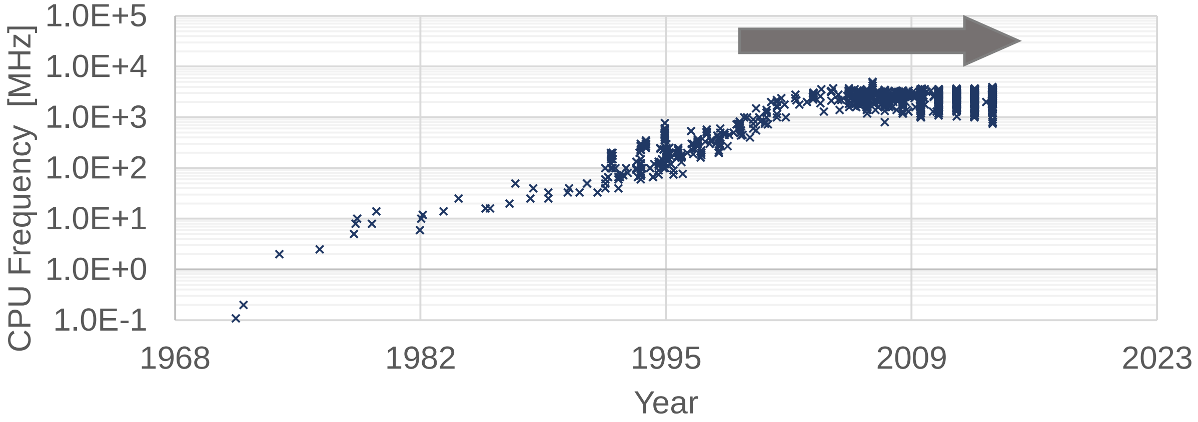
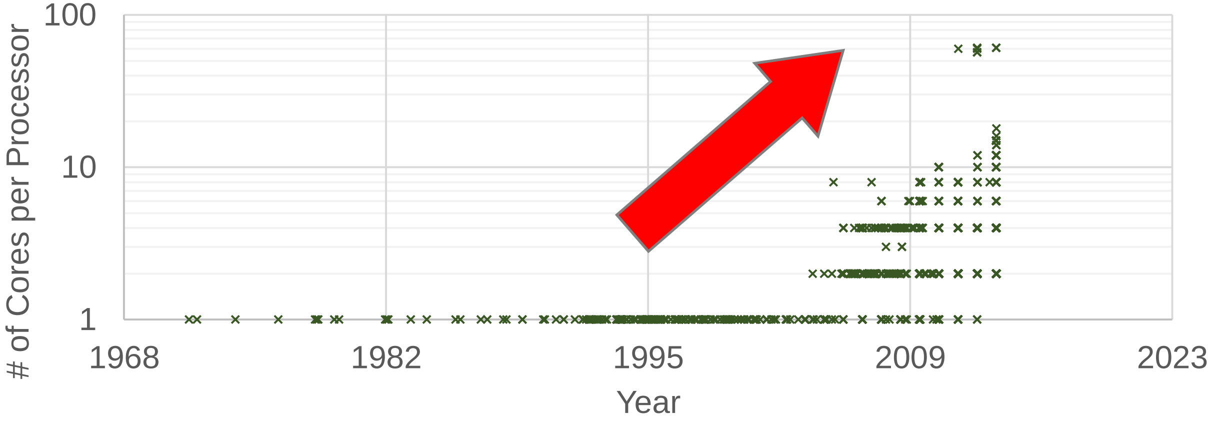
Lessons Learned from Analyzing Dynamic Promotion for User-Level Threading

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

Demands for Lightweight Threads



Frequency

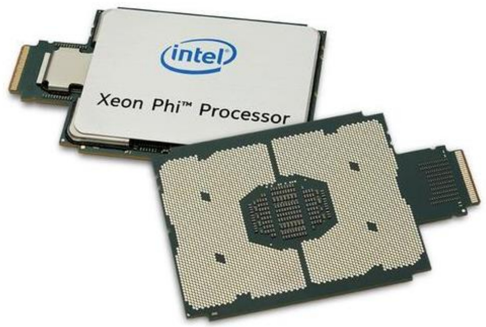


# of cores

CPU DB (http://cpudb.stanford.edu/)

- Increase of cores in a processor.
- Finer-grained parallelism is important to exploit modern CPUs.

➡ Lightweight threads are demanded.

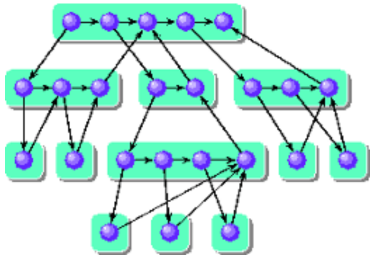


ARM ThunderX2 up to 32 cores, 128 HWTs (https://www.servethehome.com/cavium-thunderx2-review-benchmarks-real-arm-server-option/)

Intel Xeon Phi (Knights Landing) 72 cores, 288 HWTs (https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing)

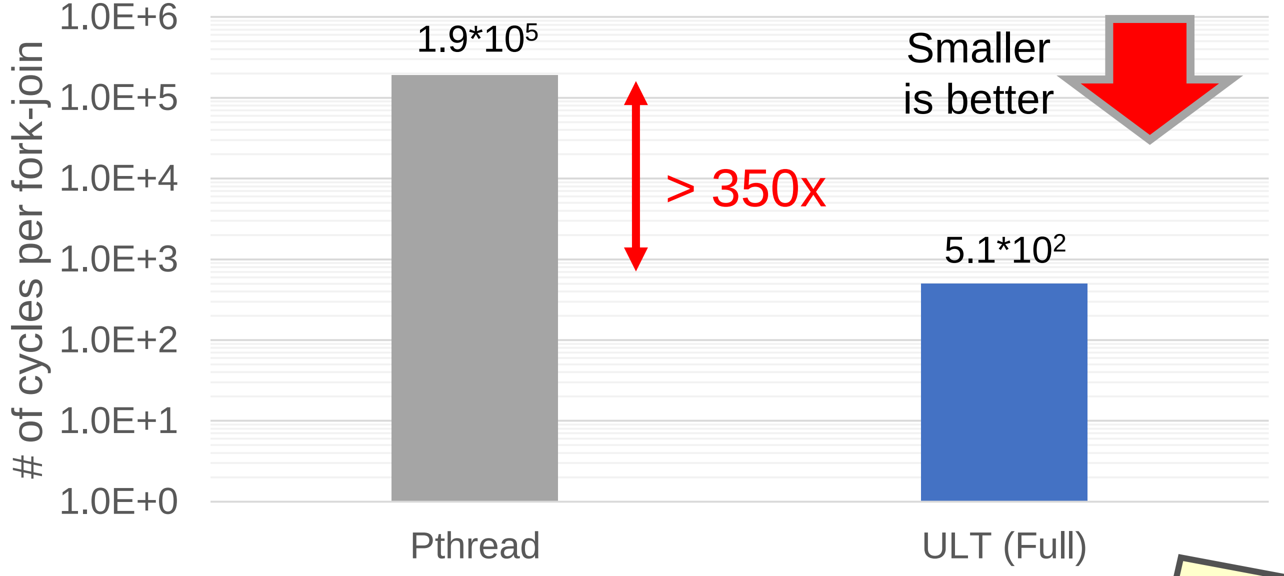
User-Level Threads

- Numerous parallel systems adopt user-level threads (ULTs)
  - Sometimes more than 100x faster than OS-level threads (=kernel threads, e.g., Pthreads)
  - Adopted as lightweight parallel units.
    - Cilk, Intel TBB, CilkPlus, OmpSs (=Nanos), Qthreads, Intel/LLVM OpenMP, Charm++ (=Converse), Filaments, MassiveThreads, Argobots and many



From each official webpage

OS-Level Threads vs. User-Level Threads



- ULTs are 350x faster than Pthreads

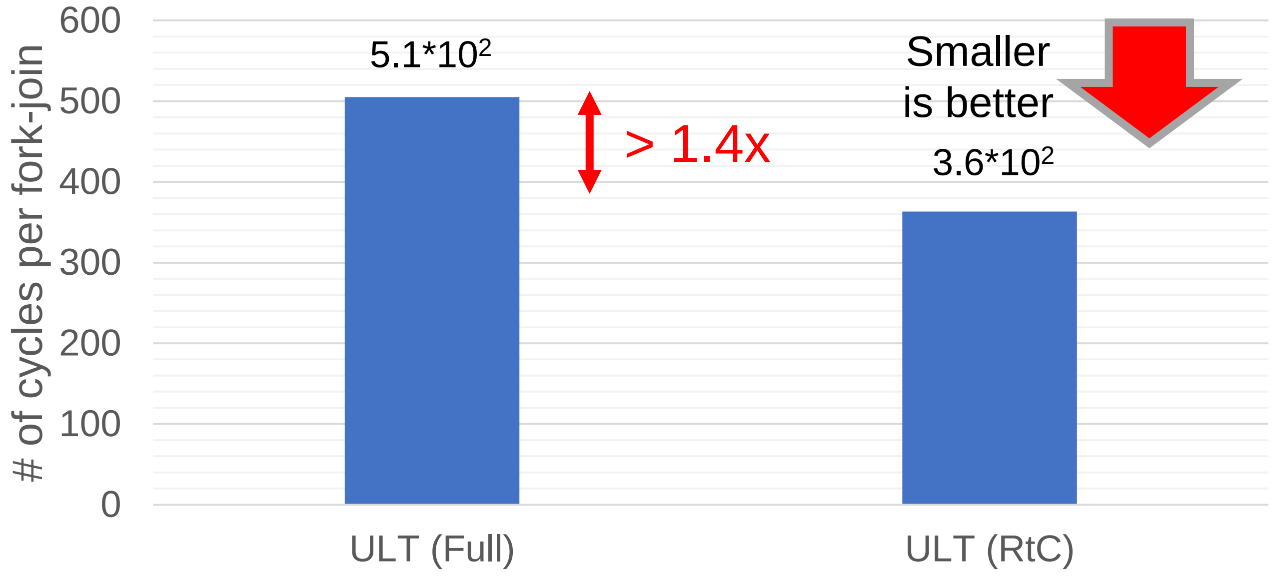
➡ We can create more ULTs.

- Dynamic load balancing (e.g., irregular parallelism)
- Latency hiding (I/O & network) (e.g., latency-intensive applications)

We used **Argobots**:
- http://www.argobots.org/
- https://github.com/pmodels/argobots

Full = LC, repeat creating 256 threads & joining all of them; no suspension. / pthread: 500 iterations & 10 warm-ups / Full, RtC: 5000 iterations & 100 warm-ups Average of 10 times execution / Intel Xeon Phi 7210 (KNL) 1 core / 1.3GHz (turbo-boost: off) / Compiler: icc 17.2.174 / Stack size : 16KB / Error < 5% / Red Hat 4.8.5-16 / Huge page is enabled.

Two Opposite ULT Techniques



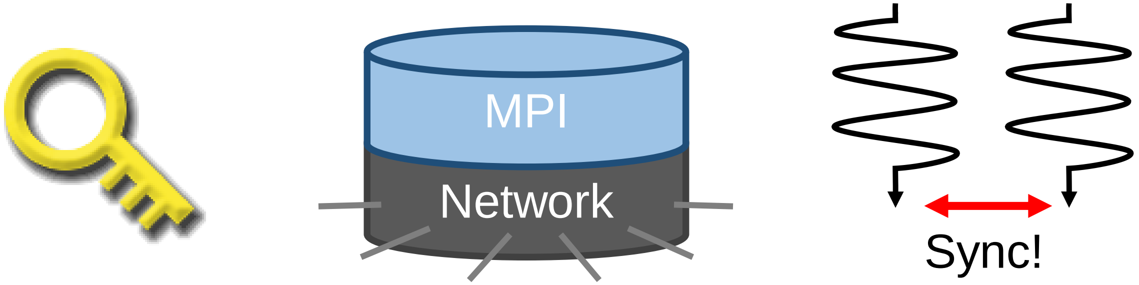
1. Fully-fledged thread (**Full**): fully capable ULTs (i.e., suspendable)
   - **Full** has larger overheads.
     - Adopted by Cilk, CilkPlus, Nanos, Qthreads, MassiveThreads, Argobots, …
2. Run-to-completion thread (**RtC**): ultimately lightweight ULTs
   - **RtC** cannot suspend.
     - Adopted by Filaments, Qthreads, Intel TBB, Argobots, …

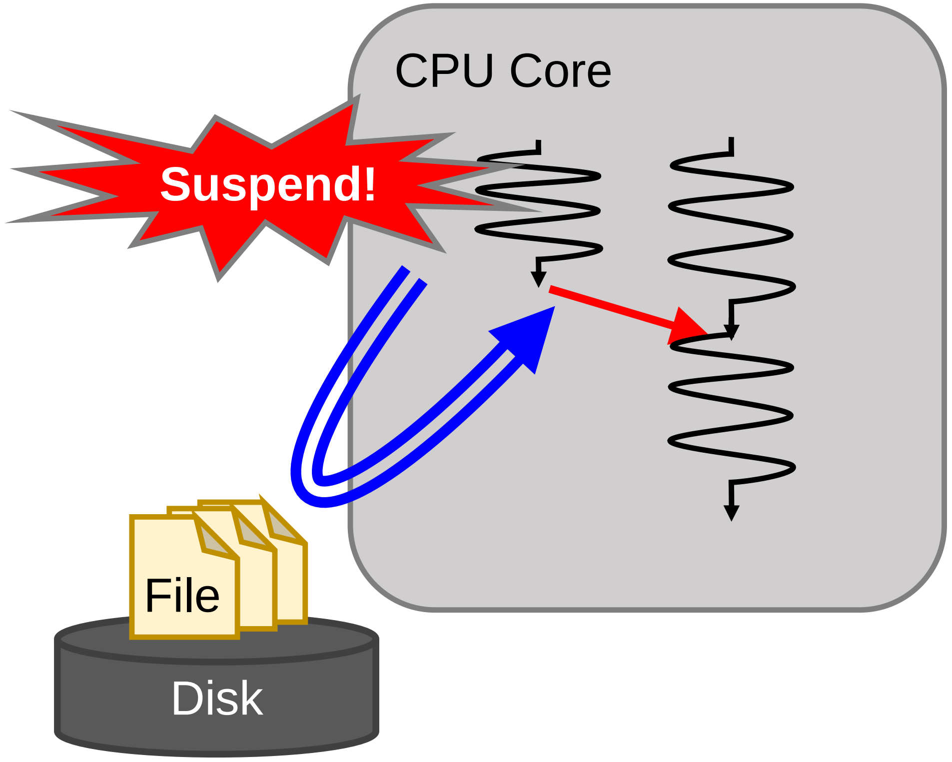
Suspension: Use Cases

- Suspension: save the thread context, and switch to another thread (similar to `pthread_yield()`)

Full can while **RtC** cannot.

- Suspension is used to efficiently utilize compute resources.
  1. Waiting for a lock (mutex, critical section).
  2. Waiting for I/O or communication.
  3. Waiting for completion of other threads

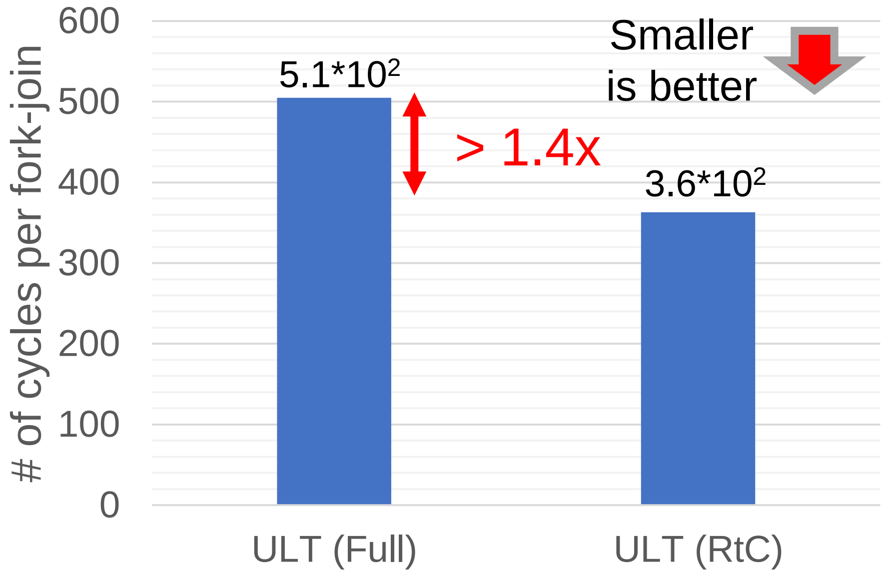




Costs of Suspension Capability

- If a ULT never suspends,
  RtC is faster than **Full**.
  - **Full** has additional threading overheads on fork/join to prepare context switching.

Describe later.



- Suspension demand is application-dependent.
- Case: very few ULTs suspend (e.g., low resource contentions)

Between **Full** and **RtC**: Dynamic Promotion

- Our work investigates a ULT which is
  - as fast as **RtC** if it does not suspend, but
  - able to suspend as well as **Full**
- Key idea: dynamic promotion from **RtC** to **Full**.
  - All of them are applicable to building a threading library.
- Our contributions:

  Most previous work evaluated whole packages, not the individual methods.

  - In-depth analysis of full spectrum of user-level threading techniques.
  - Two new techniques that do not exist in a past literature.

Index

1. Introduction : Lightweight threads
2. Background : How ULTs work
3. Analysis & Proposals
4. Evaluation
5. Conclusions

Quick Overview

	Change stack?	# of context switches (nonsuspension)	Performance (nonsuspension)	Constraints	Performance (suspension)
Full	Yes (Eager)	2	Slow	---	Fast
LSA	Yes (Lazy)	2		---	
RoC	Yes (Lazy)	1		-	
SS	Yes (Lazy)	0		Scheduler must be stateless.	
SC	No	0		Scheduler must be stateless. Stack size is shared.	Slow
					(Cannot Suspend)

New (RoC)

New (SS)

TRADE-OFF

Faster!

Stricter constraints!

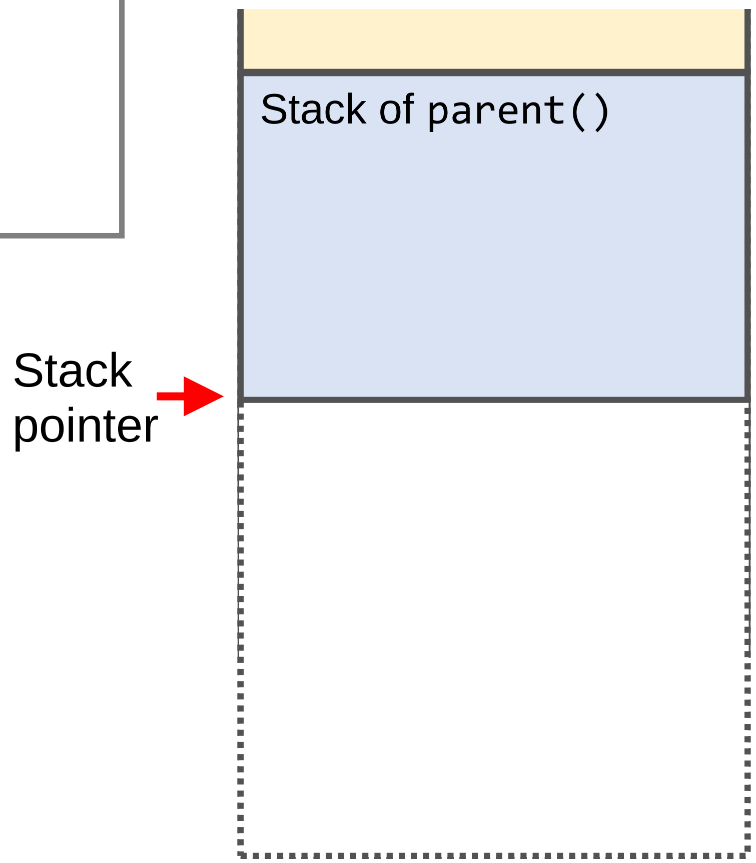
https://www.utep.edu/extendeduniversity/utepconnect/blog/march-2017/4-ways-to-differentiate-a-good-source-from-a-bad-source.html

Flow of Function Call

```
void parent() {
  ...
  child();
  ...
}
```

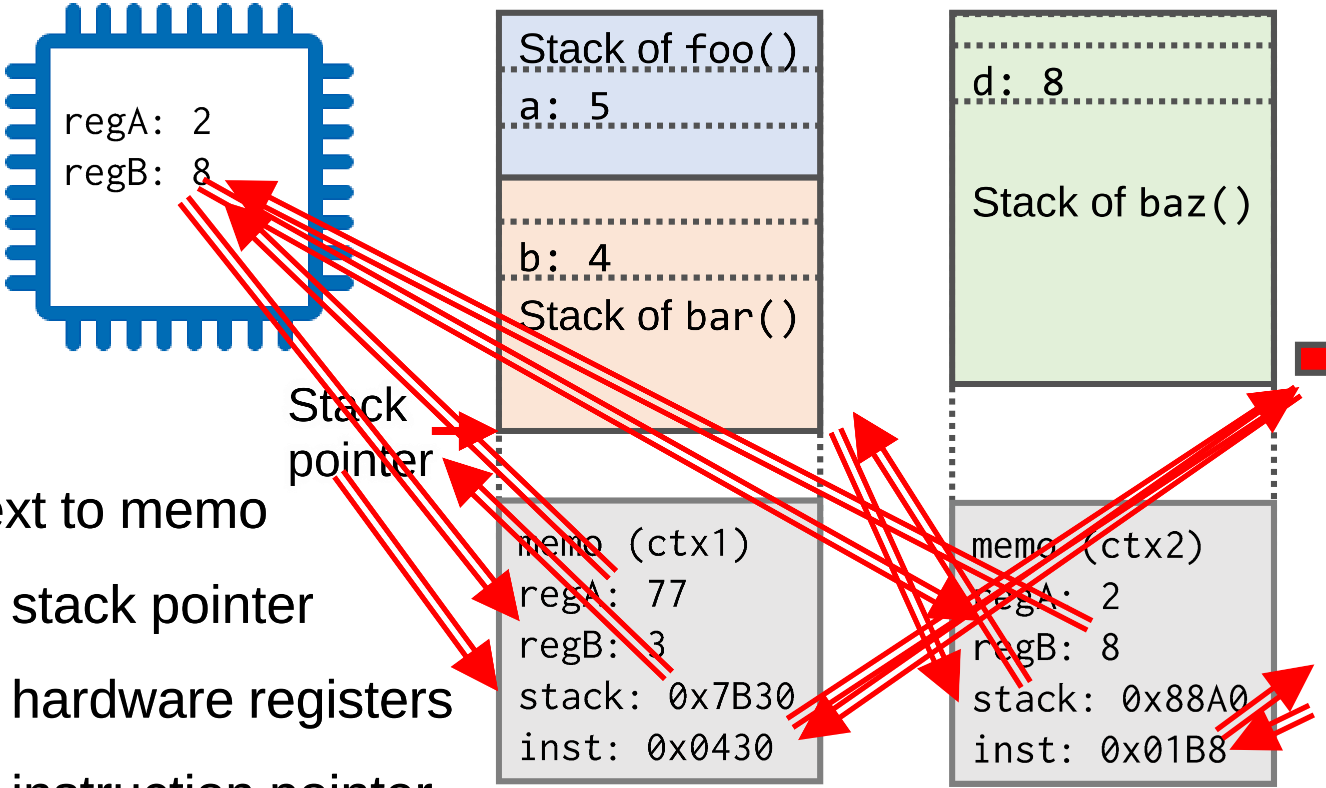
```
void child() {
  [push registers.]
  [...];
  [pop registers.]
}
```

1. P(): call to child()
2. C(): push registers
3. C(): run a body of a function
4. C(): pop registers
5. C(): return to parent()



(Naïve) Function Context: Stack & Registers

- Function context = execution state of a function.
- Composed of register values and a function stack.

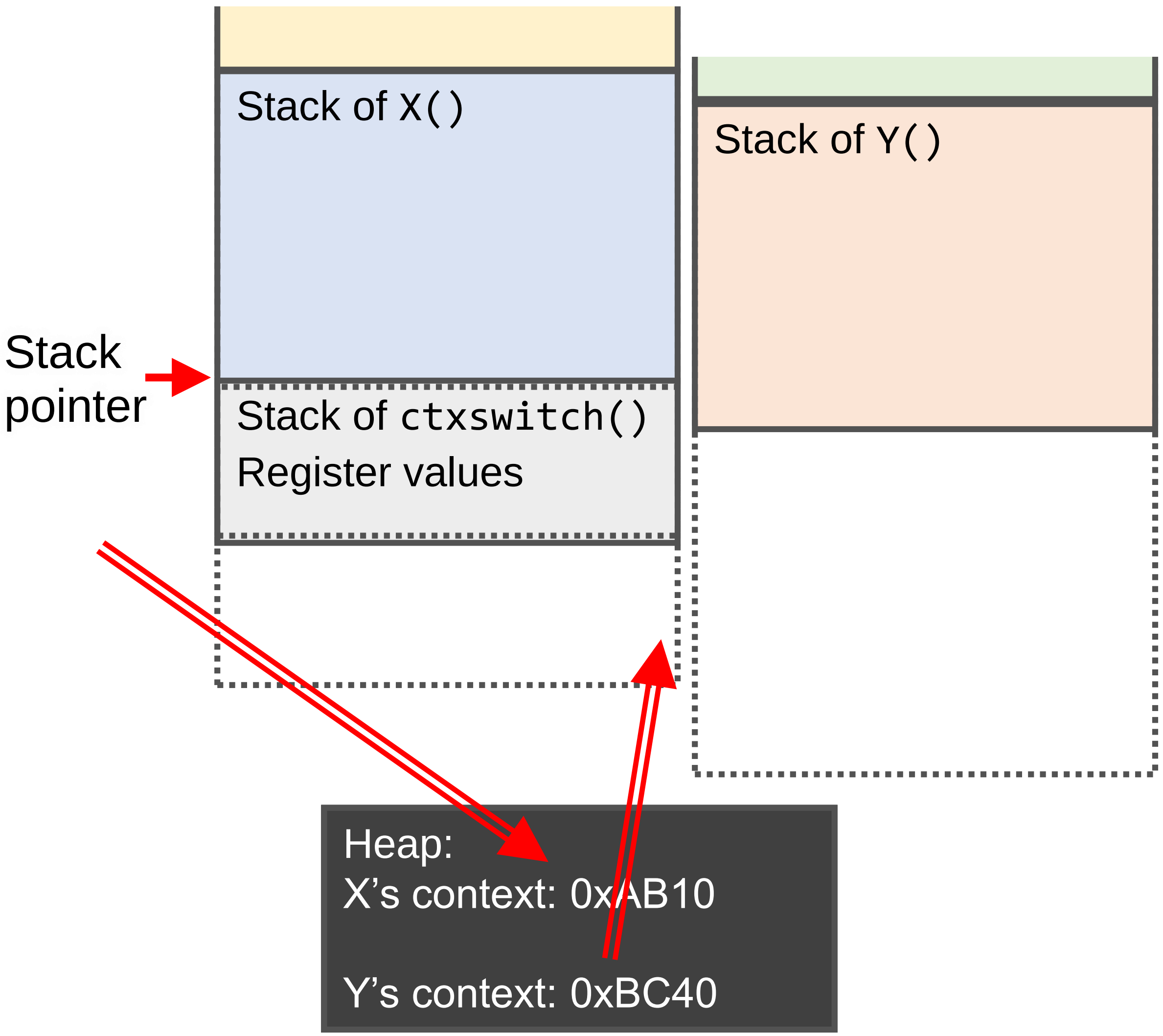


1. Write context to memo
2. Update the stack pointer
3. Update the hardware registers
4. Update the instruction pointer

```
void foo() {
  int a = 5; // in stack
  bar();
  [...];
}
void bar() {
  int b = 4; // in stack
  int c = 3; // in regB
  context_switch(&ctx2);
  [...];
}
void baz() {
  int d = 8; // in stack
  int e = 2; // in regA
  context_switch(&ctx1);
  context_switch(&ctx1);
  [...];
}
```

User-level Context Switch

- Switch from X() to Y()
  1. `X()`: call `ctxswitch()`
  2. `X()`: push registers
  3. `X()`: save a `X()`'s stack pointer
  4. `X()`: set a `Y()`'s pointer
  5. `Y()`: pop registers
  6. `Y()`: jump to a return address

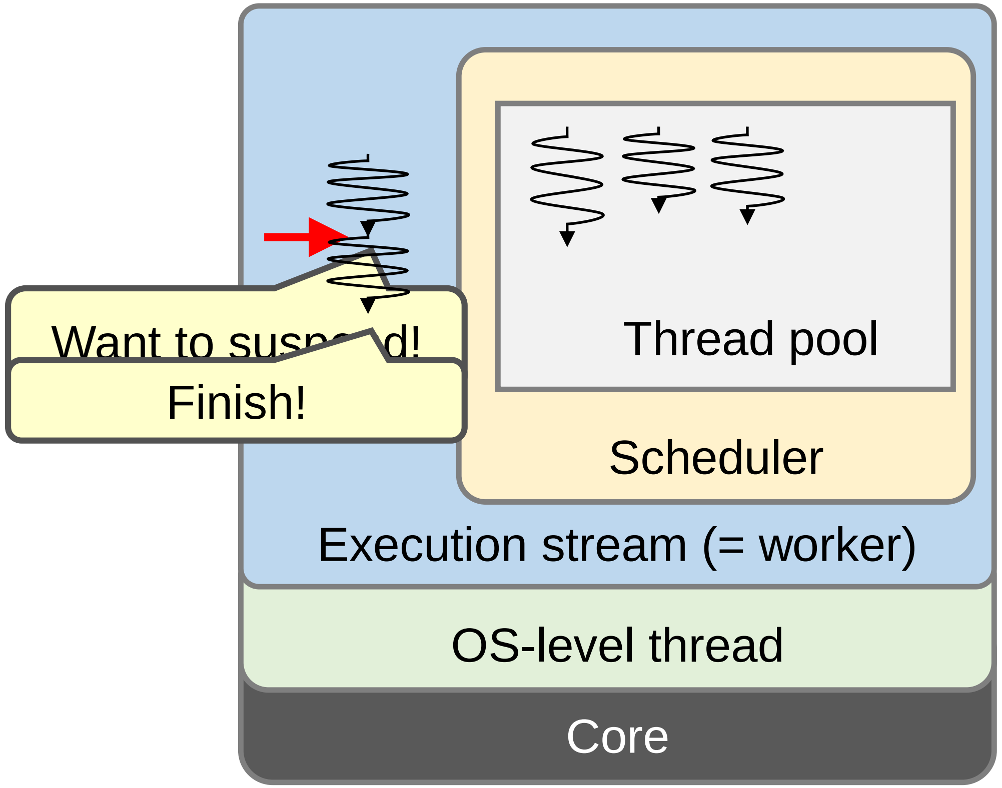


Parent-first

Execution Model of ULTs

- An execution stream (= a worker) is bound to a core.
- A scheduler is running on an execution stream.
  - The scheduler has a loop to execute ULT in the pools.

```
void scheduler() {
  while (1) {
    [get ULT from pool(s)]
    resume ULT;
    if (!ULT.finished)
      [add ULT to a pool]
  }
}
```



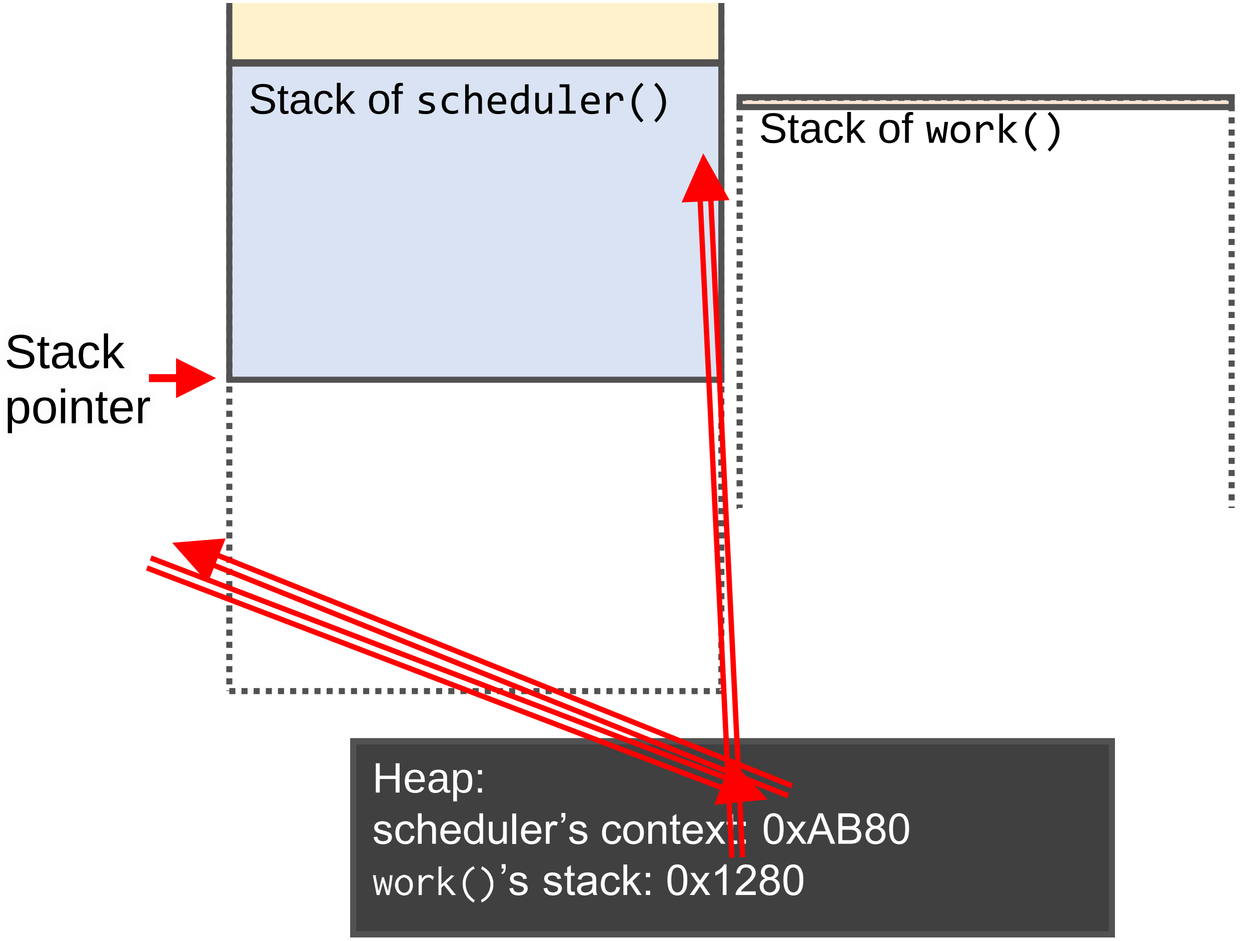
Full : Nonsuspension Case

"call"

1. `S()`: call `ctxswitch()`
2. `S()`: push registers
3. `S()`: save a `scheduler()`'s stack pointer
4. `S()`: set a `work()`'s stack pointer
5. `S()`: call `work()`
6. `T()`: [… run a function body …]

"return"

7. `T()`: restore an `scheduler()`'s stack pointer
8. `S()`: pop registers
9. `S()`: jump to a return address



RtC : Nonsuspension Case

- Ultimately, **RtC** is a function pointer and its argument.
  - Schedulers can just call it

```
void scheduler() {
  while (1) {
    [take ULT from pool(s)]
    resume ULT;
    if (!ULT.finished)
      [add ULT to a pool]
  }
}
```

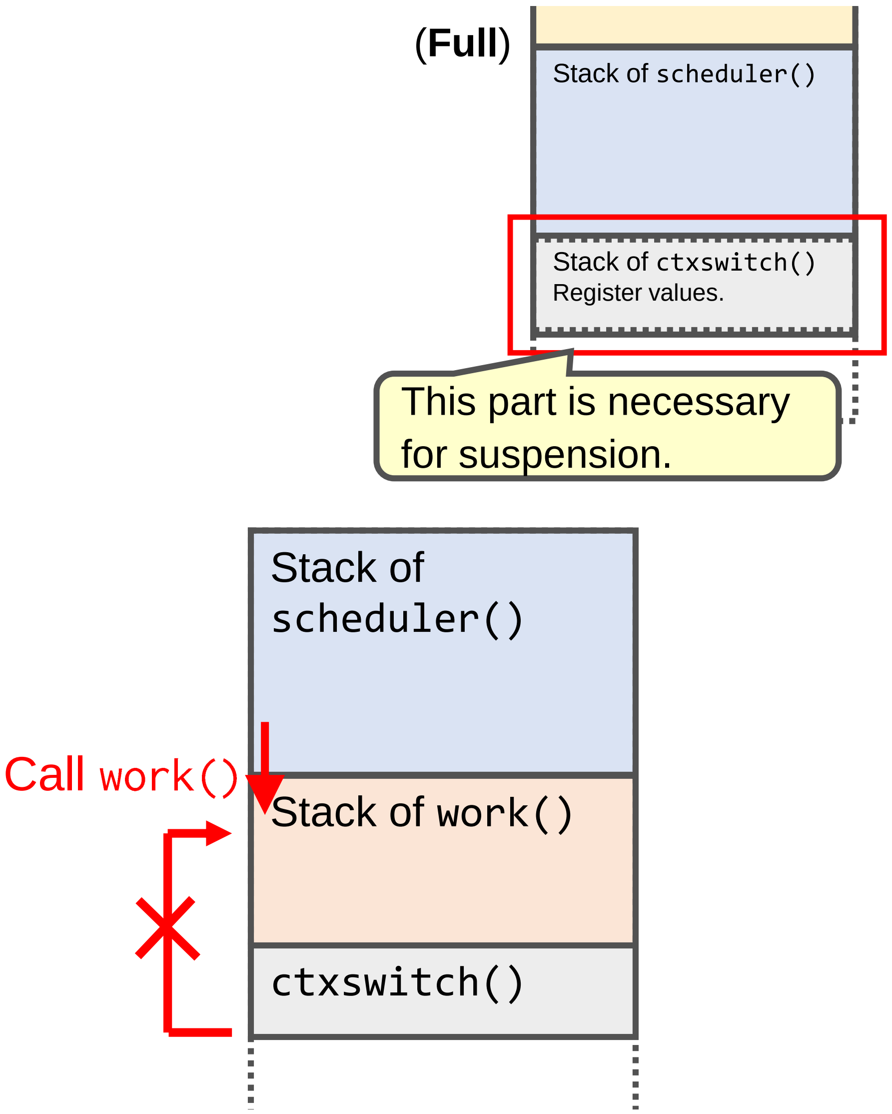
```
void scheduler() {
  while (1) {
    [take ULT from pool(s)]
    call ULT.work(ULT.arg);
  }
}
```

RtC never suspends.

RtC Can't Suspend

- Because registers, a stack pointer, and an instruction pointer are unsaved, we cannot resume `scheduler()`.

```
void scheduler() {
  while (1) {
    [take ULT from pool(s)]
    call work(arg);
  }
}
```

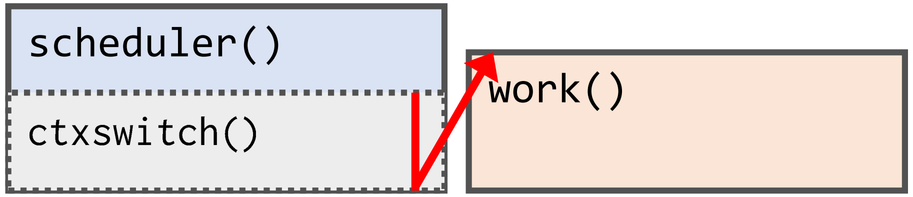


When ULTs do not suspend

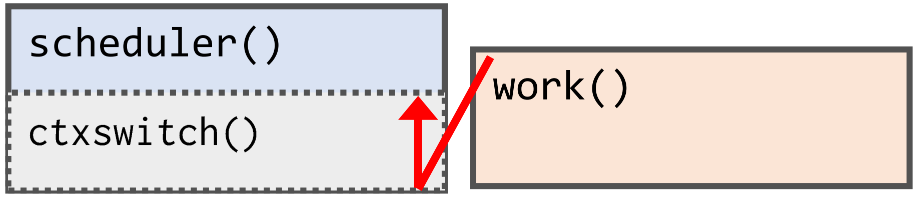
[Summary] Costs: **Full** vs. **RtC**

- **RtC** : 1 function call + scheduling
  - Scheduling = thread pool operations + descriptor management … etc.
- **Full :** 1 function call + scheduling
  + 2 user-level context switches + stack management

1. When a ULT starts
2. When a ULT finishes.



1st context switch (invoke a ULT)



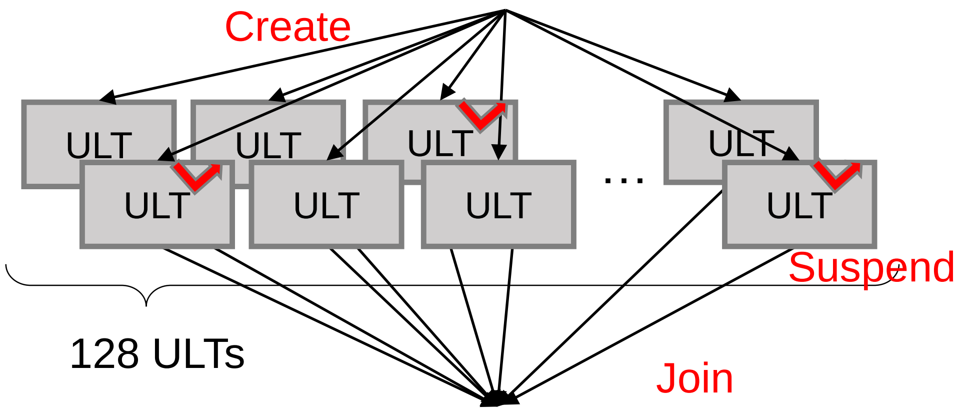
2nd context switch (resume scheduler)

Index

1. Introduction : Lightweight threads
2. Background : How ULTs work
3. Analysis & Proposals
4. Evaluation
5. Conclusions

Microbenchmark: fork-join+suspend

- Analysis is based on a fork-join + yield benchmark:
  - Create and join 128 threads
    - S % of 128 ULTs suspend once
    - We run it on Intel Xeon E5-2699 v3.

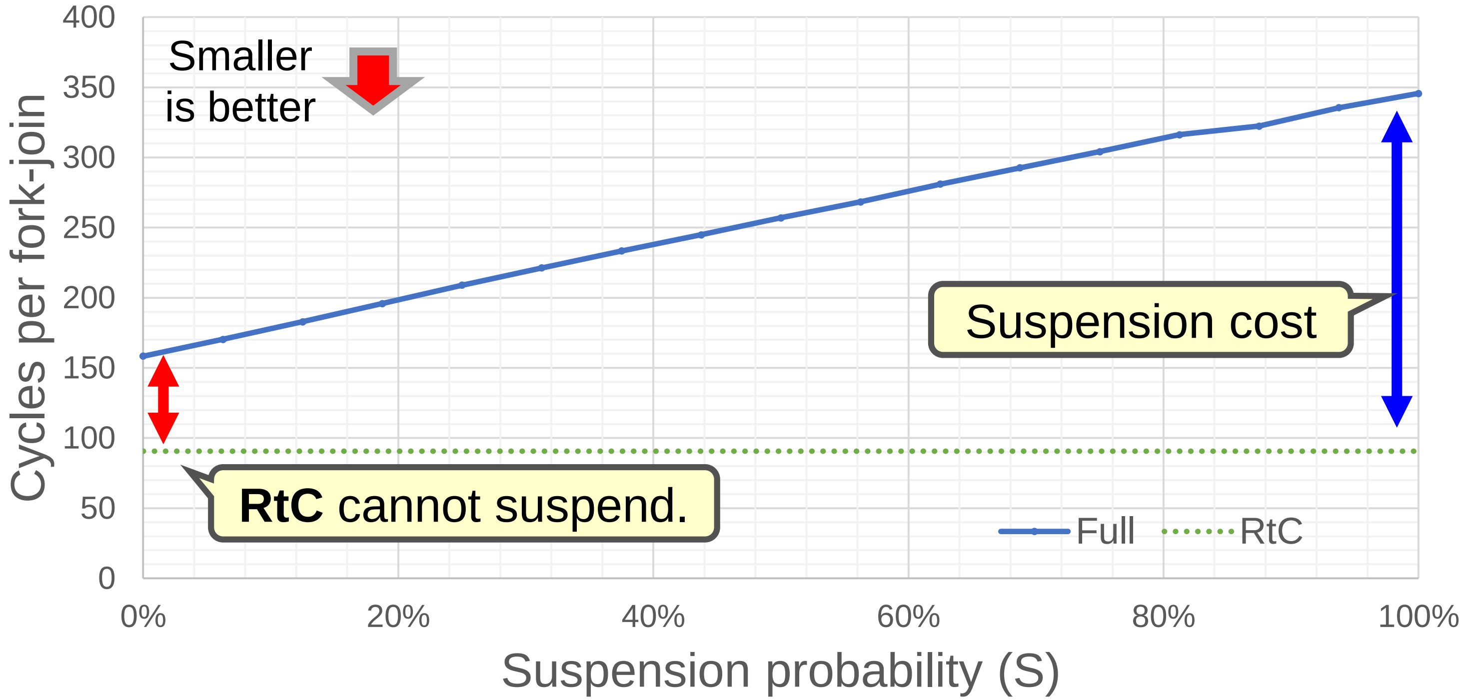



```
void body(void* arg) {
  if ((intptr_t)arg == 1)
    suspend();
}
```

```
HANDLE ts[128]
for (int i = 0; i < 128; i++)
  create(body, suspend_flags[i], &ts[i]);
for (int i = 0; i < 128; i++)
  join(ts[i]);
```

- Show dynamic promotion techniques from **Full**
  - Focus on the performance when threads do not suspend.

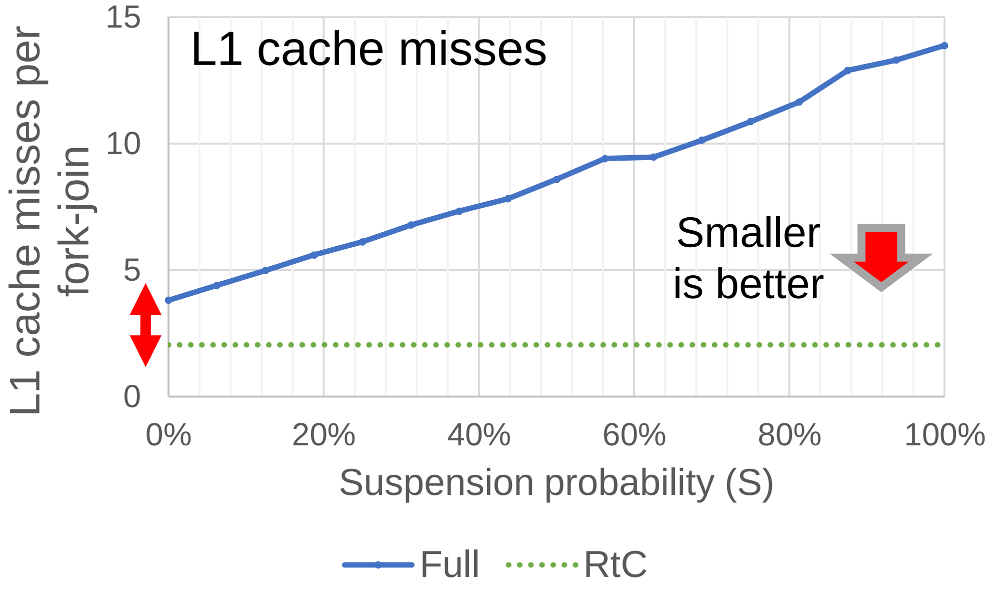
From **Full** to **RtC**



- Suspension probability (= S) = $\frac{\text{\# of threads that suspend}}{\text{total \# of threads}}$
- Narrow the performance gap at S = 0%

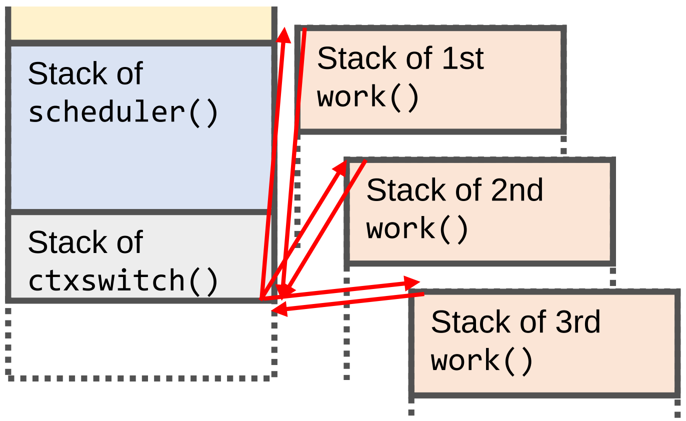
Costs of Fully Fledged ULTs (**Full**)

- **Full:** more cache misses because all ULTs use different function stacks.
  - Stacks are allocated when **Full** is created.

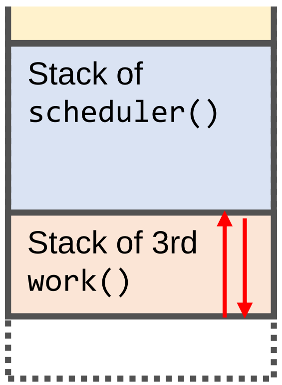


- **RtC:** small cache misses because they use the same function stack.
  - The scheduler's stack is reused.

Full:

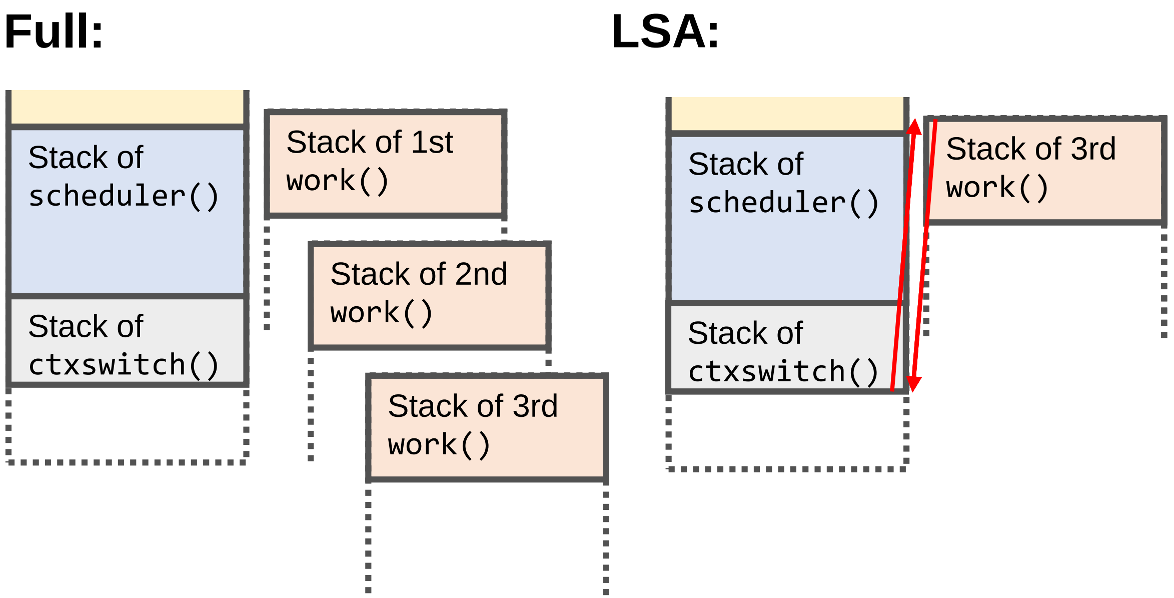


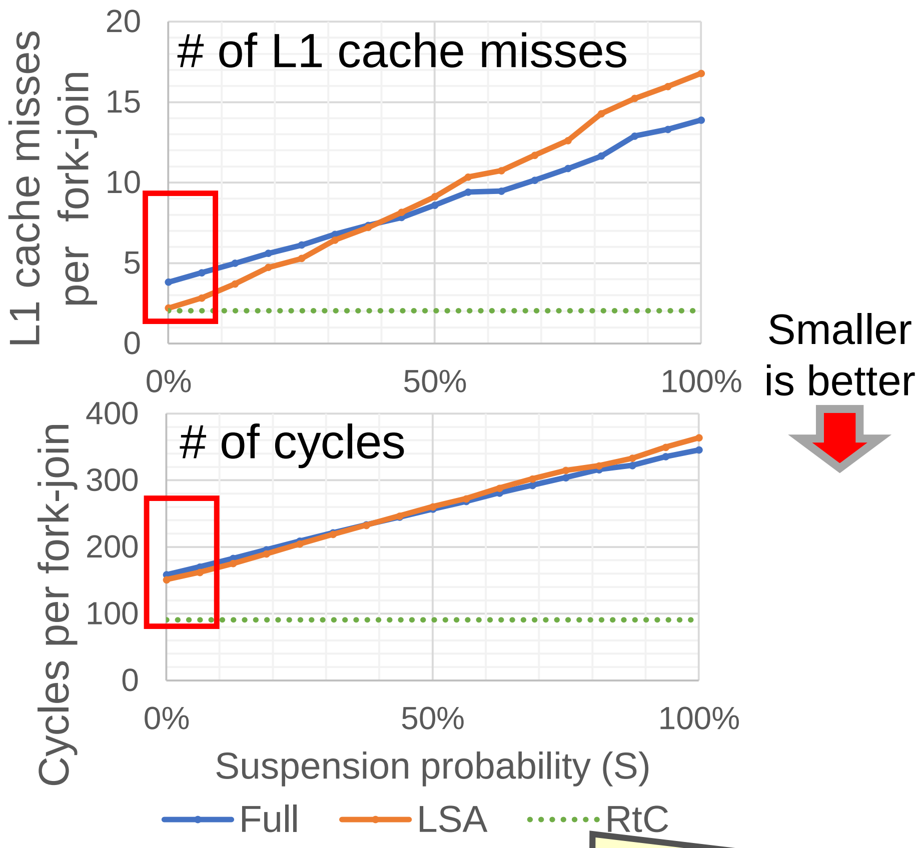
RtC:



Lazy Stack Allocation (**LSA**)

- Lazy stack allocation (**LSA**): allocates stacks when ULTs are invoked, not created.
  - If a ULT did not suspend, the next ULT uses the same stack.

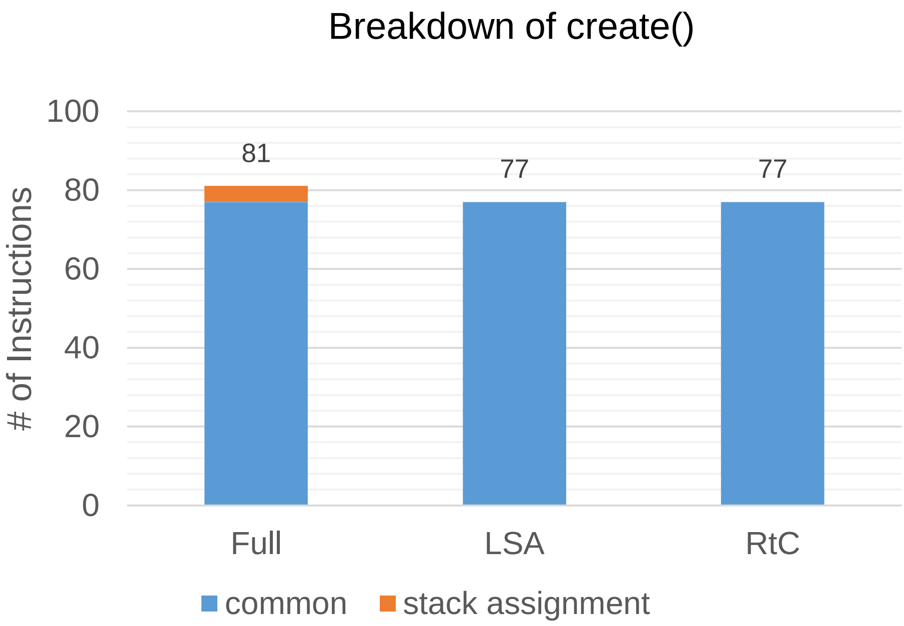


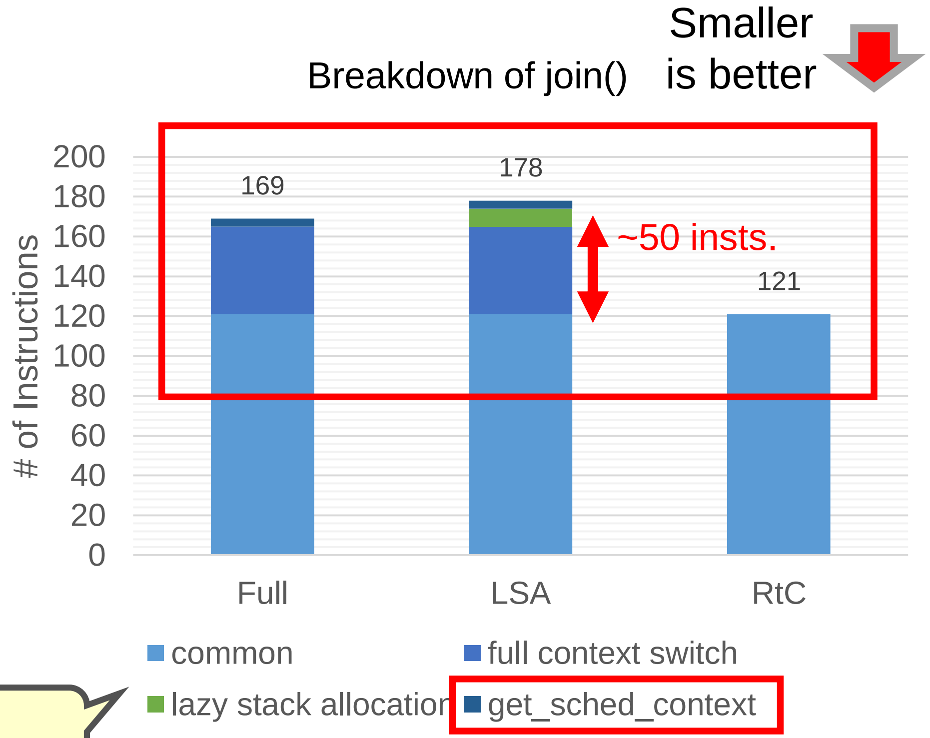


Full allocates a thread descriptor and stack at once, while **LSA** does separately. It degrades **LSA**'s performance when the suspension probability is high.

Costs of **LSA** : Two Context Switches

- Compared to **RtC**, # of instructions is quite large.
  - Costly part: user-level context switches (=stack and register manipulation)





Common includes overheads of schedulers, thread pool operations, and memory management of thread descriptors.

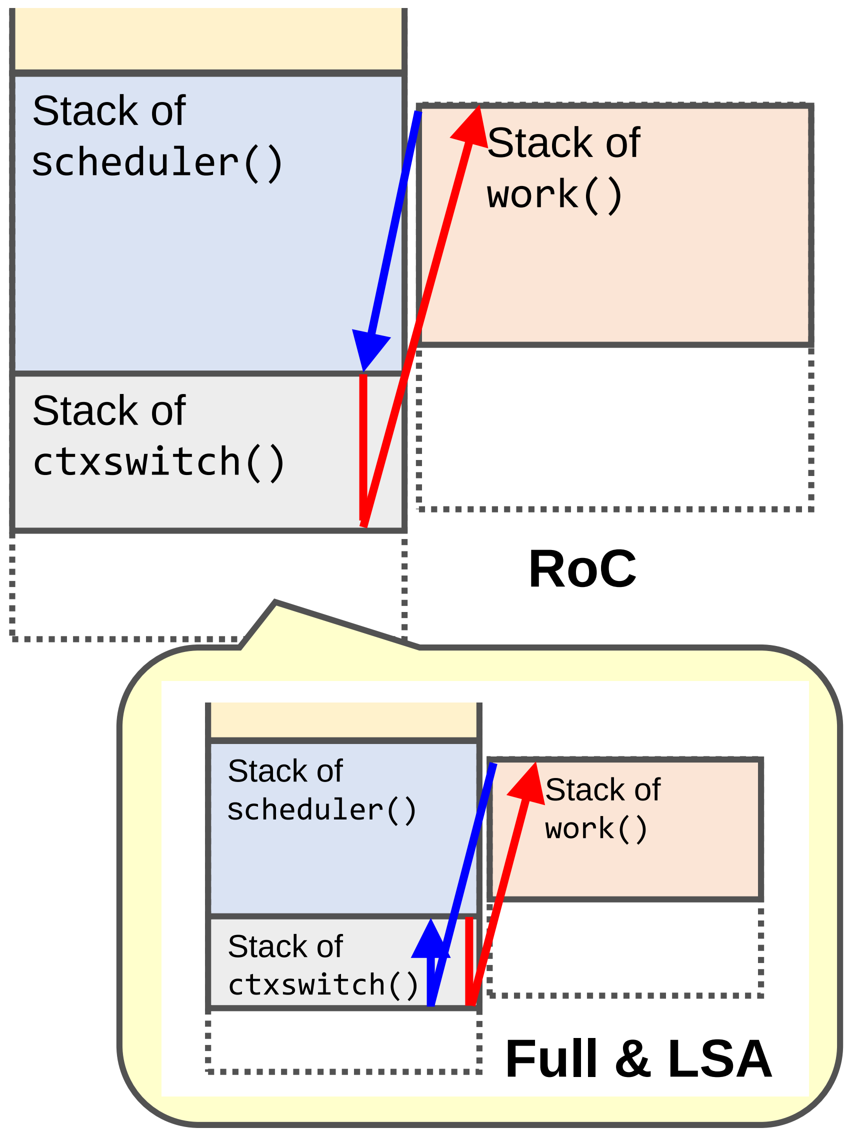
Return-on-Completion (**RoC**)

- The first context switch is necessary to save the scheduler's context.
  - Needed for the future resume.
- The second context switch can be replaced by return if it just jumps to the parent if the ULT never suspends.
  - An assembly-level trick enables it.

(*) In general, a caller cannot be resumed by "return" because user-level context switch does not follow a standard ABI.

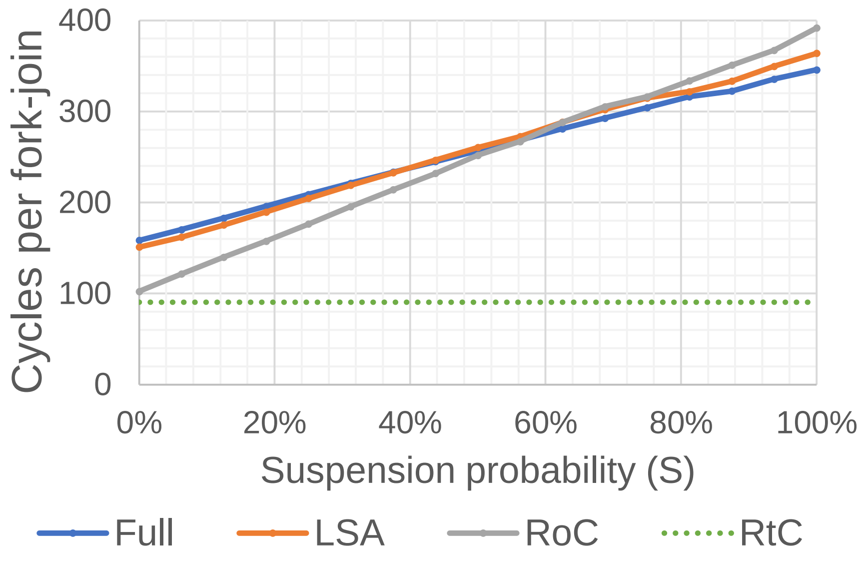
  - If the ULT suspends, `ctxswitch()` is called at the end of `work()`.

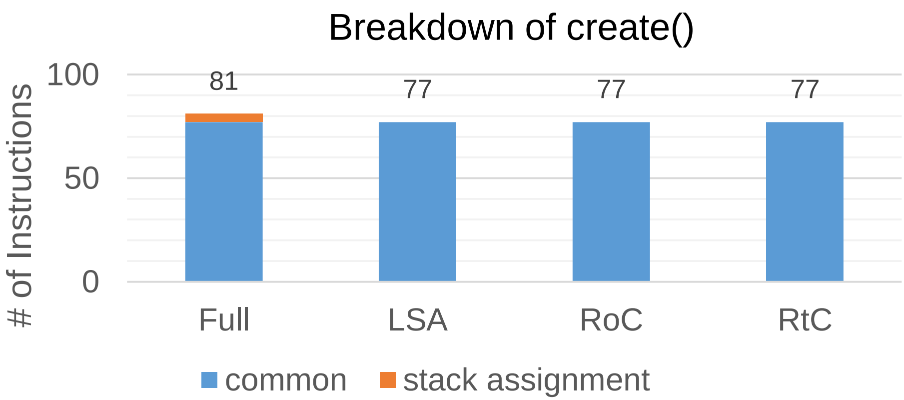
➡ *Return-on-completion* (**RoC**)

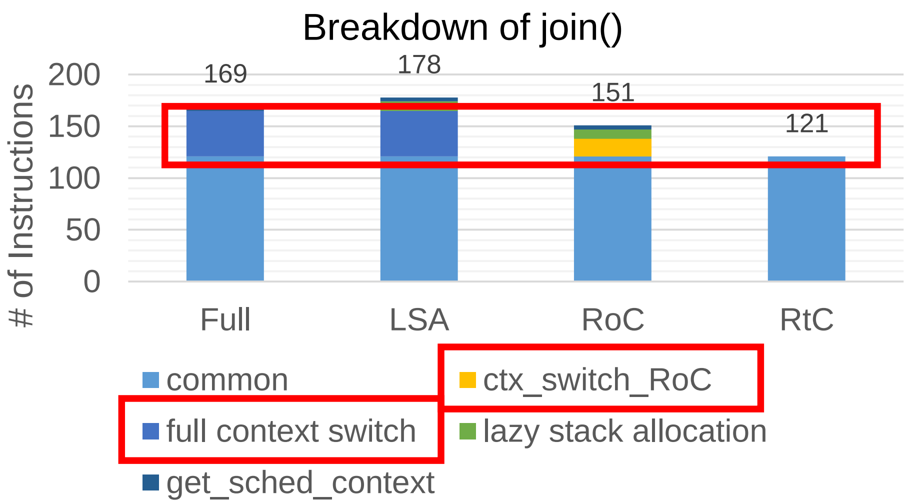


RoC: Performance

- **RoC** successfully reduces # of instructions.
  - Good performance when the suspension probability is low.



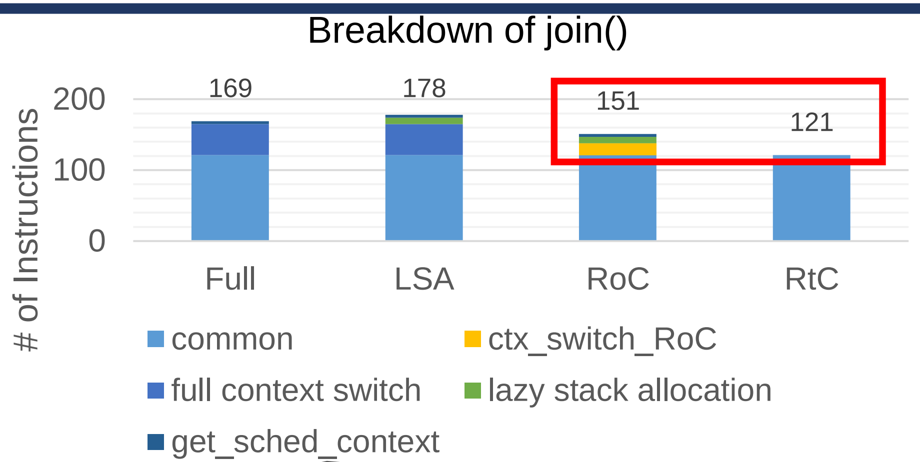




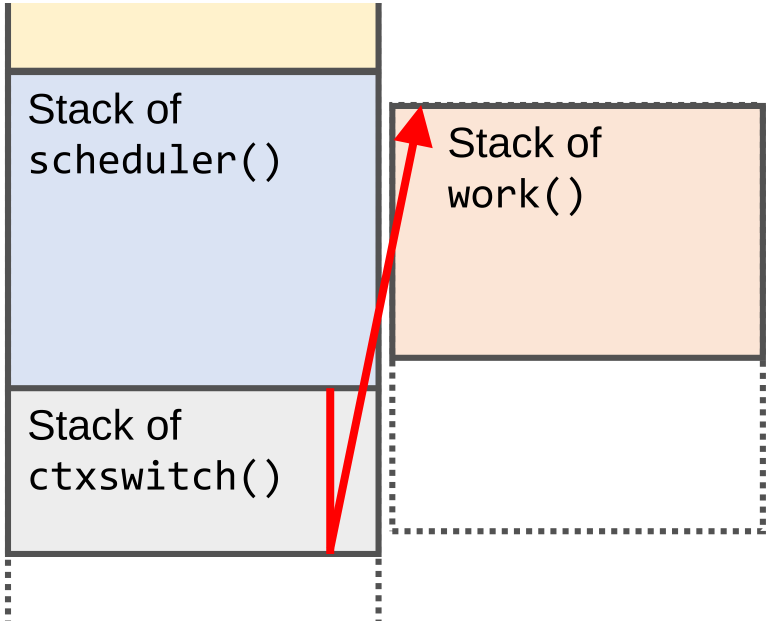
Costs of **RoC** : One Context Switch

- Compared to **RtC**, # of instructions of **RoC** is still large.
  - Caused by the first user-level context switch and the stack management.
  - They are necessary to resume a parent ULT.

- What if we can restart a scheduler instead of resuming it?

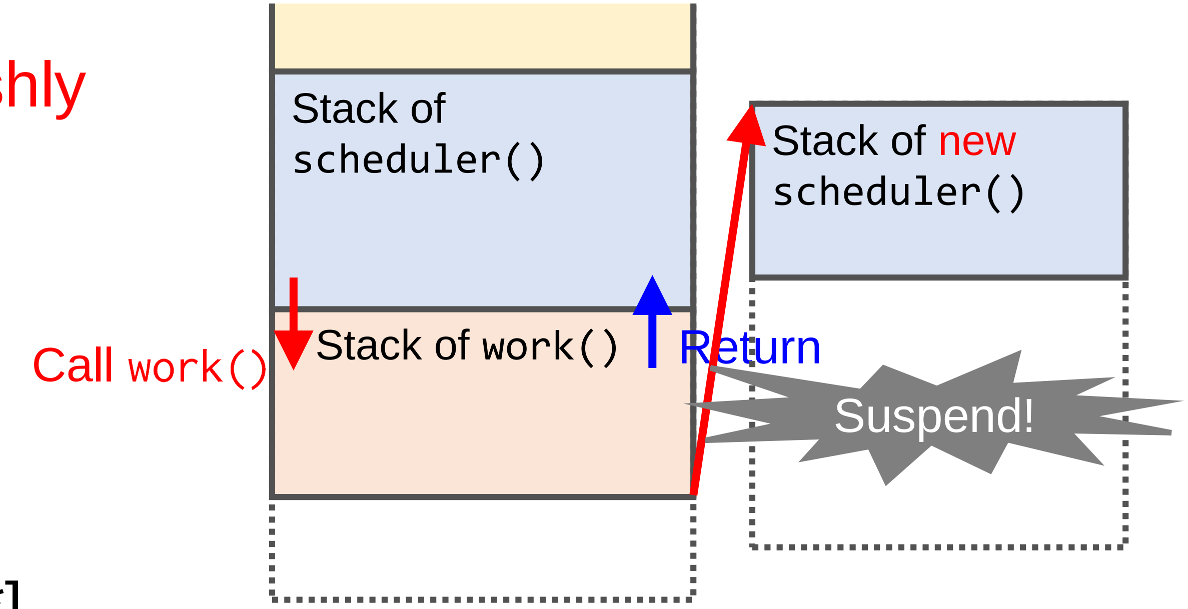


`ctx_switch_RoC` includes one context switch.



Scheduler Creation (**SC**)

- Assume schedulers are running on ULTs.
- If the scheduler is *stateless*, we can freshly start a scheduler on the new ULT.
  - The context of the original scheduler is abandoned.
- It has been previously proposed [*] - [***].
- Let's call *scheduler creation* (**SC**).

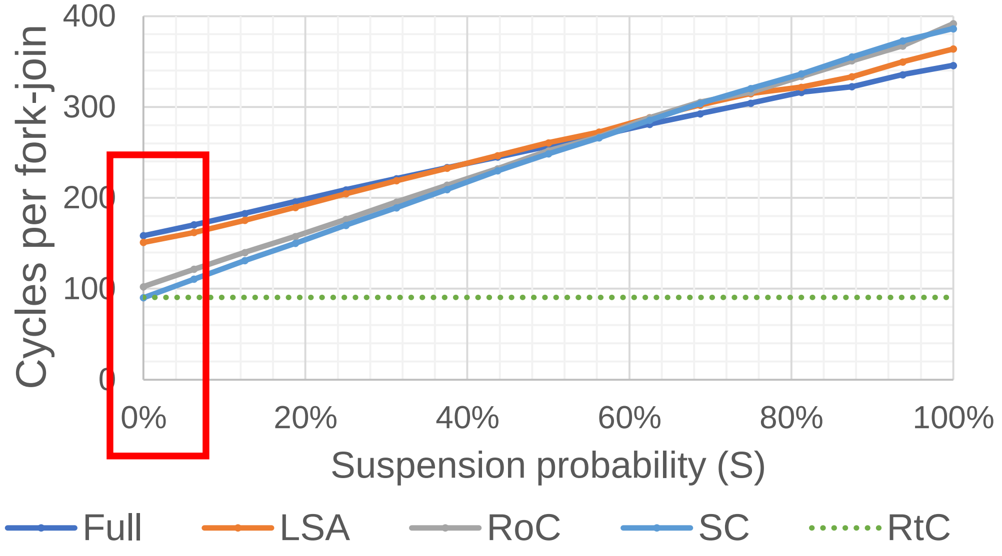


It has almost the same execution flow of **RtC.**

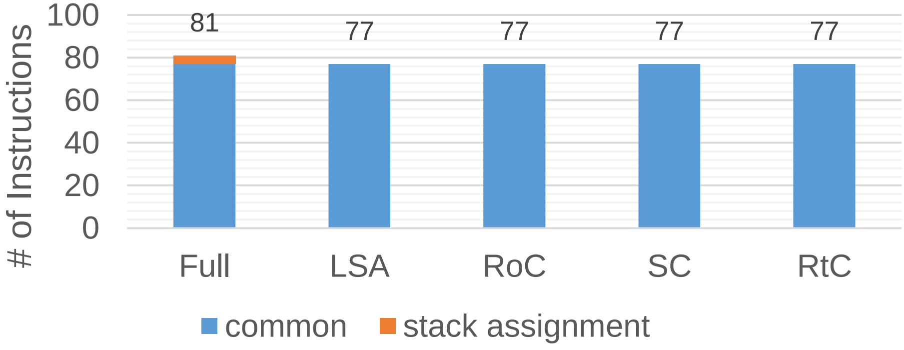
[*] D. L. Eager and J. Jahorjan. Chores: Enhanced run-time support for shared-memory parallel computing. TOCS. 1993
[**] K.-F. Faxén. Wool - A work stealing library. SIGARCH Comput. Archit. News, 2009.
[***] C. S. Zakian, T. A. Zakian, A. Kulkarni, B. Chamith, and R. R. Newton. Concurrent Cilk: Lazy promotion from tasks to threads in C/C++. LCPC '15, 2016

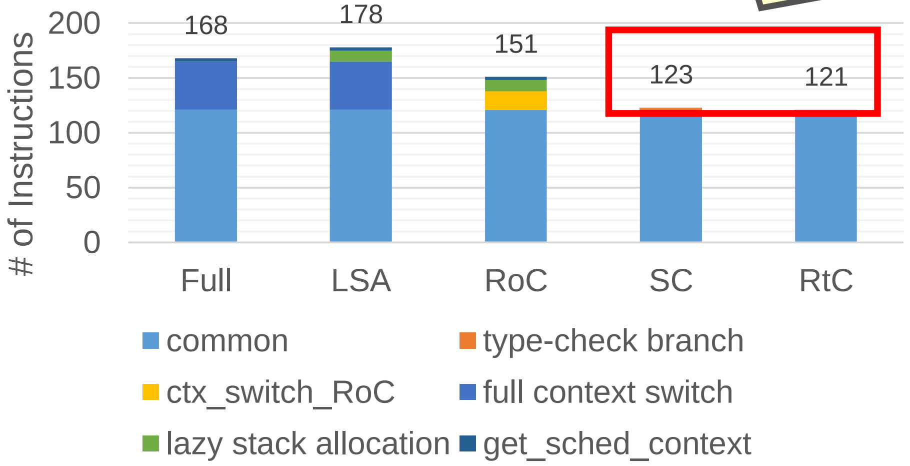
Performance of **SC**

- SC performs as well as RtC when S = 0%.



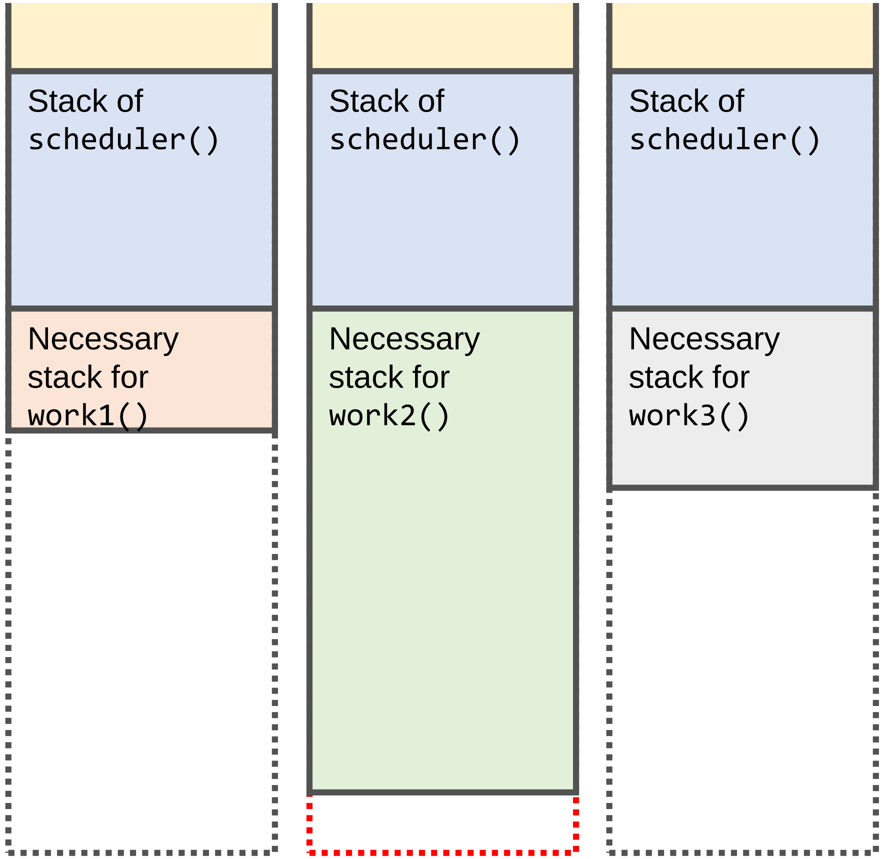
Difference is only 2 instructions!





Constraints of **SC**

1. The scheduler must be stateless.
2. Stack size of schedulers and ULTs must be shared.
   - e.g., an application has multiple types of work each of which requires different stack size.

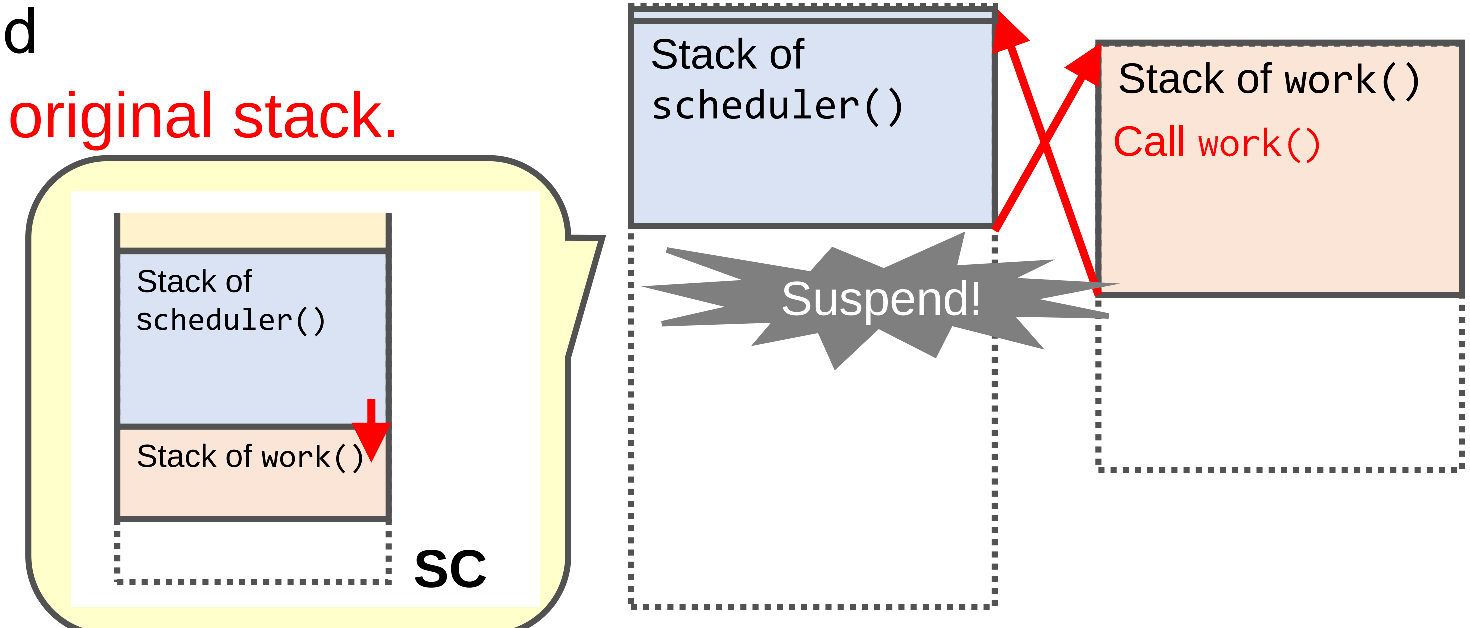


➡ Remove the 2nd constraint by using different stacks.

Individual ULTs cannot specify the size of stacks
➡ Need to use largest size!

Stack Separation (**SS**)

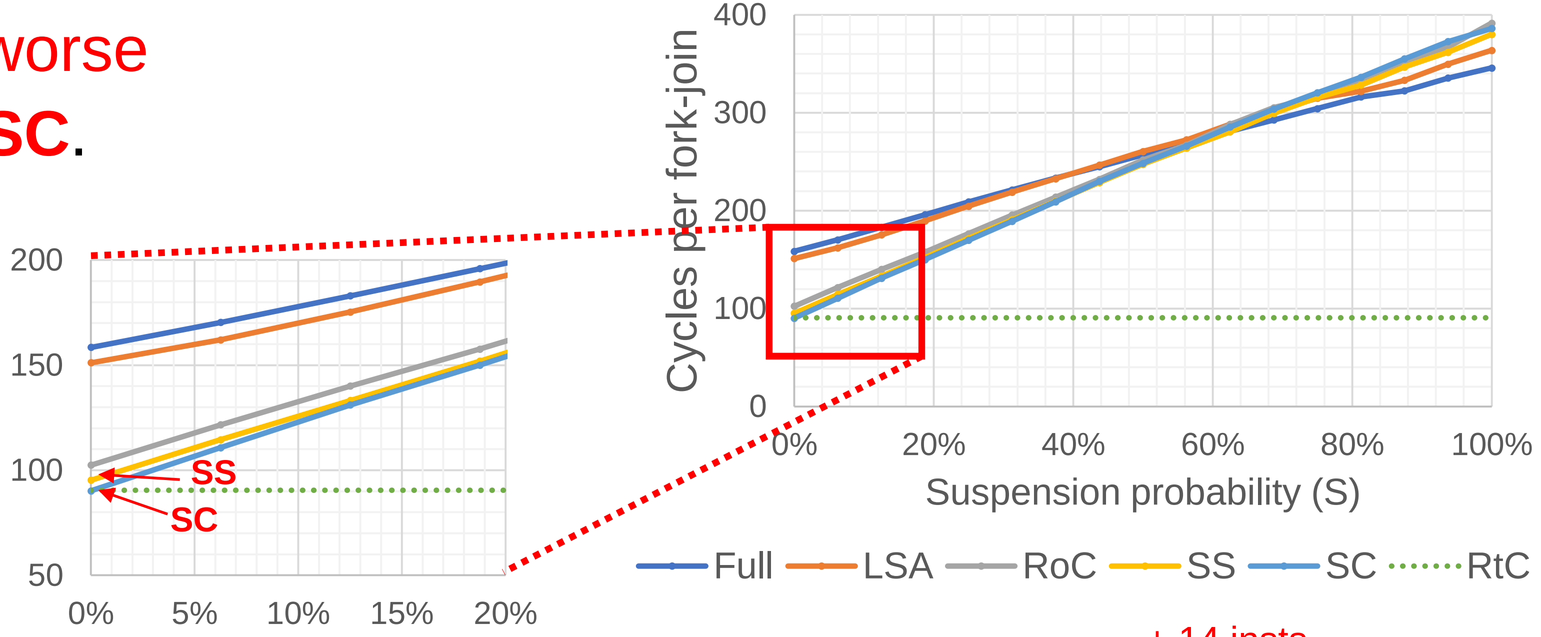
- *Stack separation* (**SS**): it does not save register values of the scheduler, but uses different stacks.
  - Because the context of the parent scheduler is not fully saved, the scheduler must be stateless.
- When `work()` suspends, it renews the `scheduler()`'s stack and calls `scheduler()` over the original stack.

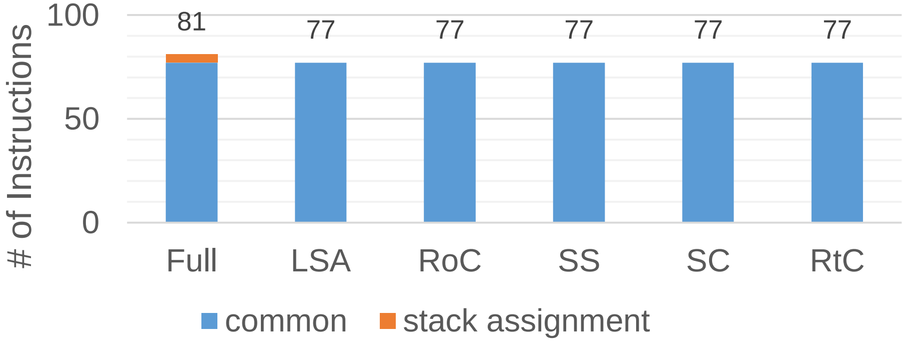


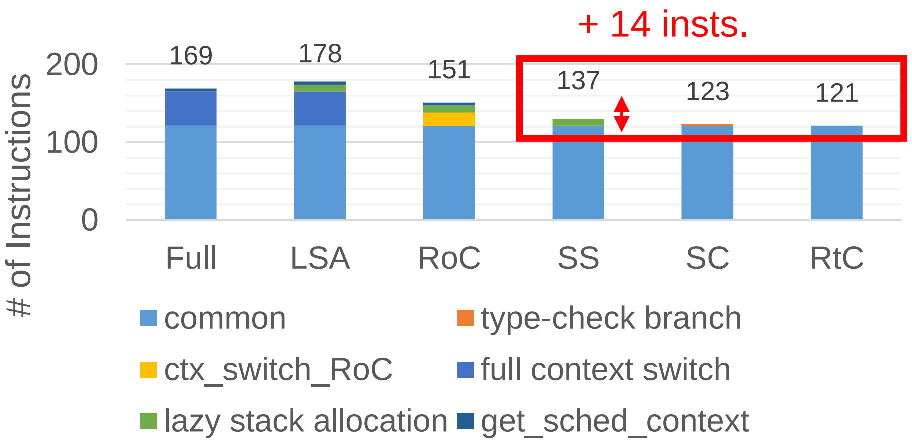
Performance of **SS**

- **SS** shows slightly worse performance than **SC**.

Because of additional instructions!







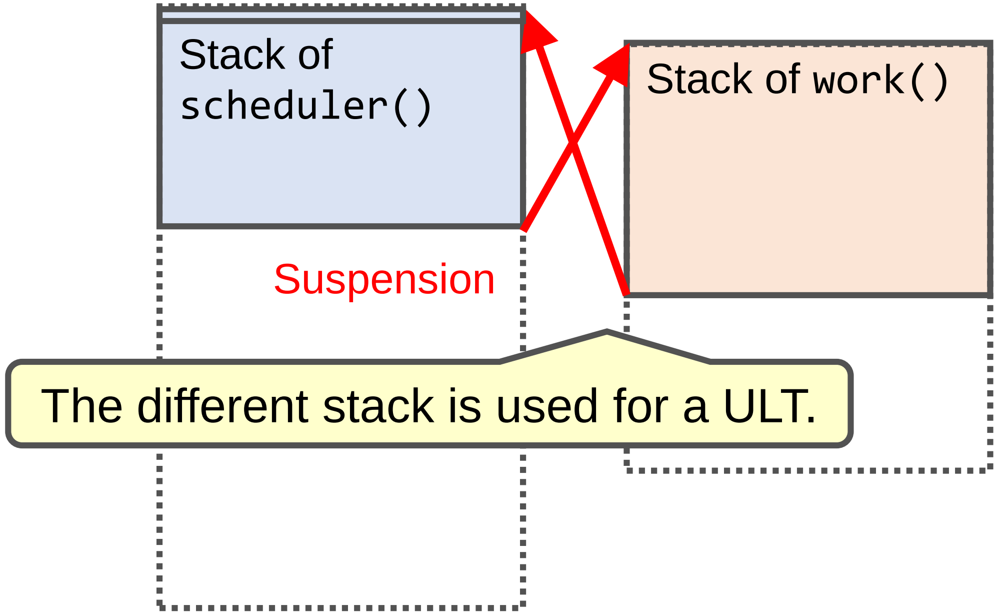
Constraints of **SS**

1. The scheduler (or in general, the parent function) must be stateless.

= 1st constraint of **SC.**

2. ~~Stack size of schedulers and ULTs must be shared.~~

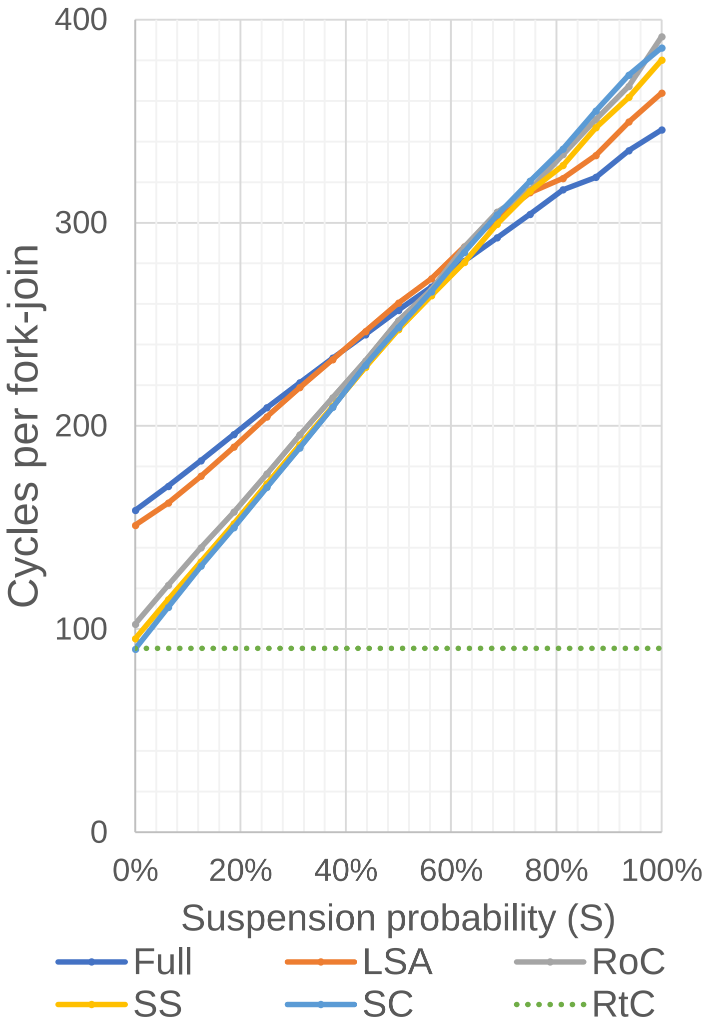
- Stacks are not shared!



Summary

	S=0% Case (No suspension)			S=100% Case		
	Change Stack?	# of ctx switches	Overhead	Rerun sched.?	Overhead	Constraints
Full	Yes	2	High	No	Low	No
LSA	Yes	2		No		No
RoC	Yes	1		No		No
SS	Yes	0		Yes		*
SC	No	0		Yes	High	**
RtC	No	0	Low	-	-	***

* Schedulers must be stateless.
** Schedulers must be stateless. Stack size of schedulers and ULTs is shared.
*** Threads are unable to yield.



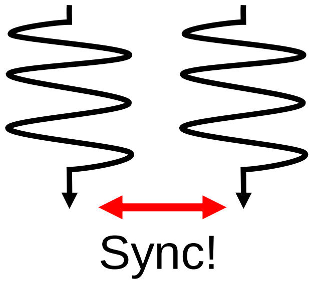
- Typical trade-off relationship.
    - Performance at S=0% and performance at S=100%.
- **SS**, **SC**, and **RtC have** additional constraints.

Index

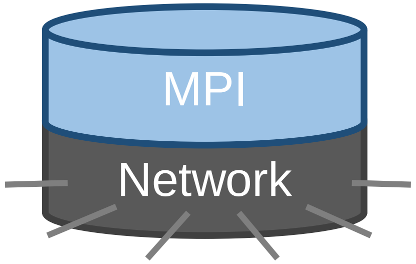
Three Motivating Cases

1. Waiting for mutexes.
   - KMeans: simple machine learning algorithm.
     ULTs access shared arrays with locks.

2. Waiting for completion of other threads
   - ExaFMM: divide-and-conquer O(N) N-Body solver.
     Parent ULTs need to wait for children.

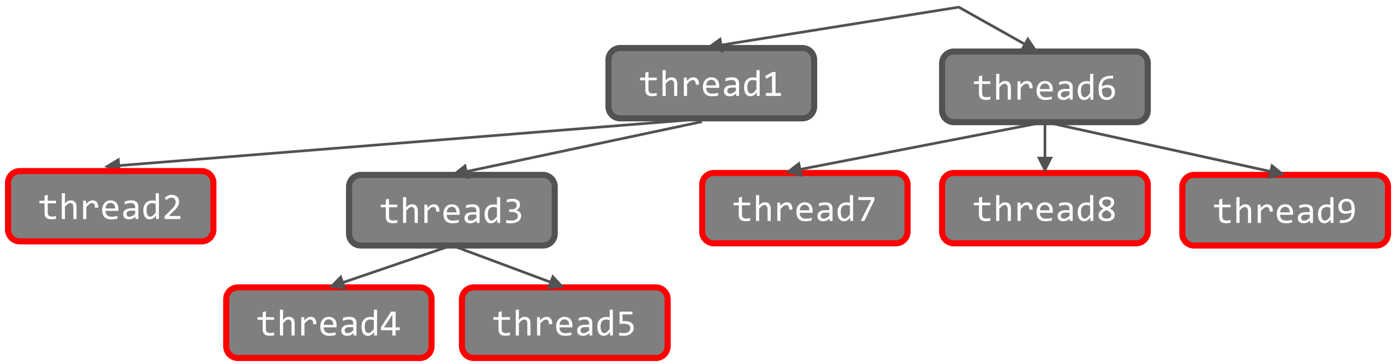


3. Waiting for communication.
   - Graph500: fine-grained MPI program
     ULTs conditionally call MPI functions.



1. ExaFMM: Recursive Parallelism

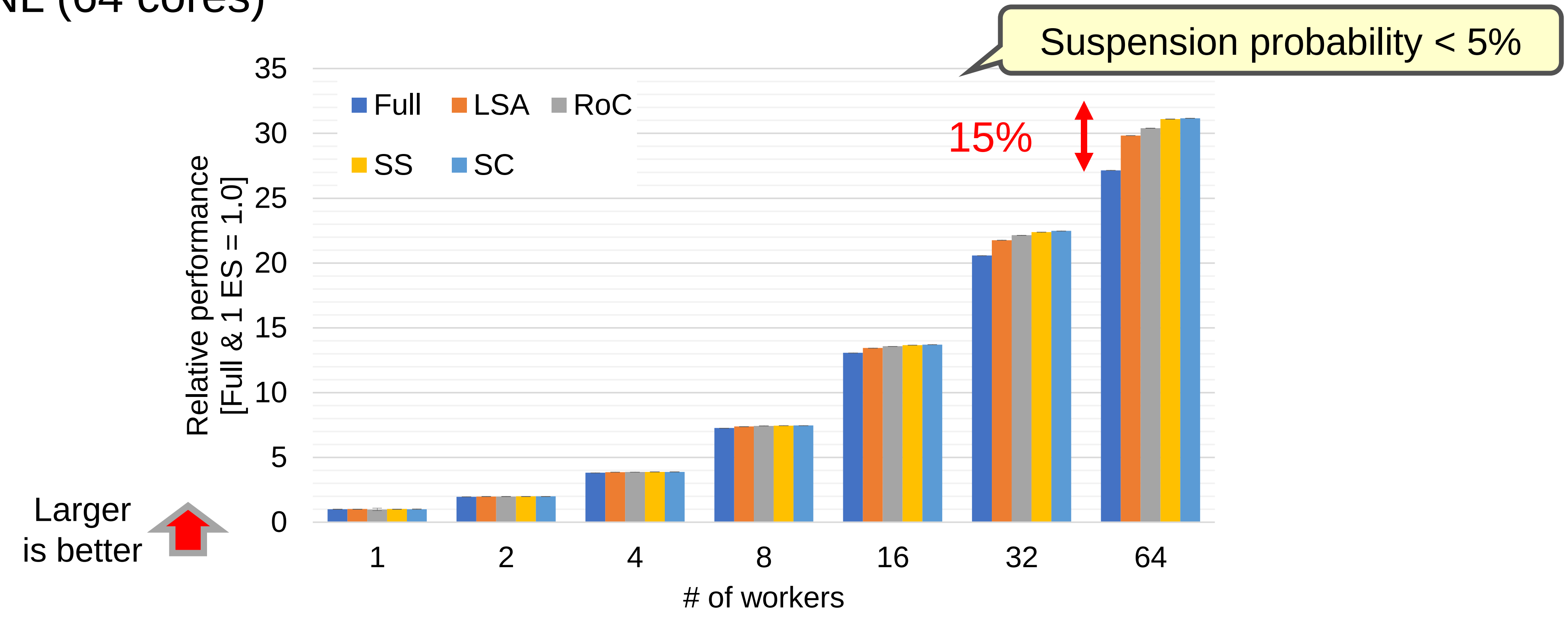
```
void task(void* arg) {
  if (is_leaf(arg))
    calc(arg)
  else {
    for (...)
      create task(...);
    join_all;
  }
}
```



- ExaFMM: Optimized O(N) N-body solver.
- Parent ULTs need to suspend if child ULTs do not finish at `join_all`.
- However, leaf ULTs never suspend since they do not join.
  - Suspension rarely happens ➡ dynamic promotion techniques should perform better!

1. ExaFMM: Performance

- Keep “# of ULTs / worker” for load balancing and increase # of workers on KNL (64 cores)



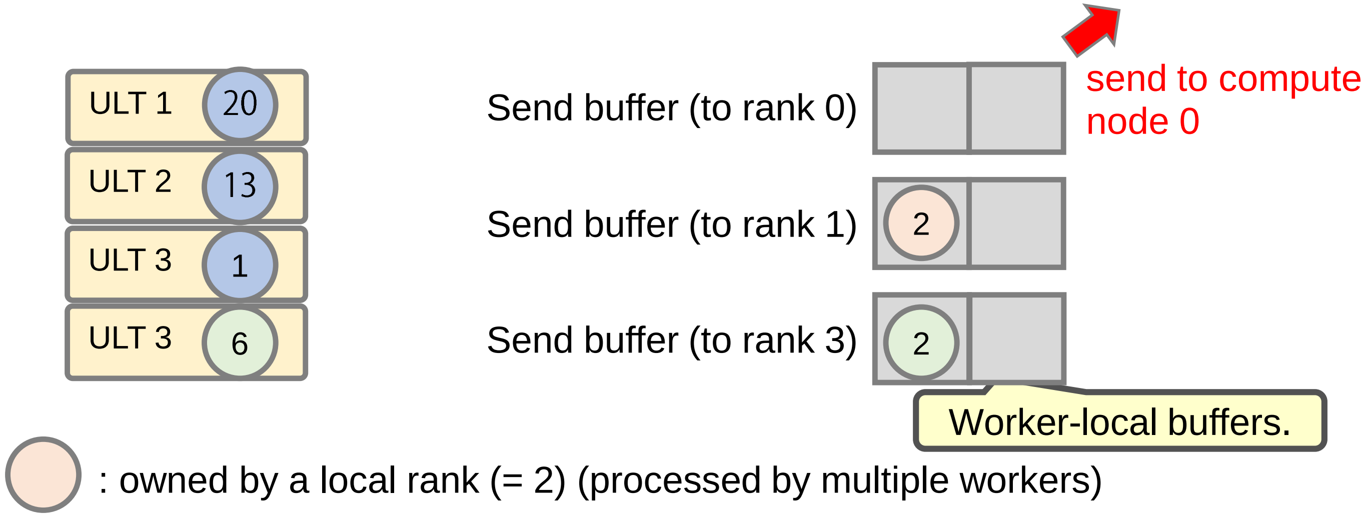
- Performance: **Full** < **LSA** < **RoC** < **SS, SC**

Dynamic promotion performs better.

2. Graph500: Latency Hiding

- `MPI_MULTIPLE_THREADS` on ULT-Aware MPI : one process per node
- Fine-grained Graph500: graph traversal on multiple nodes.
  - One ULT deals with one update vertex.

```
void update(int64_t vindex) {
  int owner_rank = get_owner(vindex);
  if (owner_rank == my_rank) {
    [...]; // update local graph
  } else {
    if (send_buffer.is_full())
      [MPI calls]; // might suspend!
    send_buffer[owner_rank].push(vindex);
  }
}
```

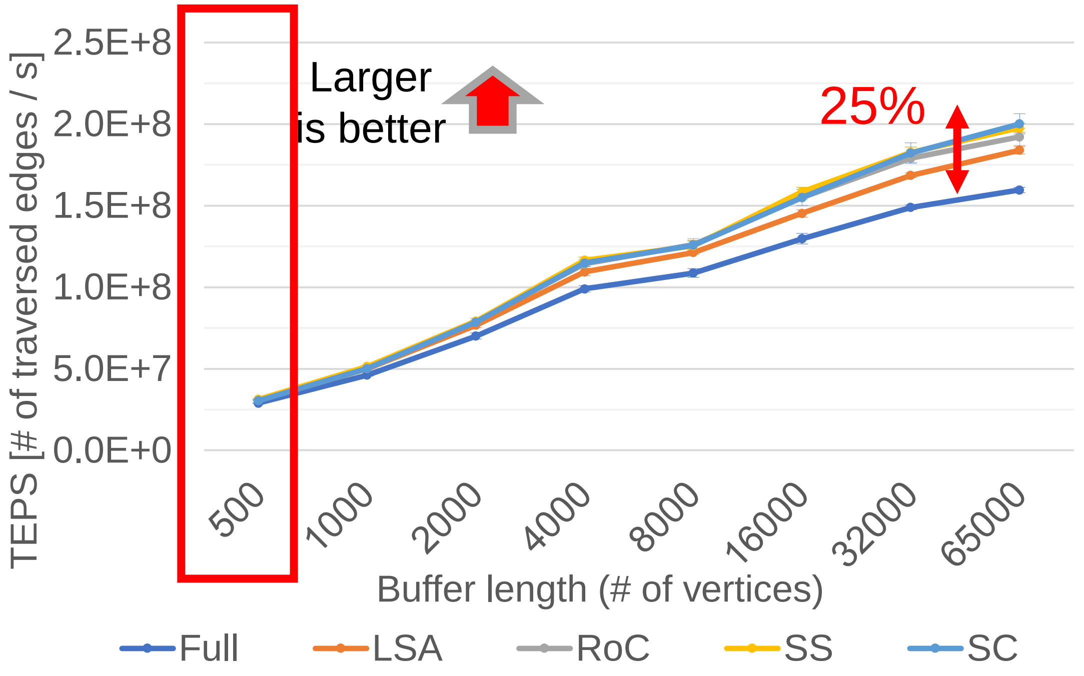


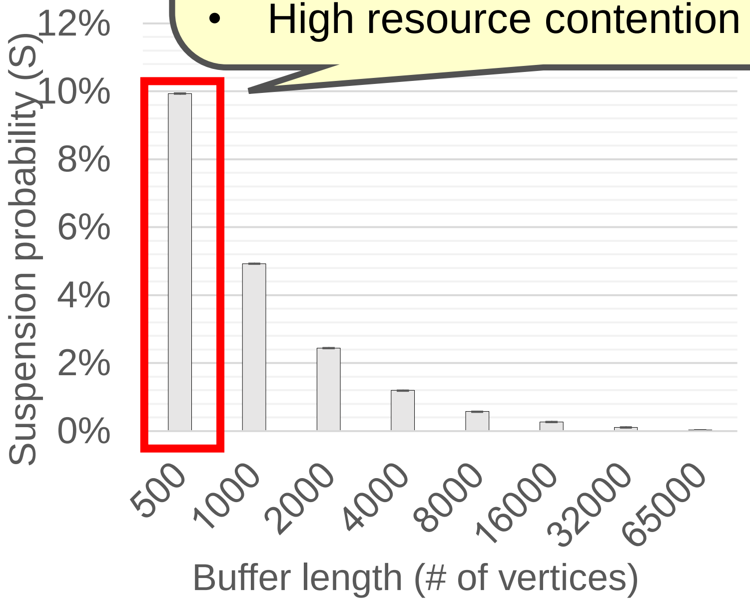
Only when the buffer is full, ULTs can suspend in MPI calls.
➡ If send buffer is large, only few ULTs suspend!

Omit the explanation on the receiver side.

2. Graph500 : Performance

- 16 KNLs (1K cores in total) + Omni-Path (MPICH3.2.x + CH3 OFI1.4.0 + PSM2)
  - The send buffer size is changed.





When S is high, Full might perform better. However, threading overheads are negligible because of *other performance issues causing suspension*

- High resource contention

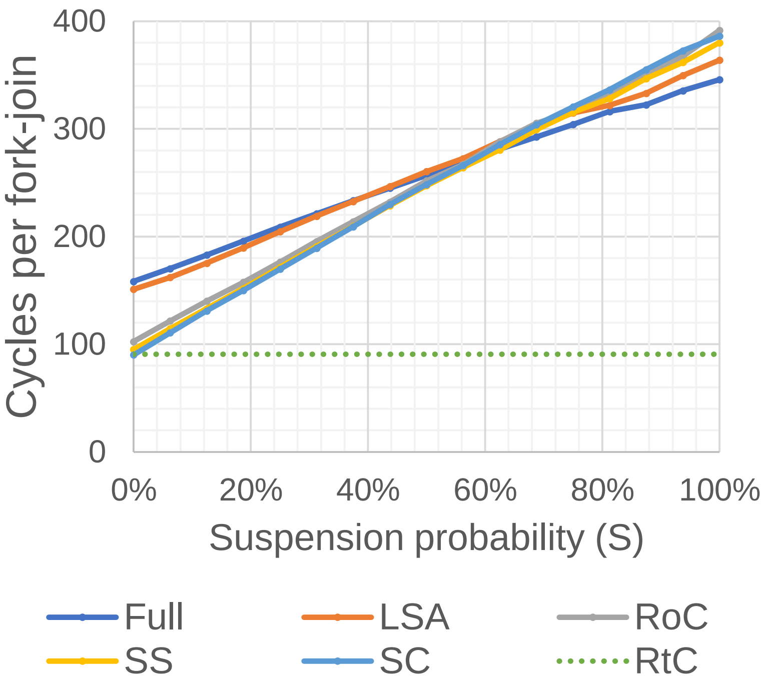
- Performance: **Full** < **LSA** < **RoC**, **SS**, **SC**

Dynamic promotion performs better.

Conclusion: Lessons Learned from Analysis

	Nonsuspension Case			Suspension Case		
	Change Stack?	# of ctx switches	Overhead	Rerun sched.?	Overhead	Constraints
Full	Yes	2	High	No	Low	No
LSA	Yes	2		No		No
RoC	Yes	1		No		No
SS	Yes	0		Yes		*
SC	No	0		Yes	High	**
RtC	No	0	Low	-	-	***

* Schedulers must be stateless.
** Schedulers must be stateless. Stack size of schedulers and ULTs is shared.
*** Threads are unable to yield.



- Trade-off between S=0% performance and functionality
- Trade-off between S=0% and S=100% performance
- RoC shows a good trade-off
  - Full threading capability + good S=0% performance

Argobots 1.0rc employs RoC:
- http://www.argobots.org/
- https://github.com/pmodels/argobots

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

Future Work

1. Automatic selection of those techniques
   - Runtime selection based on profiling?
2. Investigating overheads of other factors
   - Scheduling policy, memory allocators, thread pools…
3. Higher-level runtime systems
   - Apply those techniques to OpenMP
     - Can we simply apply our techniques?
     - Do OpenMP parallel units have other fundamental overheads?

This research was supported by
the Exascale Computing Project (17-SC-20-SC),
a joint project of the U.S. Department of Energy's
Office of Science and National Nuclear
Security Administration, responsible for delivering a
capable exascale ecosystem, including software,
applications, and hardware technology, to support the
nation's exascale computing imperative.