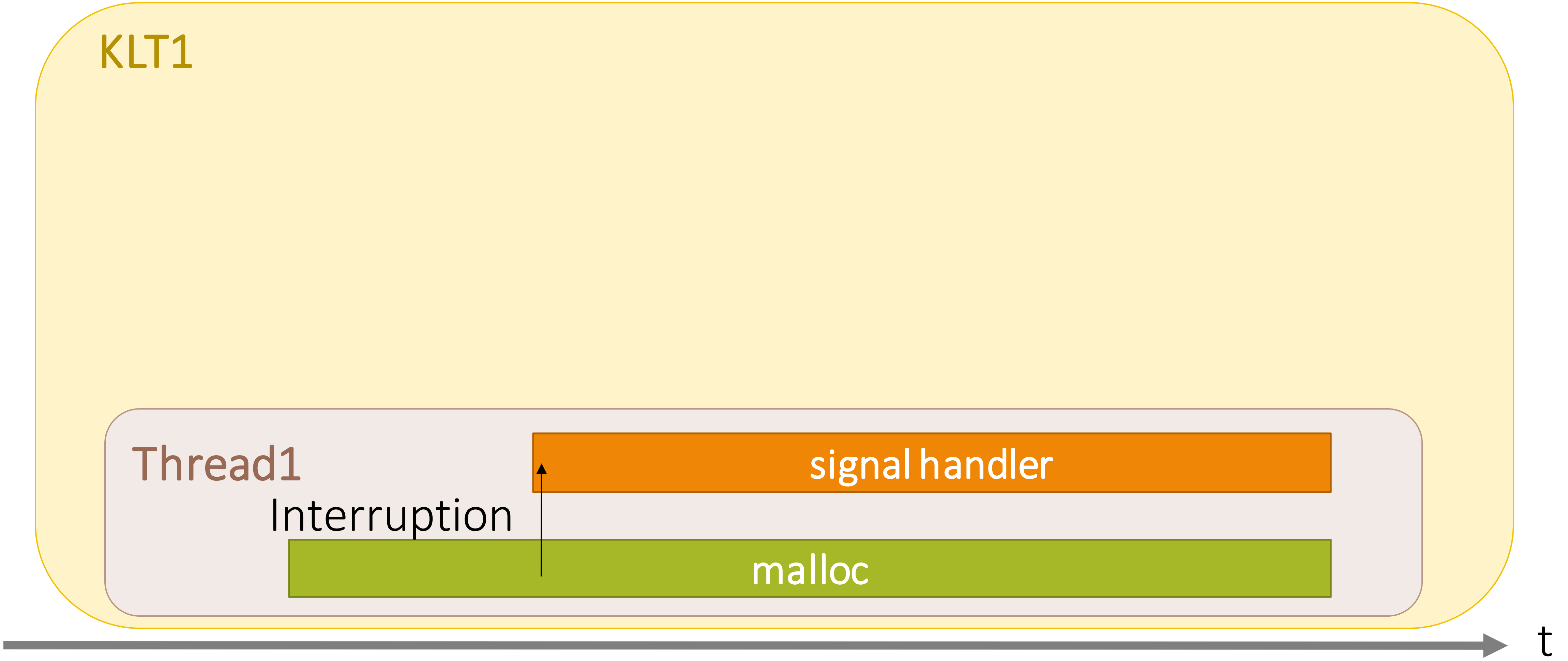
This technique has already been proposed in [1, 2, 3] and integrated to Go from v1.14 [4]



KLT-Dependence Issue in Signal-Yield

Some existing functions can access **KLT-local data**, not M:N thread-local data

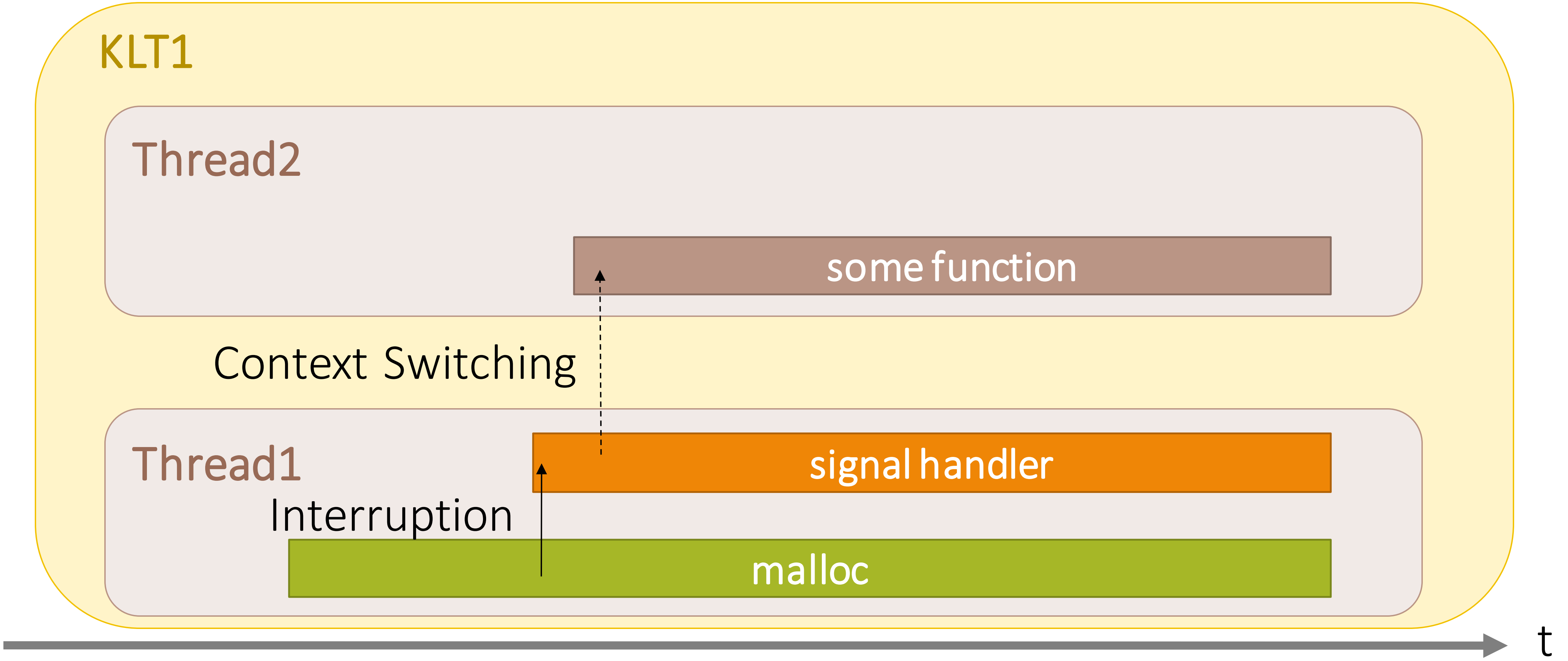
e.g., Glibc malloc() uses KLT-local data for KLT-local caching of memory blocks



KLT-Dependence Issue in Signal-Yield

Some existing functions can access **KLT-local data**, not M:N thread-local data

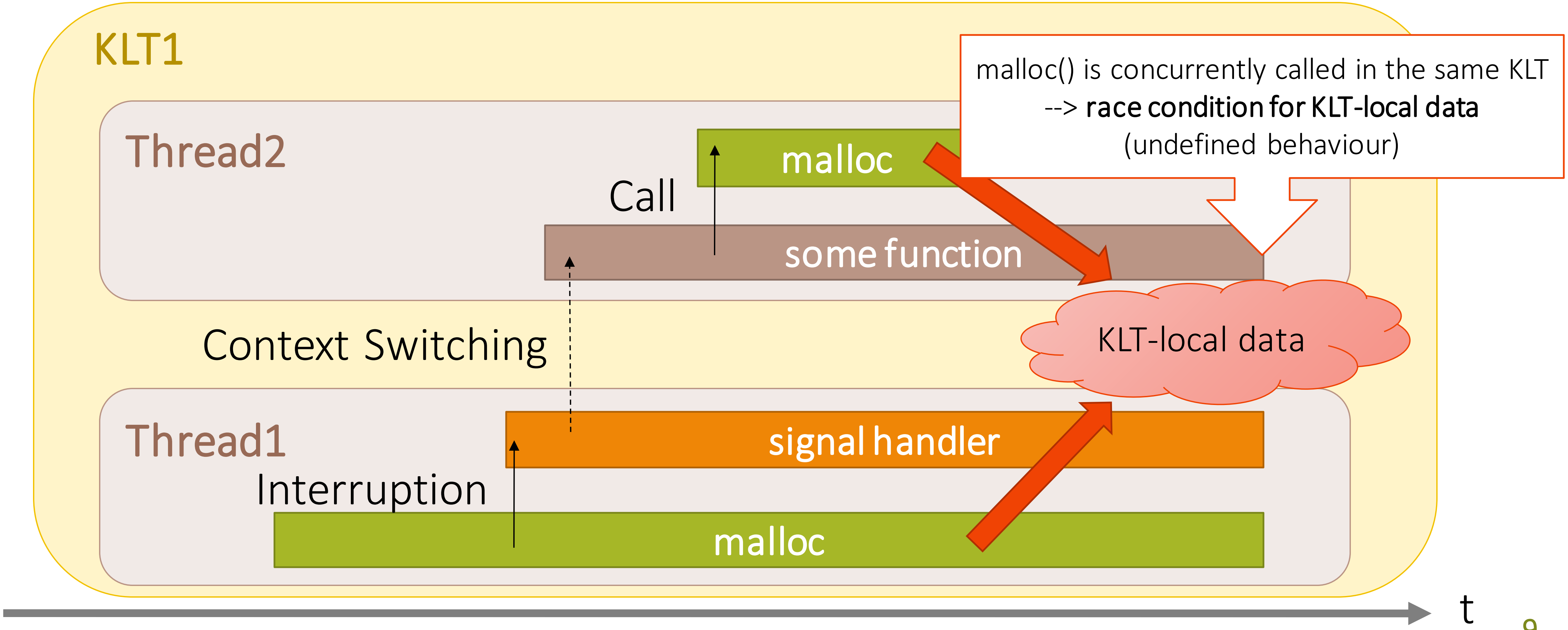
e.g., Glibc malloc() uses KLT-local data for KLT-local caching of memory blocks



KLT-Dependence Issue in Signal-Yield

Some existing functions can access **KLT-local data**, not M:N thread-local data

e.g., Glibc malloc() uses KLT-local data for KLT-local caching of memory blocks



How to Support Safe Preemption?

KLT-Dependence Issue:

- **KLT-local data** are assumed to be accessed **sequentially**, but signal-yield breaks this assumption
  - Multiple thread contexts can run on the same KLT
  - By using signal-yield, threads can be interrupted while accessing KLT-local data

Some existing library functions are not M:N thread-aware and access KLT-local data directly
(e.g., Glibc malloc uses KLT-local caching for memory blocks)

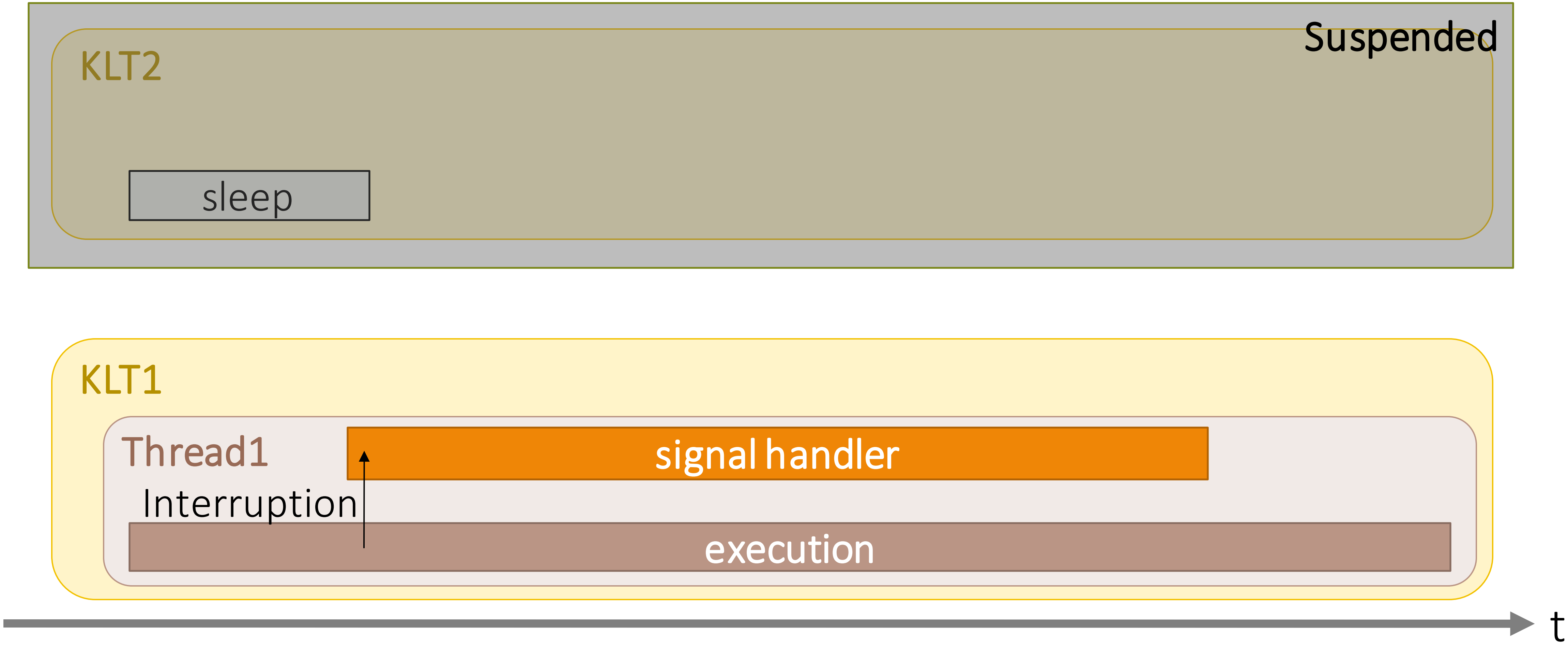
Solution:

- **Switch to another KLT when preemption happens** so that KLT-local data are not modified while being preempted
- We call it **KLT-switching** (our new proposal)

KLT-Switching

Idea (our proposal): switching to another KLT **only when preemption happens**

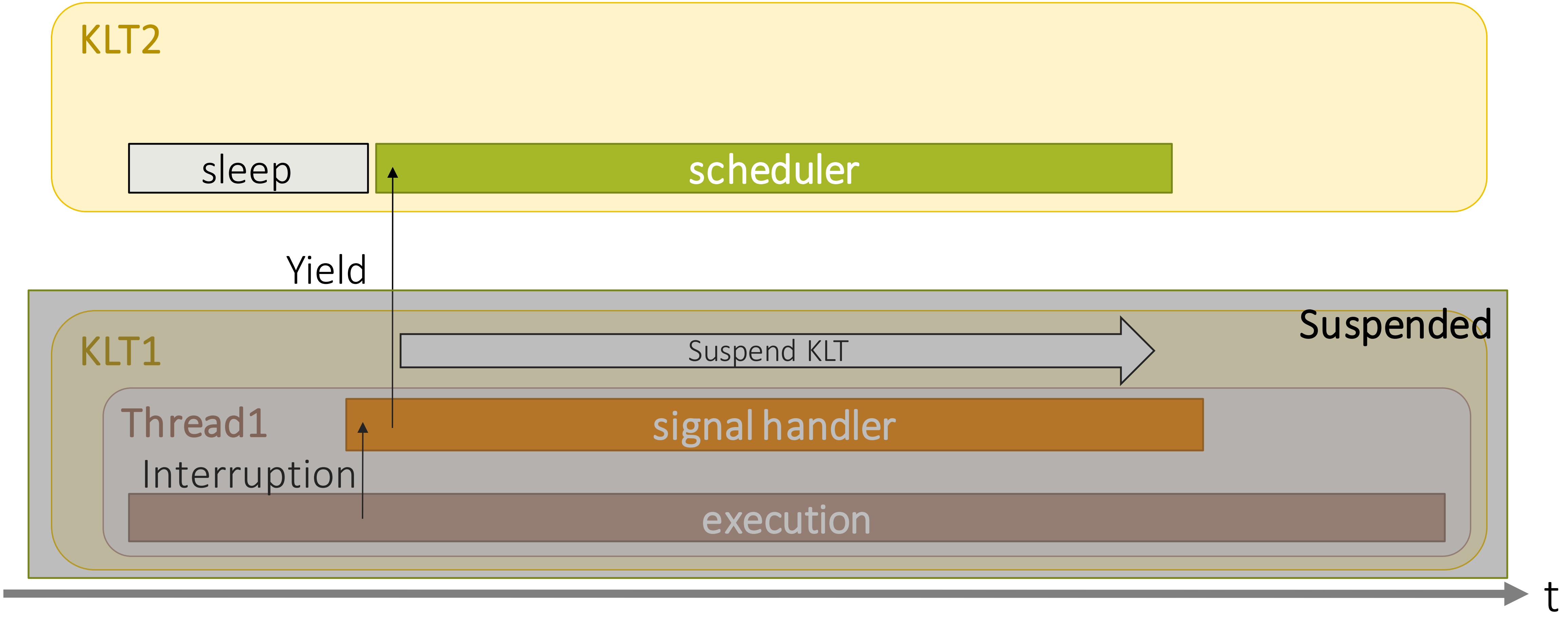
-> KLT-local data are not modified while being preempted



KLT-Switching

Idea (our proposal): switching to another KLT **only when preemption happens**

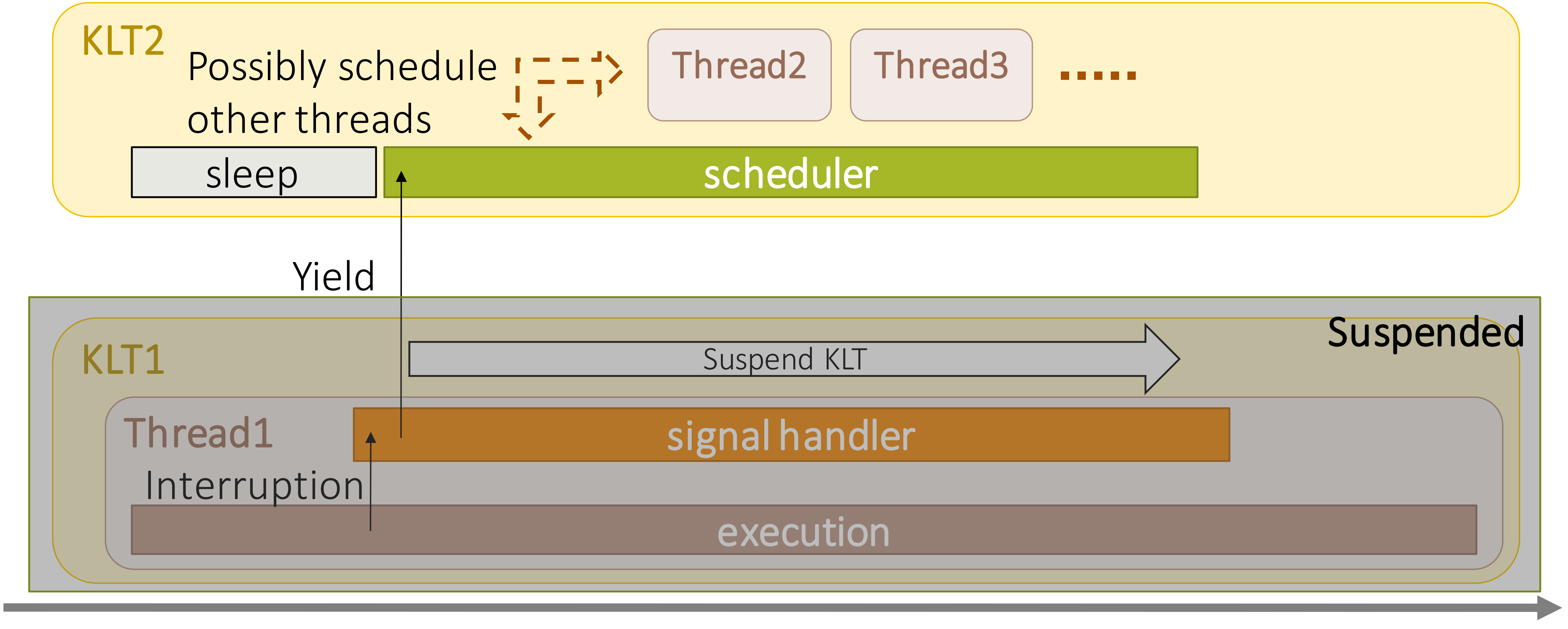
-> KLT-local data are not modified while being preempted



KLT-Switching

Idea (our proposal): switching to another KLT **only when preemption happens**

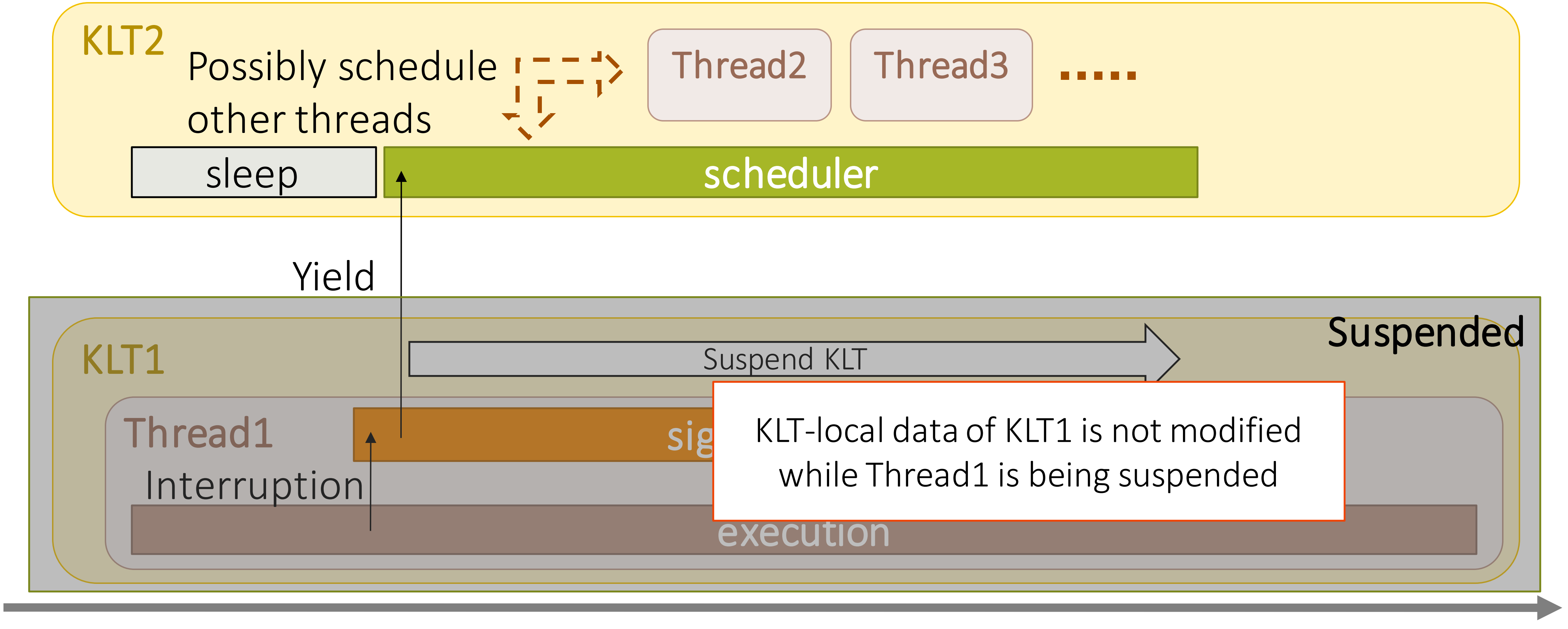
-> KLT-local data are not modified while being preempted



KLT-Switching

Idea (our proposal): switching to another KLT **only when preemption happens**

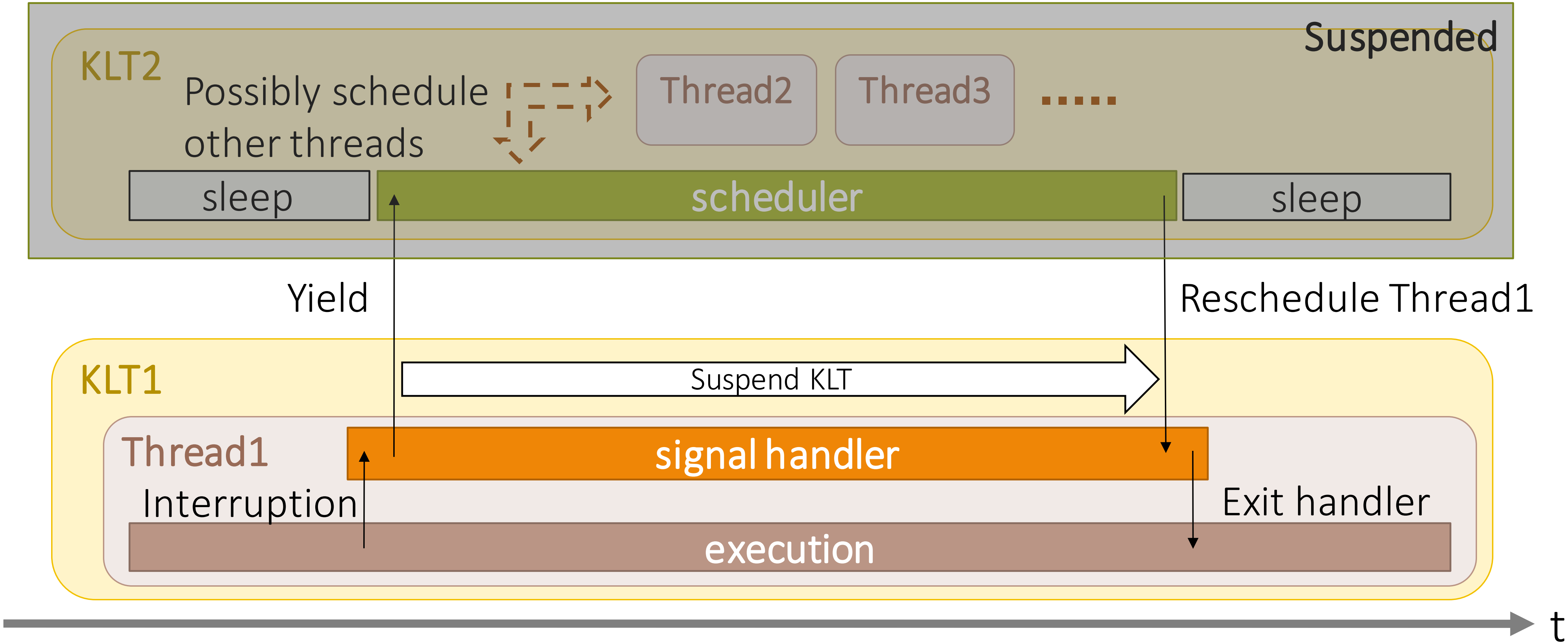
-> KLT-local data are not modified while being preempted



KLT-Switching

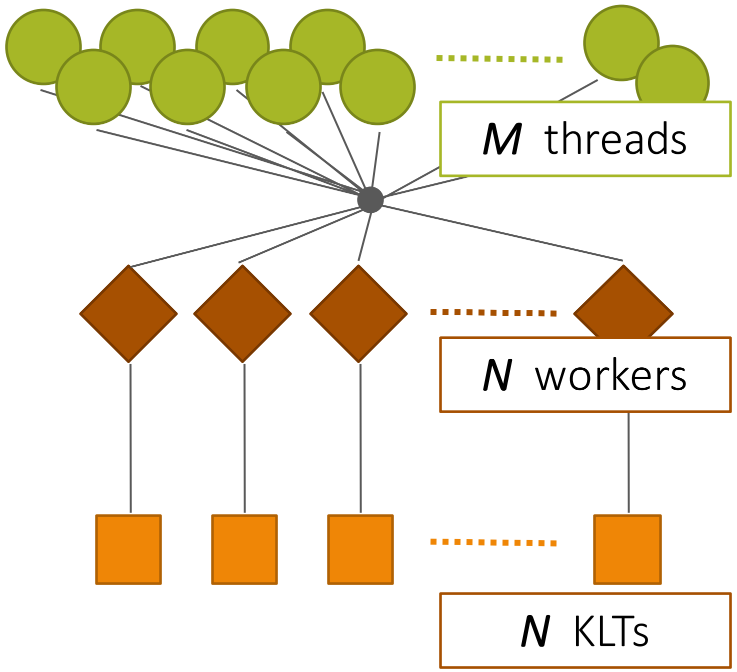
Idea (our proposal): switching to another KLT **only when preemption happens**

-> KLT-local data are not modified while being preempted

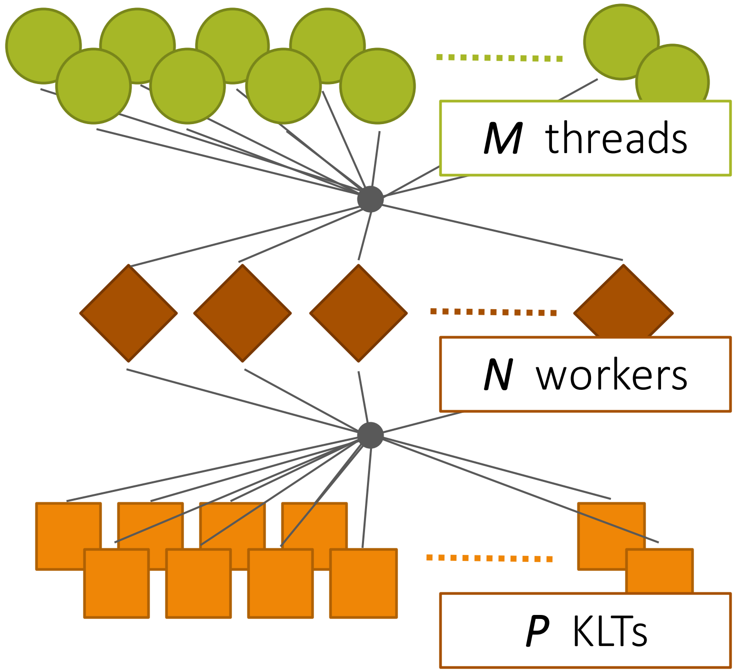


Thread Mapping in Preemptive M:N Threads

Worker: virtualization of physical cores



Nonpreemptive/signal-yield



KLT-switching

In KLT-switching,
- There is no static mapping between workers and KLTs
- Only "N" KLTs are active at the same time
  --> **Thread scheduling is customizable by the user**

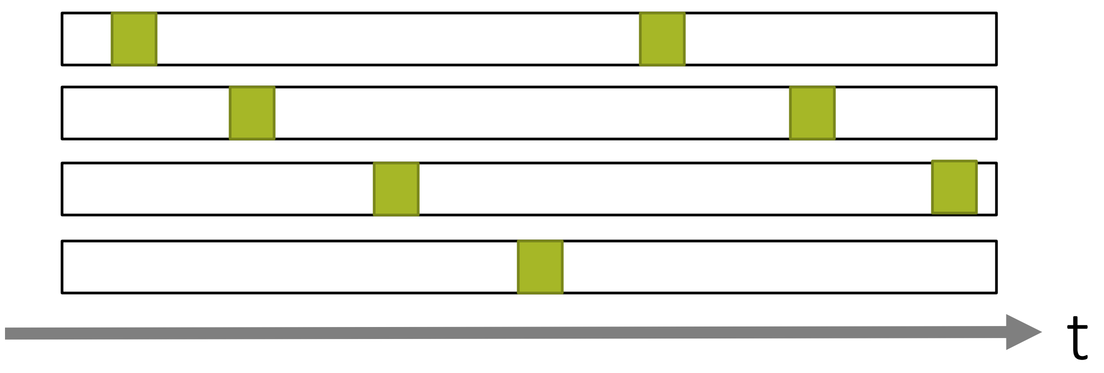
Outline

- 1:1 Threads and M:N Threads
- Design of Preemptive M:N Threads
  - Signal-Yield
  - KLT-Switching
- Optimizations for Preemptive M:N Threads
- Evaluation
  - Overhead of Preemption
  - Deadlock Prevention in Cholesky Decomposition
  - In Situ Analysis with LAMMPS
- Conclusion

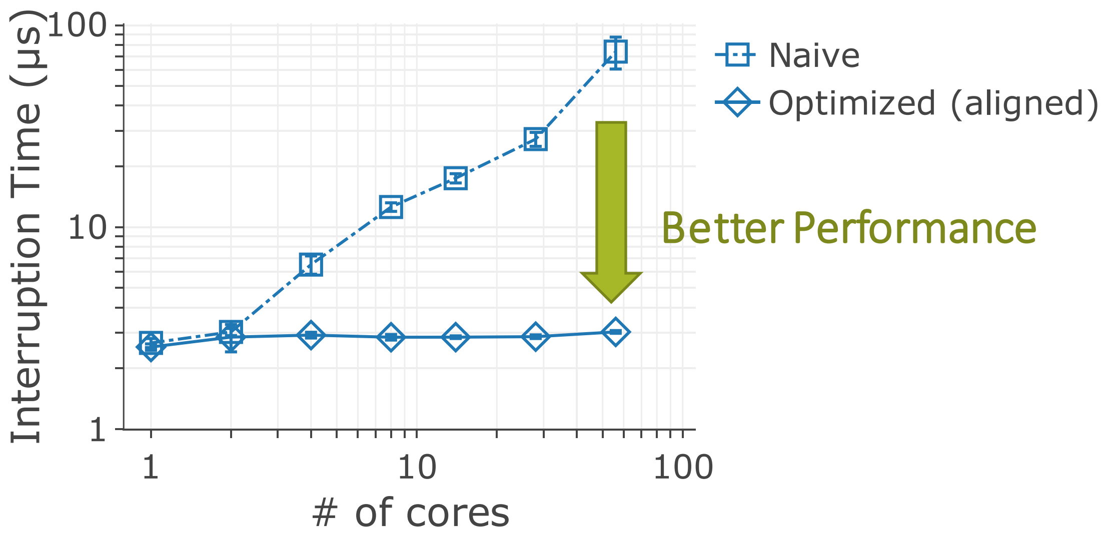
Optimizations for Preemption Timer

Preemption Timer: periodic timer interruption for each worker (using timer_create() syscall)

Optimization of aligning timings of interruption



Observation: When signals are delivered at the same time across different cores, large amounts of time are consumed because of **lock contention** in the kernel



Optimizations for KLT-Switching

- **Use futex in Linux for suspending and resuming KLTs**
  - The POSIX-compliant implementation use sigsuspend() and pthread_kill()
    - They can be called from signal handlers (async-signal-safe)
    - Heavyweight because of additional signal handling
- **Worker-local pools for suspended KLTs**
  - Each worker has its own KLT pool for caching
  - Avoid resetting CPU affinity of KLTs
    - When a KLT is mapped to a worker, the CPU affinity of the KLT should be set to the worker's CPU

Summary of Thread Implementations

(*) Whether or not the KLT-dependence issue happens

	Explicit Threading Operations	Overhead of Preemption	Scheduling Policies	Safety of Preemption (*)
1:1 Threads	Heavyweight	Low (2.8 us)	Not customizable	Safe
Nonpreemptive M:N Threads	Lightweight	-	Customizable	-
Preemptive M:N Threads (Signal-Yield)	Lightweight	Medium (3.5 us)	Customizable	Unsafe
Preemptive M:N Threads (KLT-Switching)	Lightweight	High (9.9 us)	Customizable	Safe

High overhead of preemption is acceptable because preemption is infrequent enough in practice (evaluated later)

Outline

- 1:1 Threads and M:N Threads
- Design of Preemptive M:N Threads
  - Signal-Yield
  - KLT-Switching
- Optimizations for Preemptive M:N Threads
- Evaluation
  - Overhead of Preemption
  - Deadlock Prevention in Cholesky Decomposition
  - In Situ Analysis with LAMMPS
- Conclusion

Evaluation

CPU: Intel Xeon Platinum 8180M (Skylake)

# of sockets: 2

# of cores: 56 (28 x 2)

Preemptive M:N threads are implemented on Argobots, an M:N threading library

Evaluation:

1. **Overhead of Preemption** (using a microbenchmark)
2. **Deadlock Prevention in Cholesky Decomposition**
   Preemptive M:N threads resolve a deadlock and outperforms 1:1 threads
3. **In Situ Analysis with LAMMPS**
   Evaluation of the benefit of user-defined schedulers with preemption

Overhead of Preemption

Using a compute-intensive program

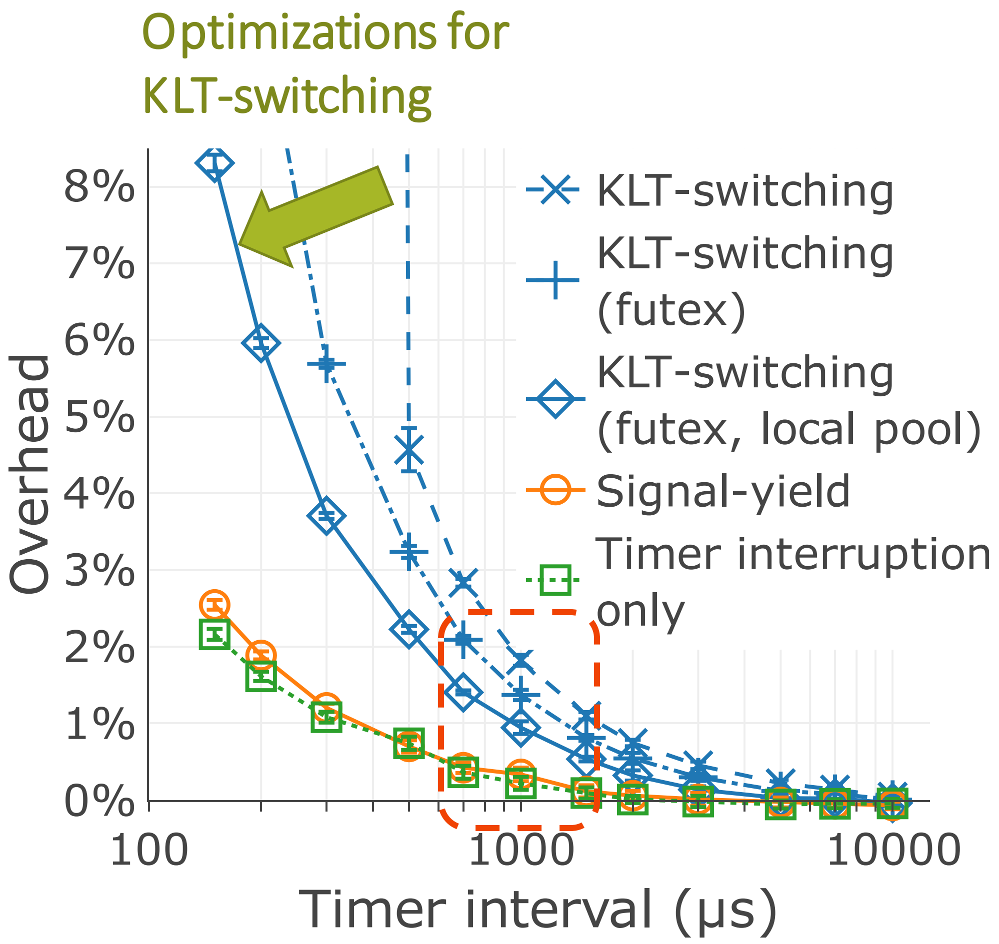
Baseline: nonpreemptive M:N threads

KLT-switching costs higher than signal-yield

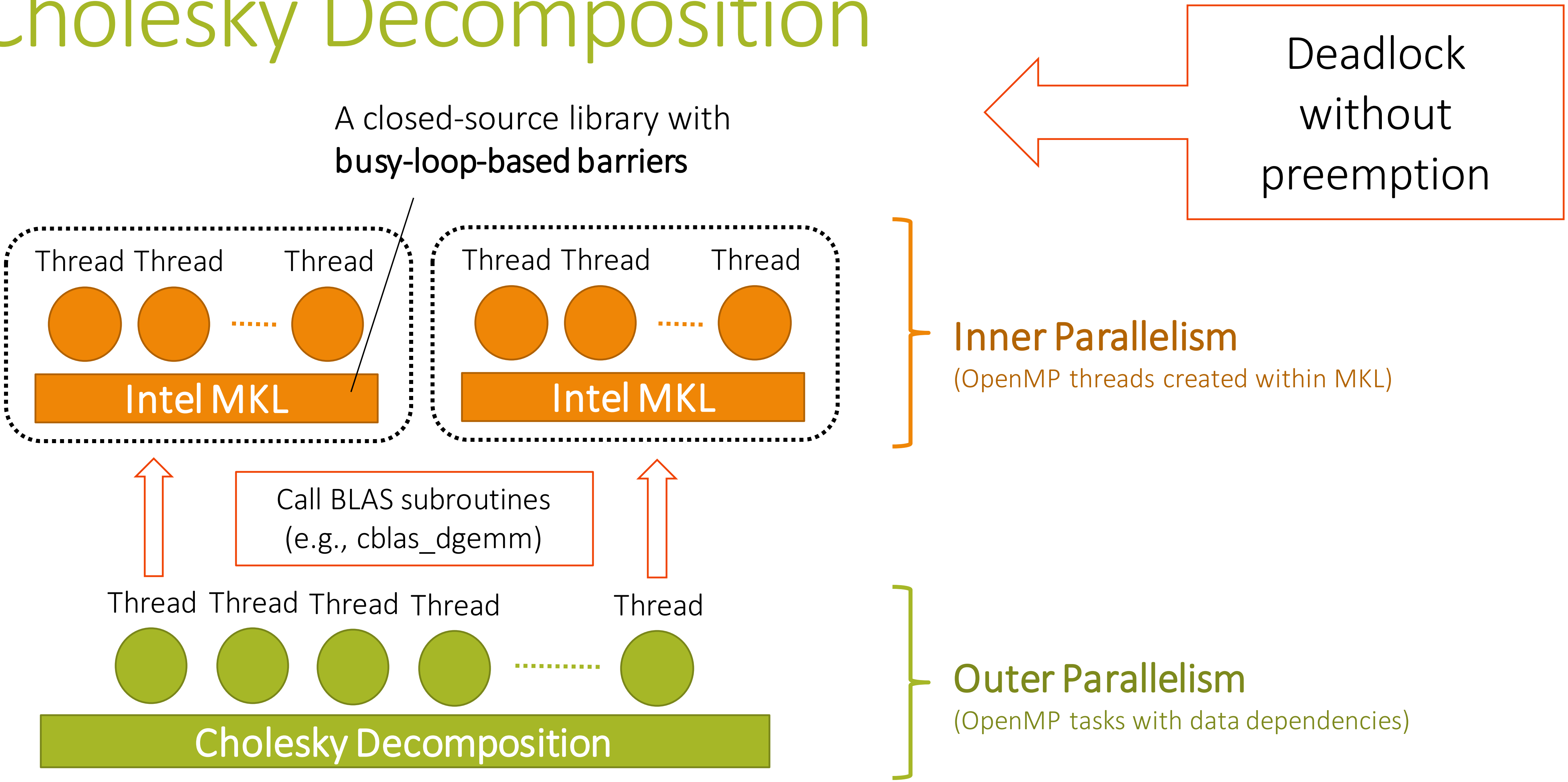
The overhead of KLT-switching is **only ~ 1% with the timer interval of 1 ms**

OS preemption interval is typically in the millisecond range

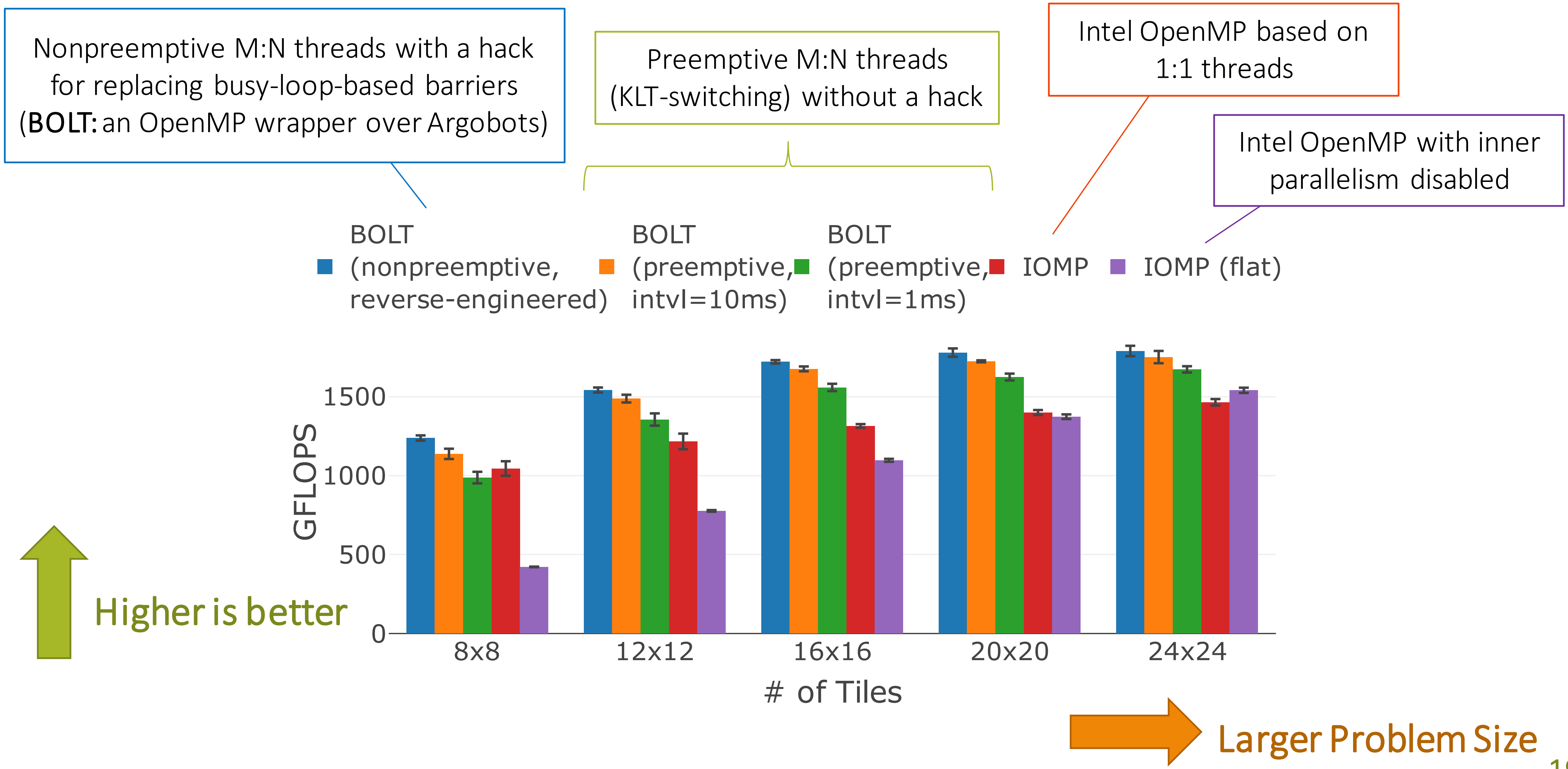
Note: Explicit threading operations (e.g., context switching and thread creation) are as lightweight as nonpreemptive threads



Cholesky Decomposition



Evaluation of Cholesky Decomposition

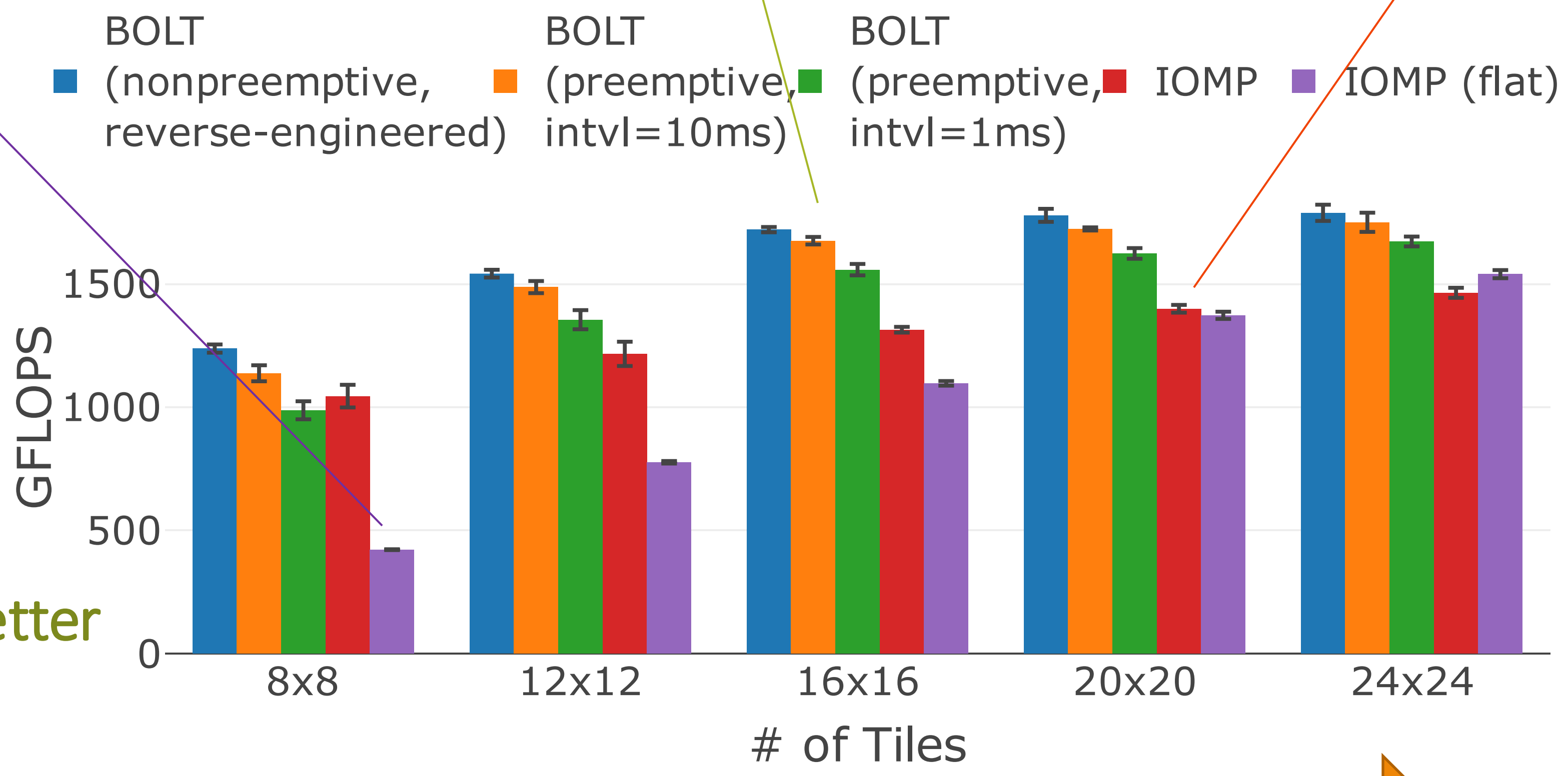


Evaluation of Cholesky Decomposition

Preemption adds some overheads to nonpreemptive M:N threads, but performs better than 1:1 threads

Outer parallelism is not enough
-> Nested parallelism is beneficial

Heavyweight threading operations of 1:1 threads with nested parallelism



Higher is better

Larger Problem Size

In Situ Analysis with LAMMPS

LAMMPS: a widely-used molecular dynamics simulator

In situ analysis: a modern way to perform analysis and simulation at the same time

Lower priority

Higher priority

Analysis threads are created based on the progress of simulation threads

- Thus, analysis threads should be evicted from cores in favor of simulation threads
- **Preemptive scheduling is effective for thread prioritization**

Evaluation of In Situ Analysis with LAMMPS

Line Plot: relative overhead compared with the baseline

Baseline: simulation only (without analysis)

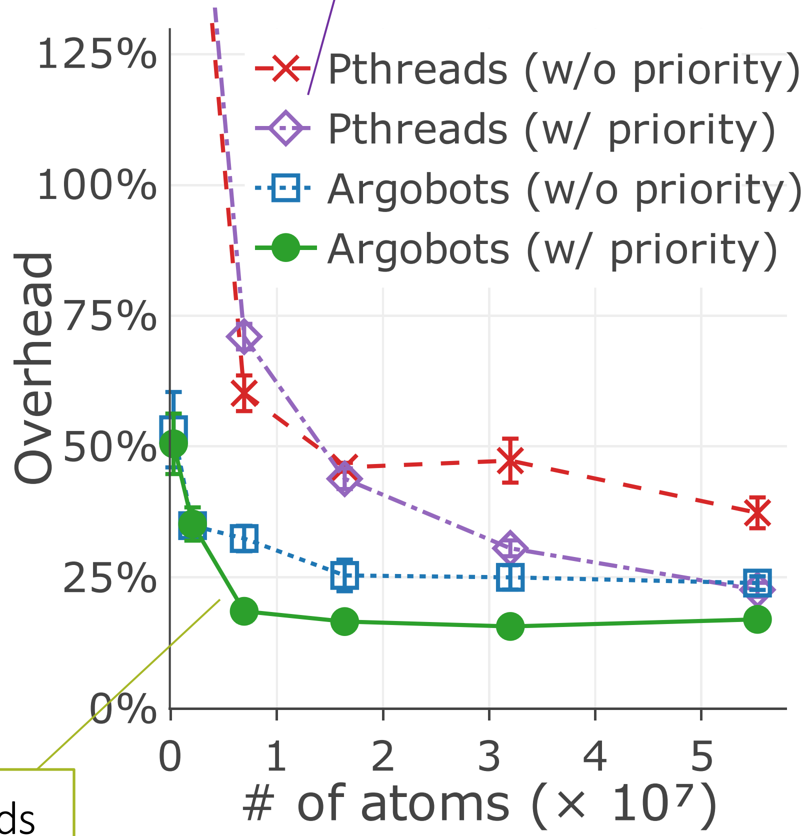
1:1 Threads (Pthreads) add large overheads because of heavyweight threading operations

Preemptive M:N threads perform better, thanks to...

- Lightweight threading operations
- Efficient prioritization of threads enabled by preemption

Lower is better

Priority of Pthreads is set via "nice" values



Preemptive M:N threads

Outline

- 1:1 Threads and M:N Threads
- Design of Preemptive M:N Threads
  - Signal-Yield
  - KLT-Switching
- Optimizations for Preemptive M:N Threads
- Evaluation
  - Overhead of Preemption
  - Deadlock Prevention in Cholesky Decomposition
  - In Situ Analysis with LAMMPS
- Conclusion

Artifact: https://doi.org/10.5281/zenodo.4420552

Conclusion

Investigated two techniques for preemptive M:N threads:

- Signal-yield: lower cost, but unsafe
- KLT-switching: Higher cost, but covers a wider range of programs

Implemented preemptive M:N threads on Argobots and evaluated them

- They can **avoid a deadlock** and outperform runtimes based on 1:1 threads
- They enable **efficient user-level schedulers specialized for specific workloads**
  - With nonpreemptive M:N threads, priority-based scheduling was hard to support

Preemption brings more freedom to M:N threads in implementing efficient user-defined schedulers with low overheads!