Improving Tool Support for Nested OpenMP Parallel Regions with Introspection Consistency

Vladimir Indic
University of Novi Sad

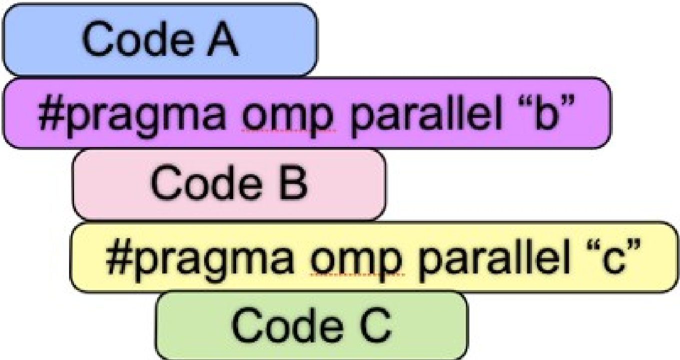
John Mellor-Crummey
Rice University

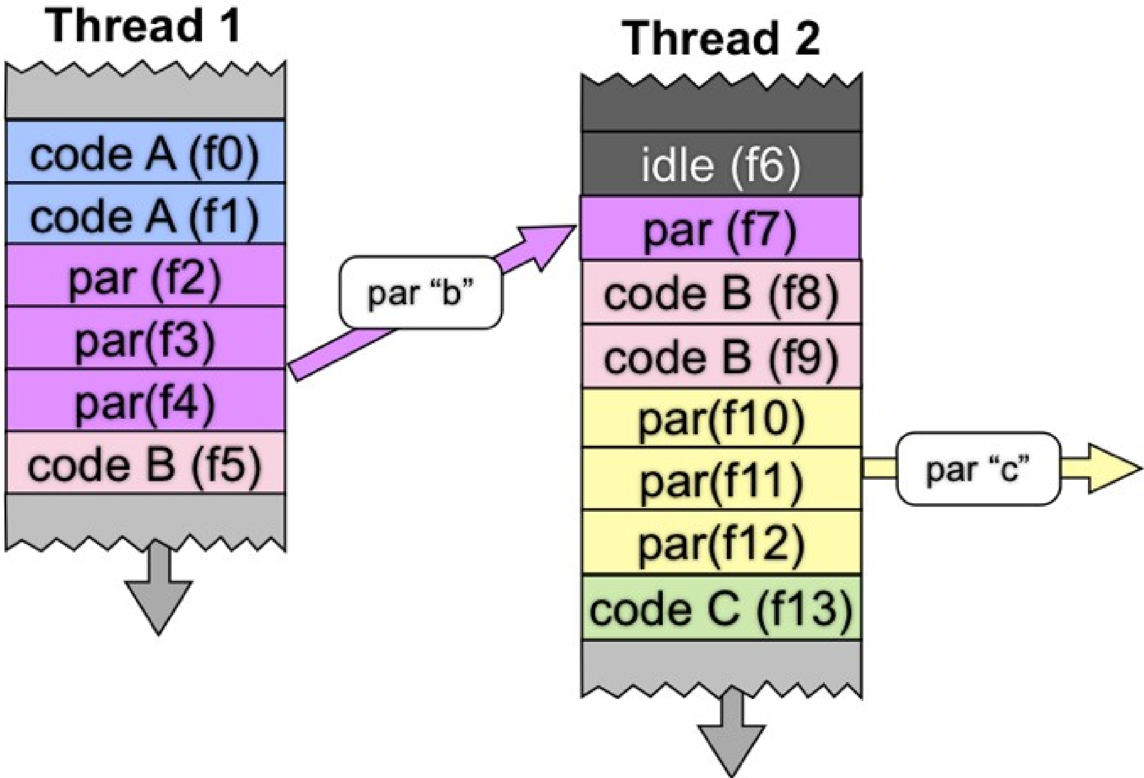
RICE UNIVERSITY

OpenMP Application-level Context is Distributed across Threads

Developer view

Implementation reality

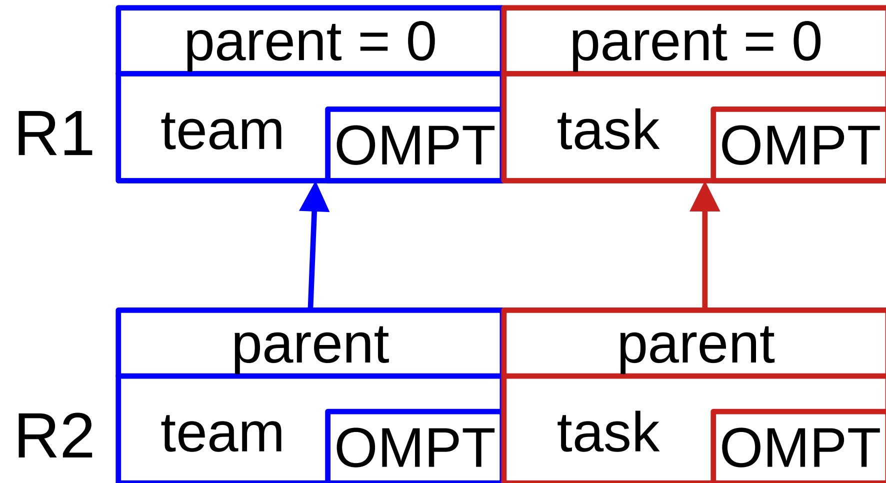




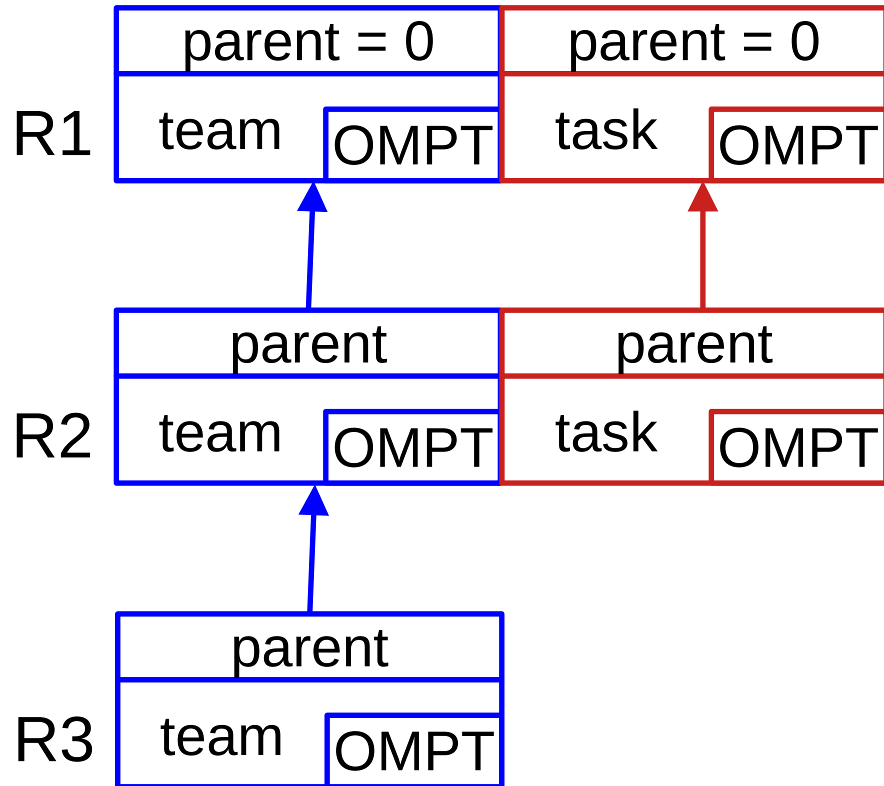
OMPT: An OpenMP Tools API

- **Provide introspection API for call stack unwinding**
  - A tool detects presence of active parallel/task regions asynchronously:
    - **ompt_get_parallel_info**(int **ancestor_level**, ompt_data_t ****parallel_data, …**)
    - **ompt_get_task_info**(int **ancestor_level**, ompt_data_t ****task_data,** ****parallel_data, …**)
- **Maintains state for each thread**
- **Provide API for tool to register and receive callbacks for important operations**

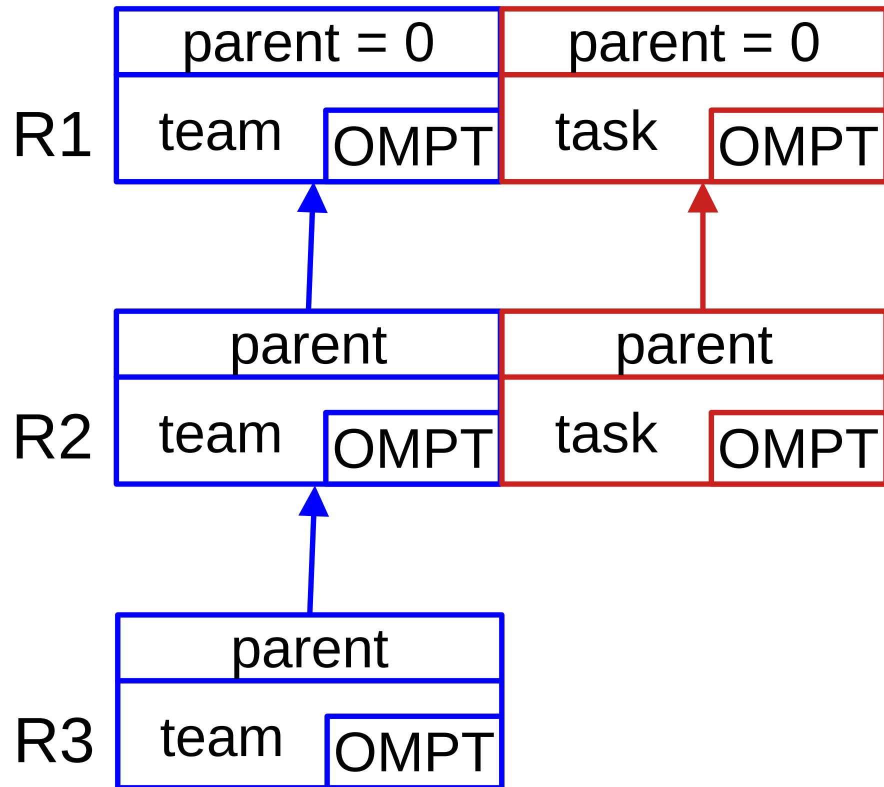
OMPT Under the Hood - LLVM OpenMP Runtime



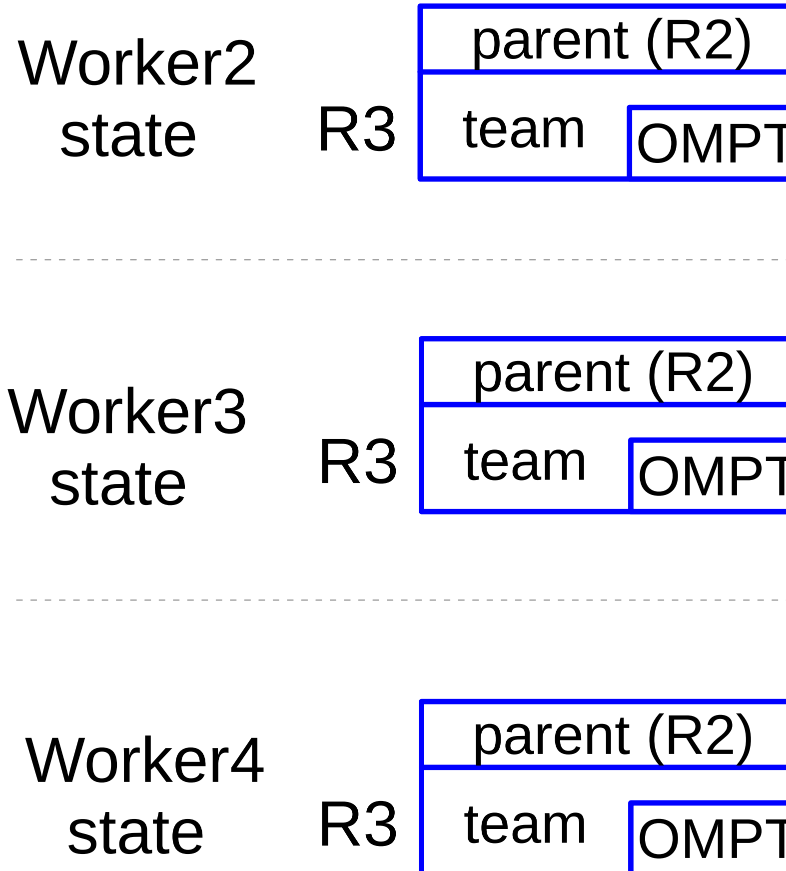
OMPT Under the Hood - LLVM OpenMP Runtime



OMPT Under the Hood - LLVM OpenMP Runtime

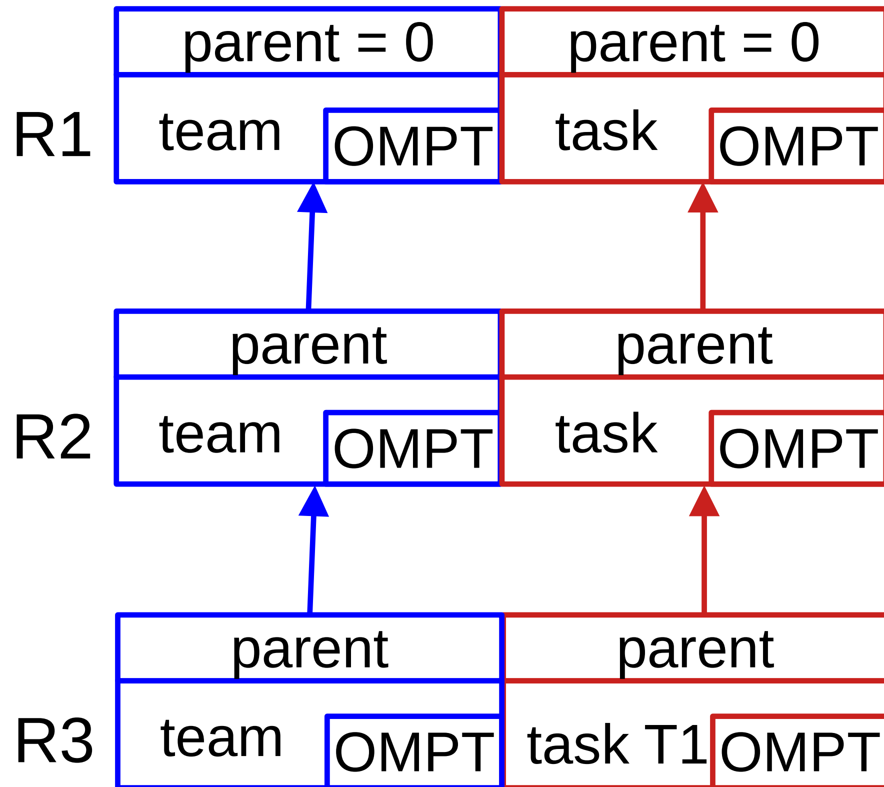


Primary Thread 1 State

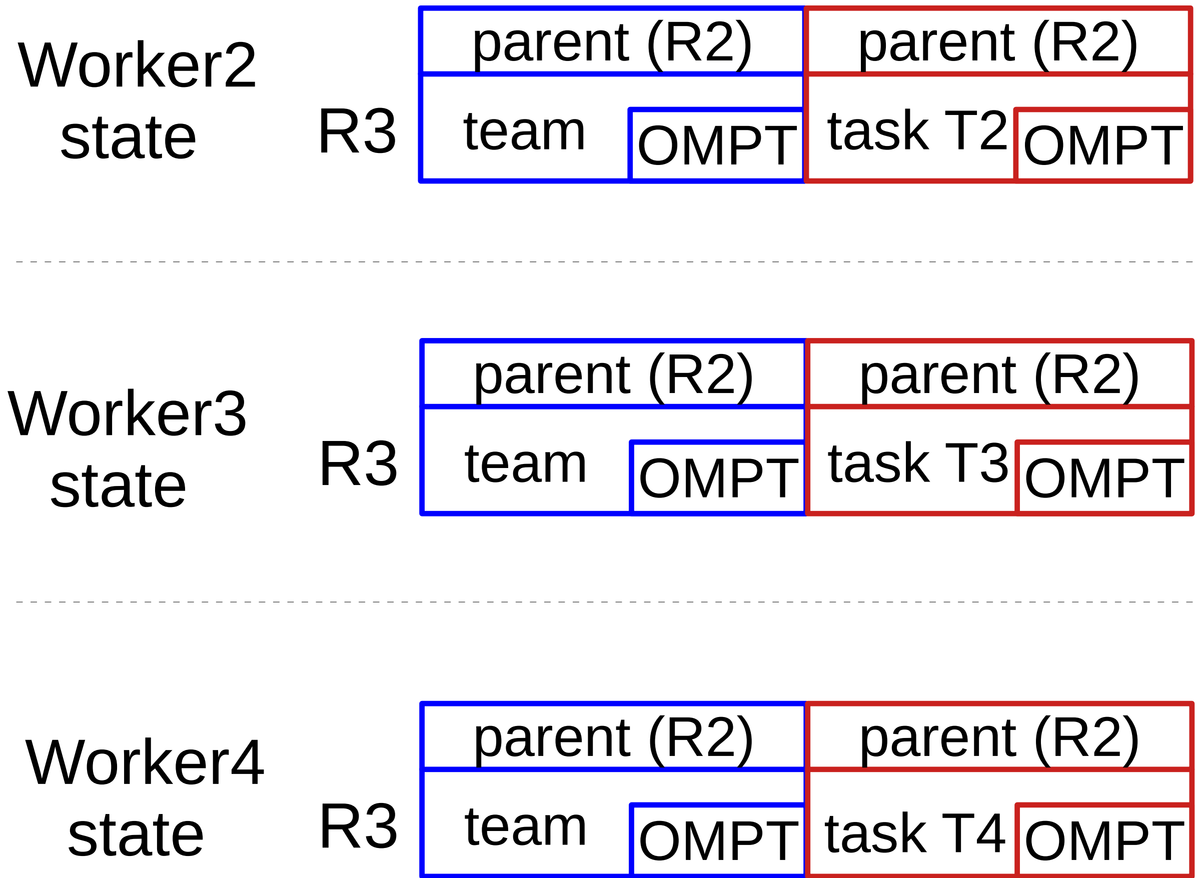


Worker threads 2-4 states

OMPT Under the Hood - LLVM OpenMP Runtime

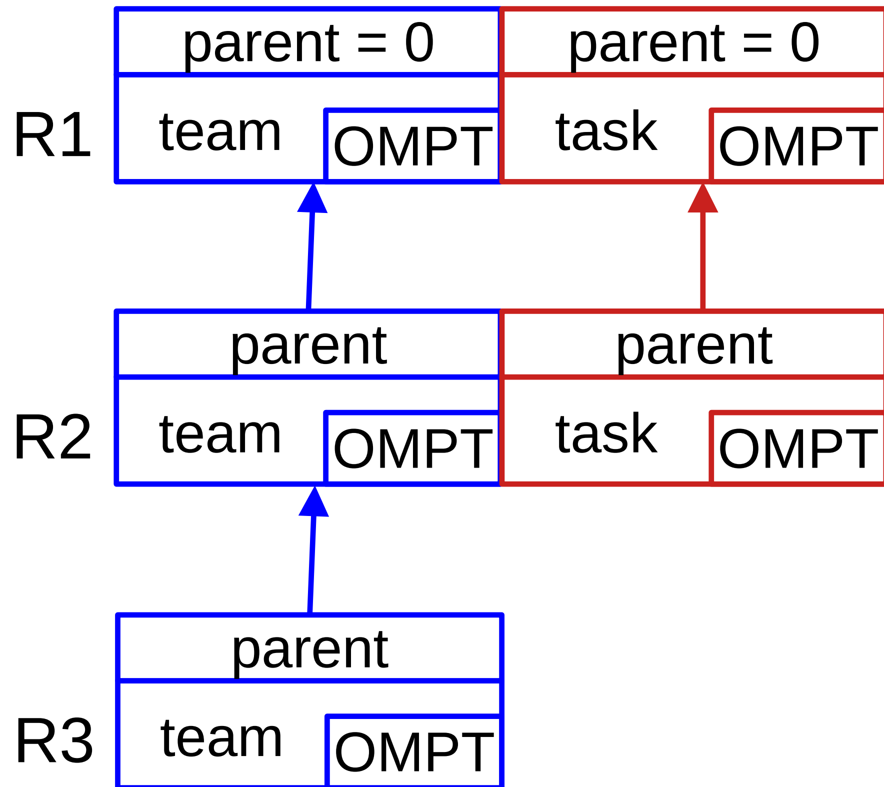


Primary Thread 1
State

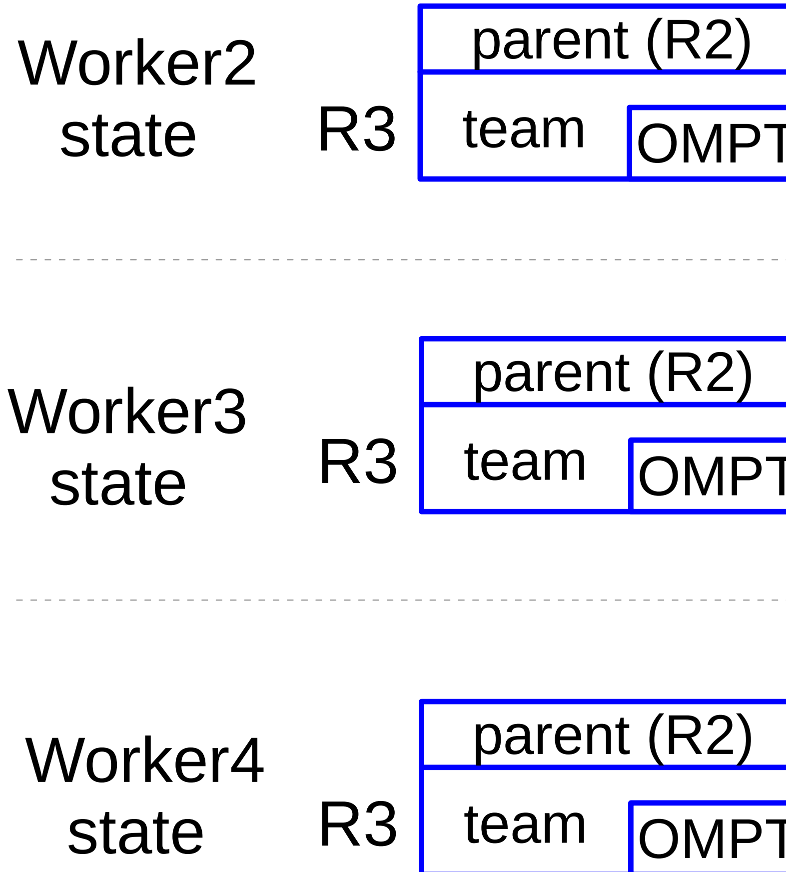


Worker threads 2-4
states

OMPT Under the Hood - LLVM OpenMP Runtime

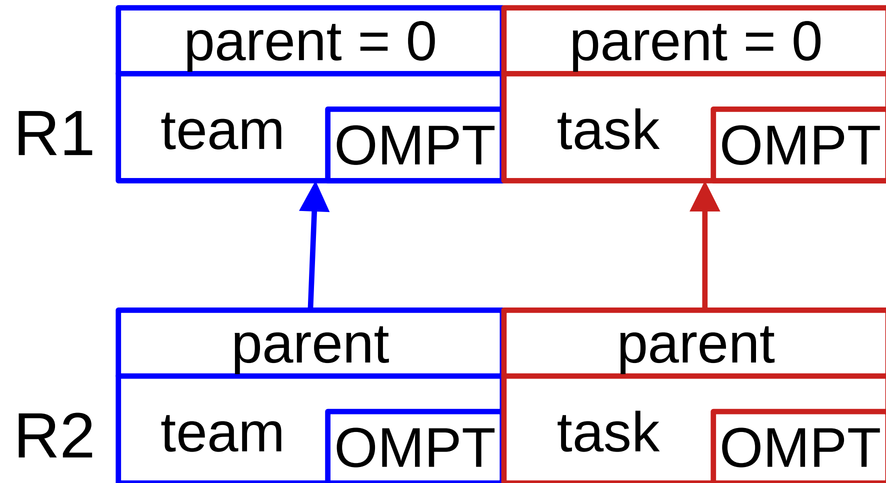


Primary Thread 1 State

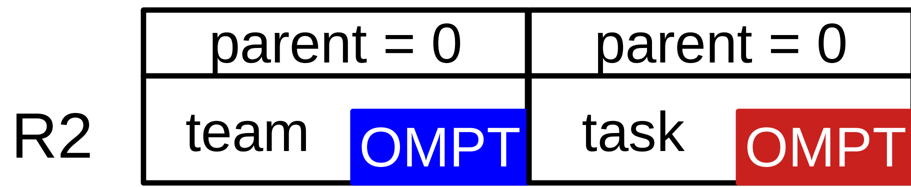


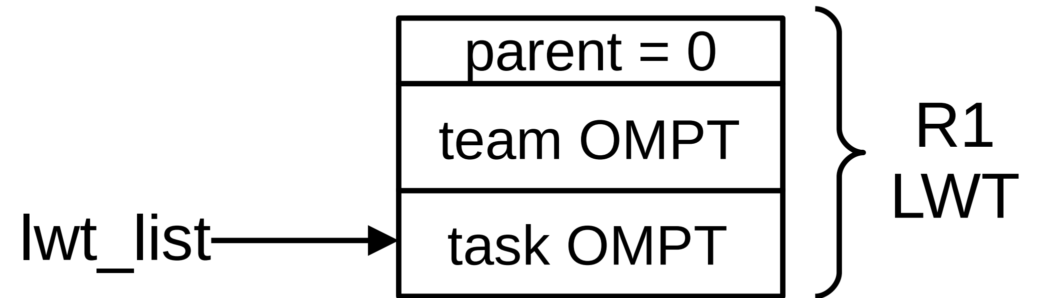
Worker threads 2-4 states

OMPT Under the Hood - LLVM OpenMP Runtime



OMPT Under the Hood - Serialized Regions

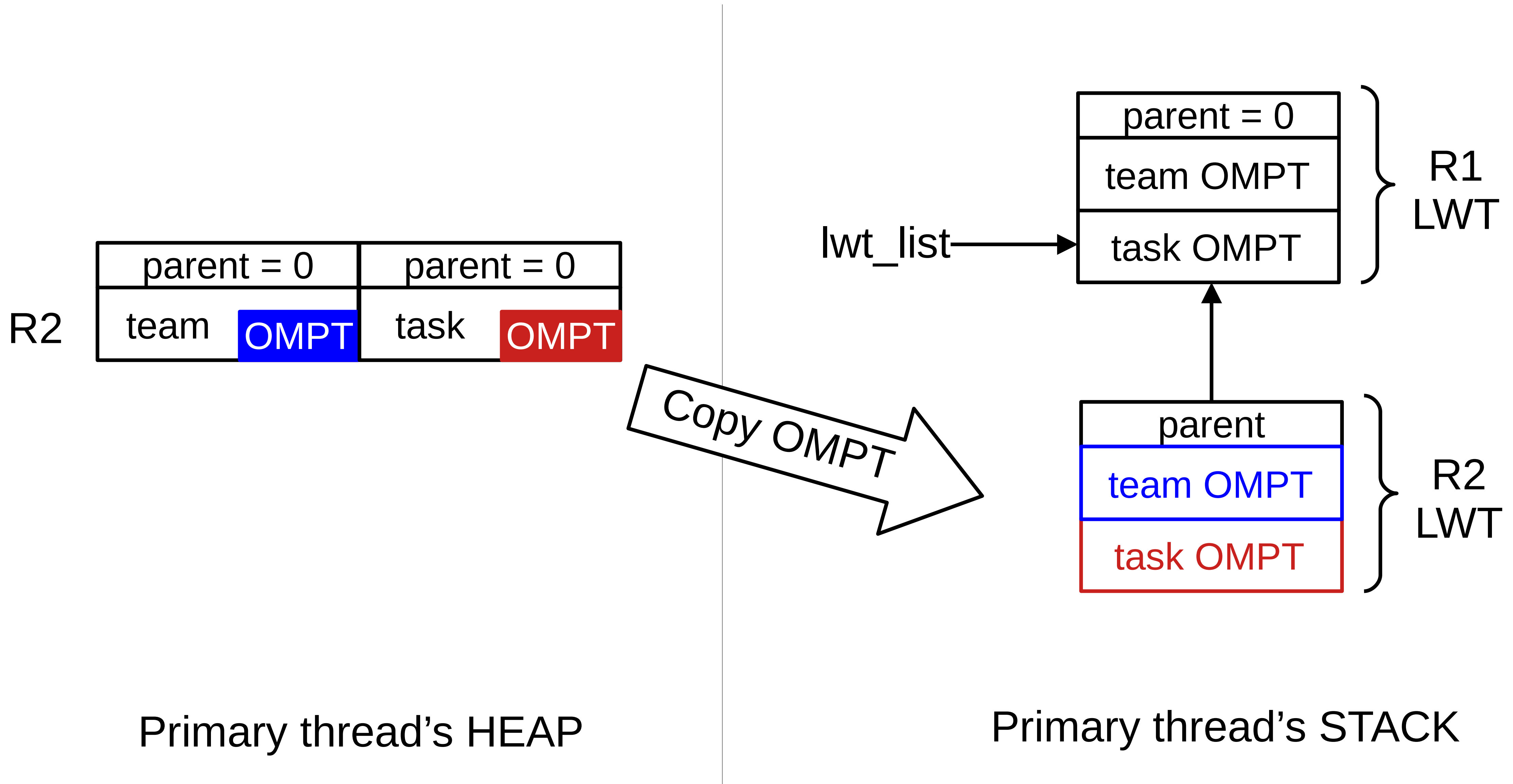




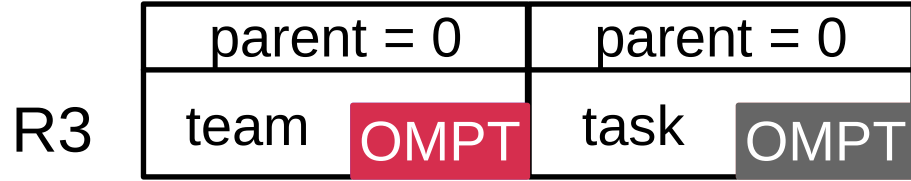
Primary thread's HEAP

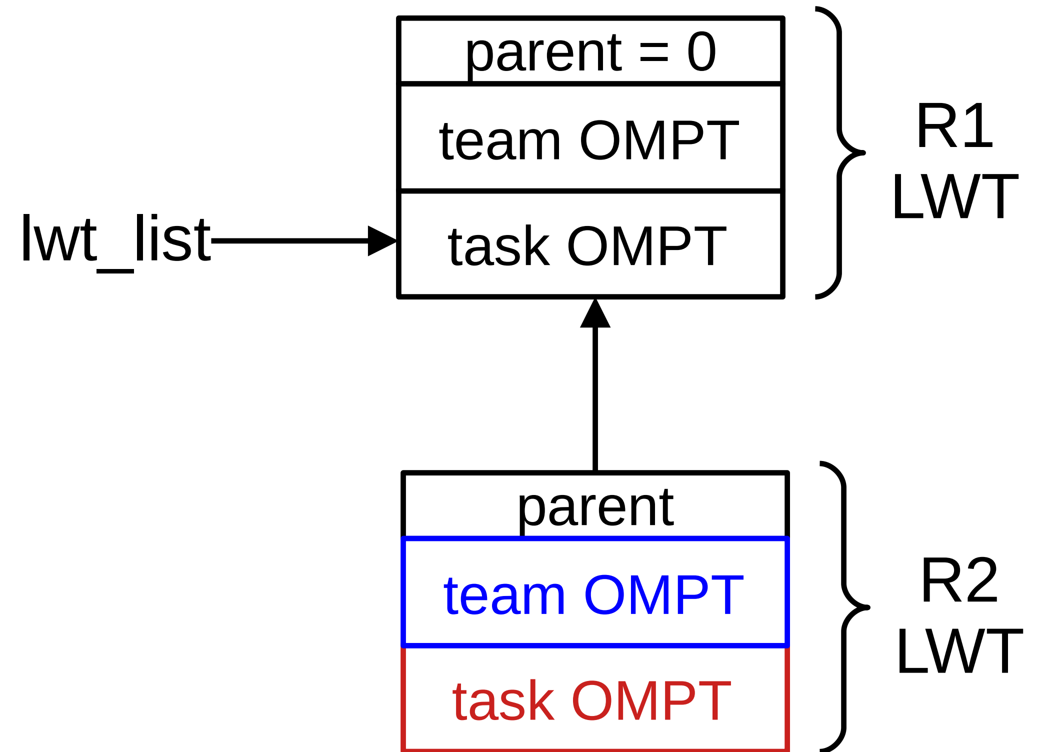
Primary thread's STACK

OMPT Under the Hood - Serialized Regions



OMPT Under the Hood - Serialized Regions

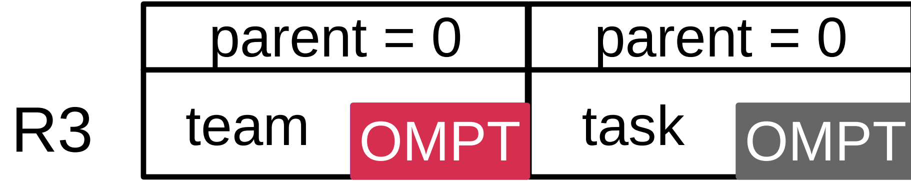




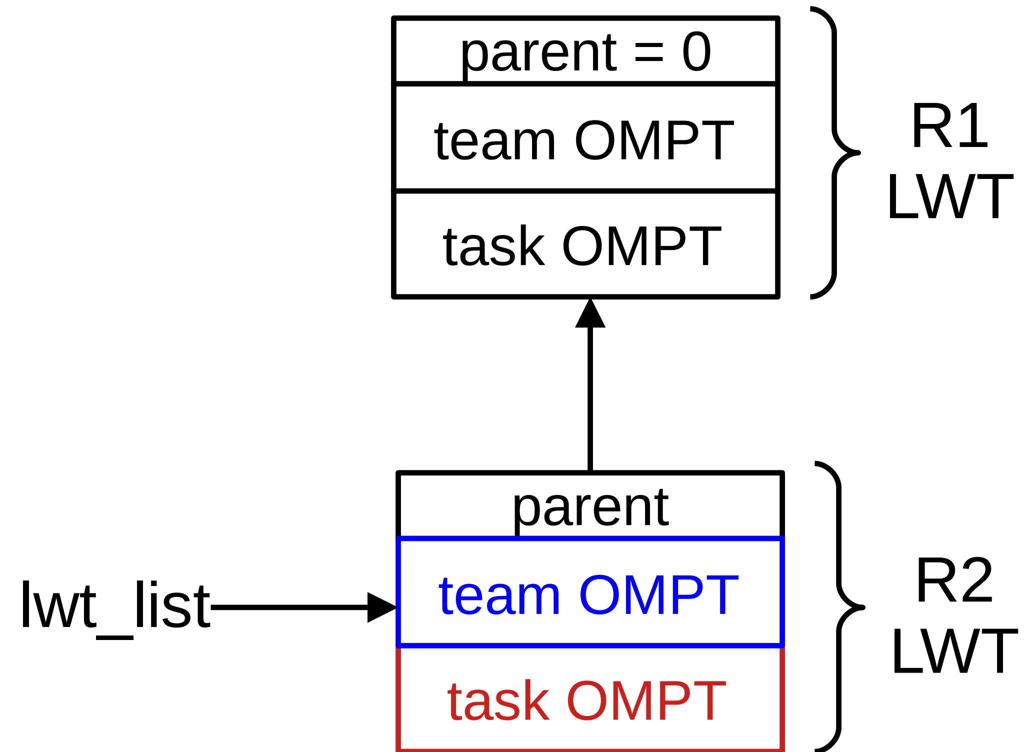
Primary thread's HEAP

Primary thread's STACK

OMPT Under the Hood - Serialized Regions



Primary thread’s HEAP

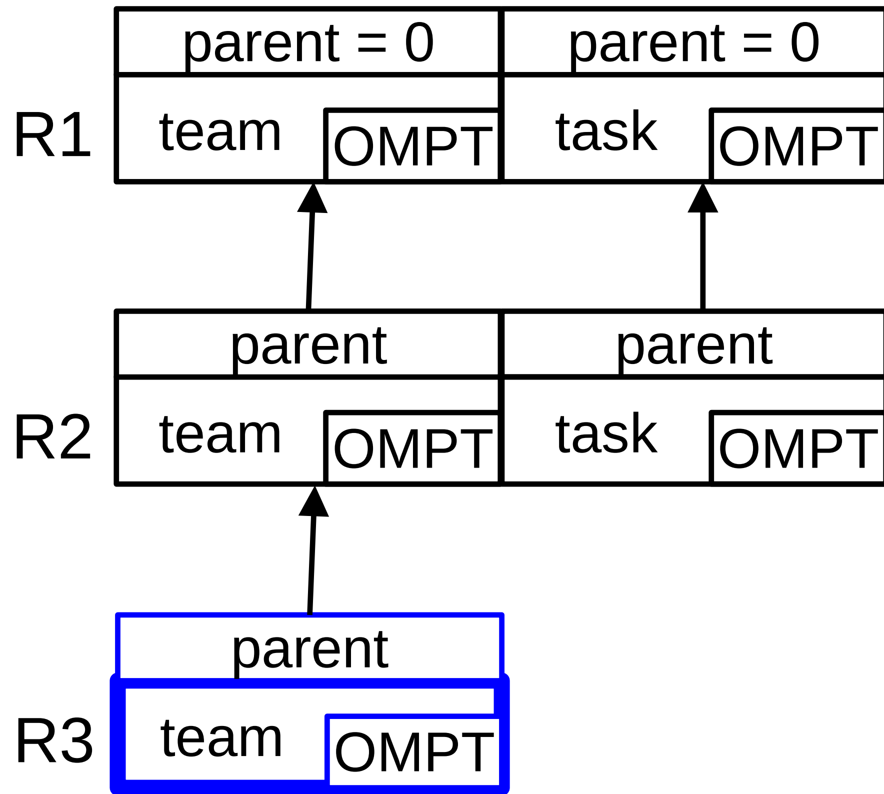


Primary thread’s STACK

Challenges

- **At an arbitrary point in time, e.g. when a timer expires**
  - Tool invokes introspection routine to inspect the current chain of teams or tasks
  - Tool may inspect and/or update the 64-bit ompt_data_t in any team/task in the chain
- **Sometimes, LLVM runtime fails to provide correct OMPT information and loses tool data**
  - **Disaster for a tool!**

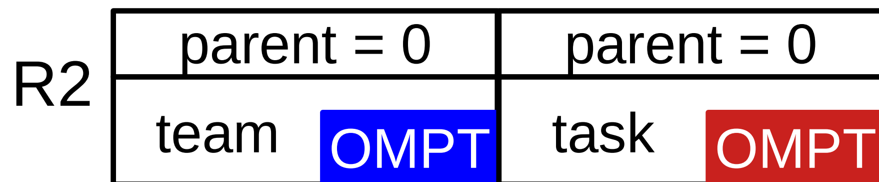
Failure 1 - Regular Parallel Creation Interrupted

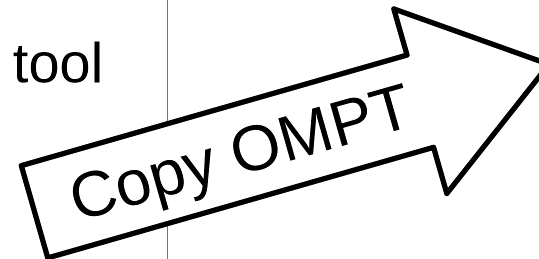


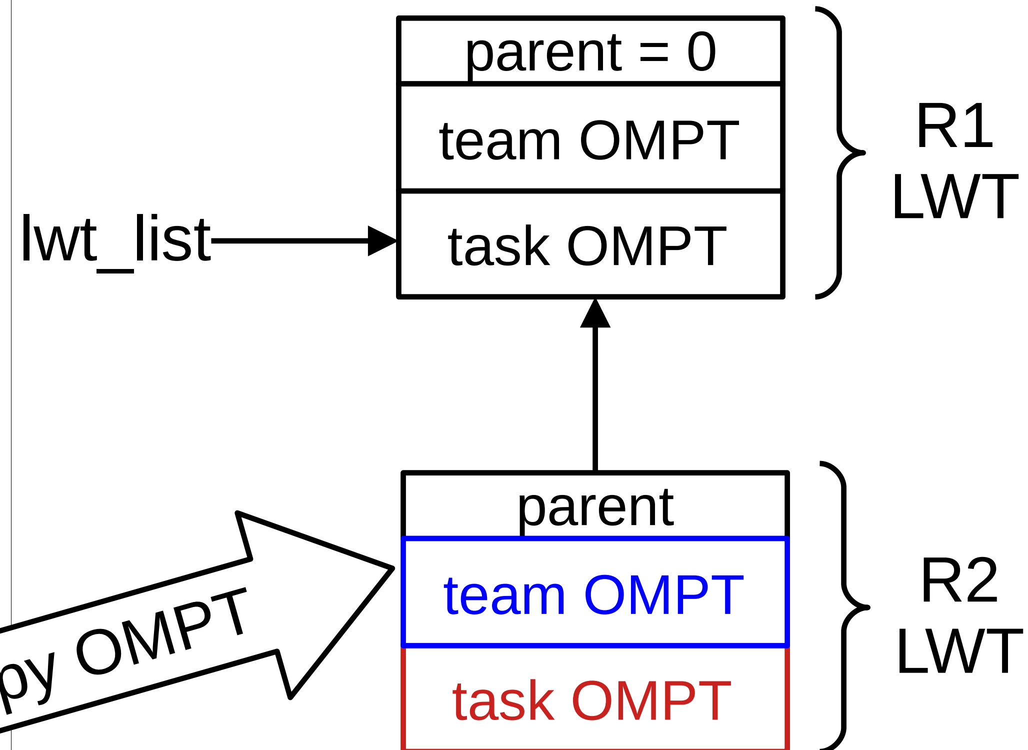
- Recall creation of a nested serialized region
  - After updating team descriptor (blue)
  - Before setting up the corresponding implicit task
  - **Interrupt!**
- **Team and task information do not match**
  - (R3 team, R2 task)
  - (R2 team, R1 task)
  - ...

Failure 2 - Nested Serialized Region Creation Interrupted

- Recall creation of a nested serialized region
  - Copying R2 OMPT when entering R3
  - **Intterupt!**
- **Where do R2 OMPT resides?**
- **Tool updates data while the copying is in progress**
  - Miss an update of task/team data by the tool







Primary thread's HEAP

Primary thread's STACK

OpenMP Introspection

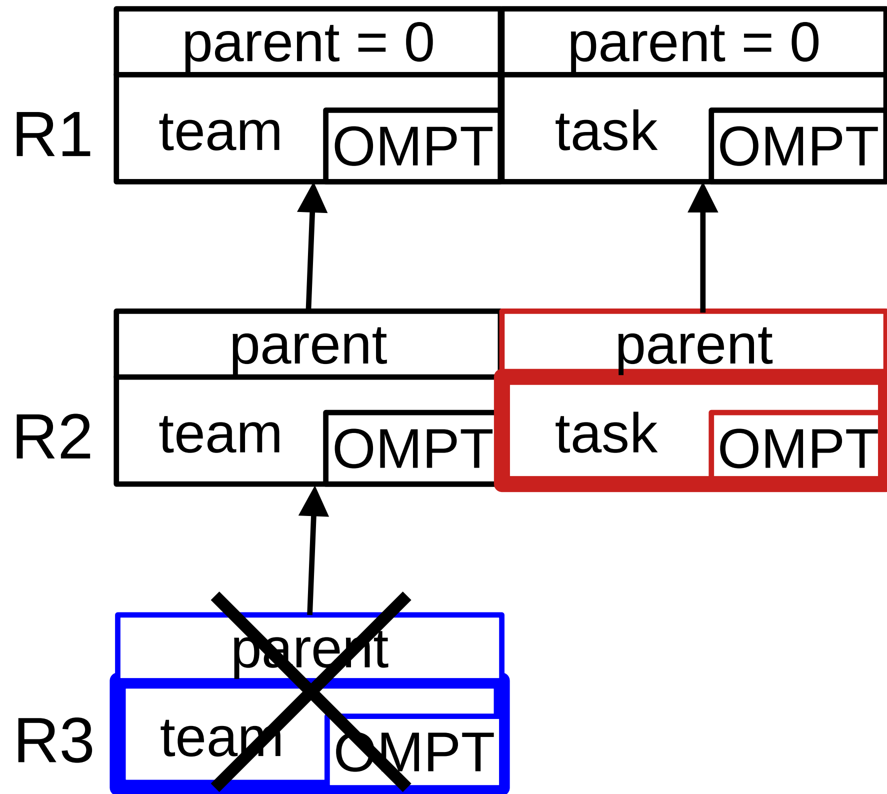
- **Standard guarantees only weak property**
  - **Information about any active region/task may be reported as unavailable**

Our Proposal: OpenMP Introspection Consistency

- **A tool must receive valid and consistent OMPT information**
  - about **parallel region** and its **implicit task**: **from implicit-task-begin until the implicit-task-end** of the primary thread in the region
- **How to preserve it?**

Must also guarantee introspection consistency for explicit tasks (elided for time).

Parallel Region Creation and Introspection Consistency

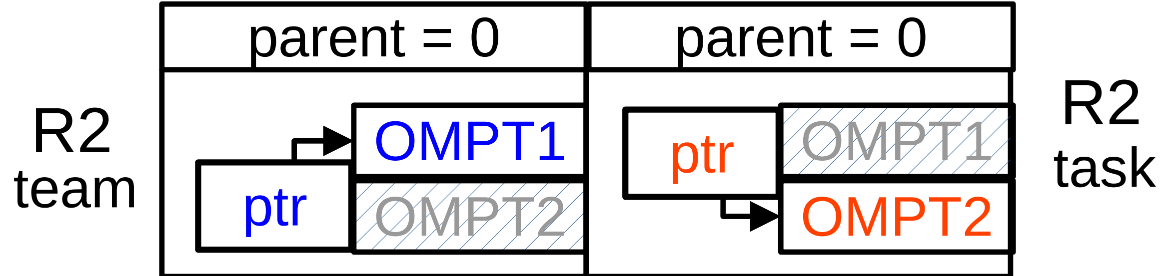


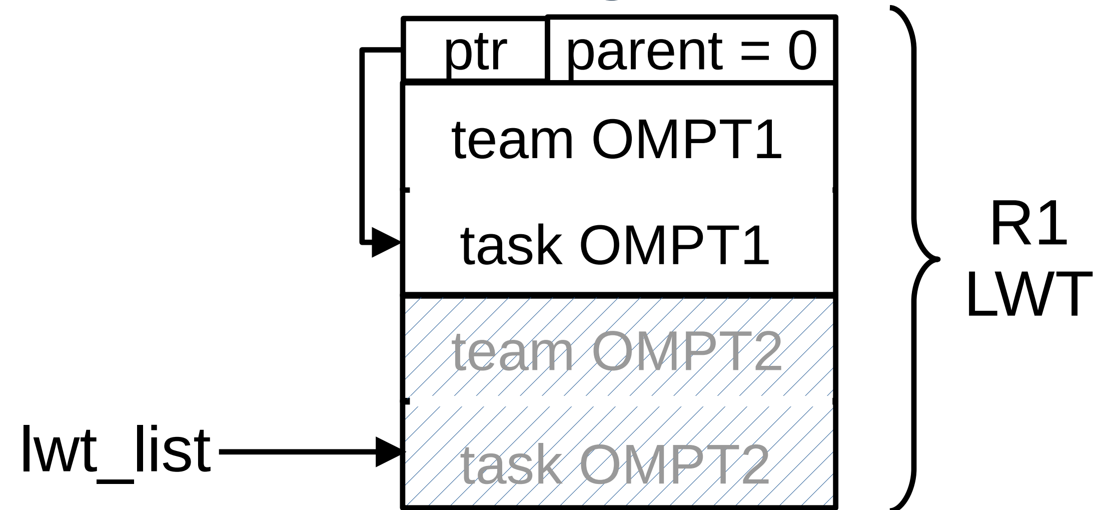
- Follow the chain of task and team descriptors unless:
  - **Current task (red) descriptor does not match current team (blue) descriptor**
    - Creating/destroying of region in progress
    - Skip the current team representing old/new region
    - Then follow the chain
  - Possibly unable to provide the information about the innermost team

Nested Serialized Parallel Regions and Introspection Consistency

- **Introspection routine (IR) needs to know what runtime is doing**
- **IR helps the runtime (RT) finish creation/destruction of nested serialized region**
- **Wait-free coordination protocol**
  - **Finite number of steps** to decide where OMPT information resides
  - Neither the runtime or introspection routines will wait for one another

OMPT Under the Hood - Serialized Regions

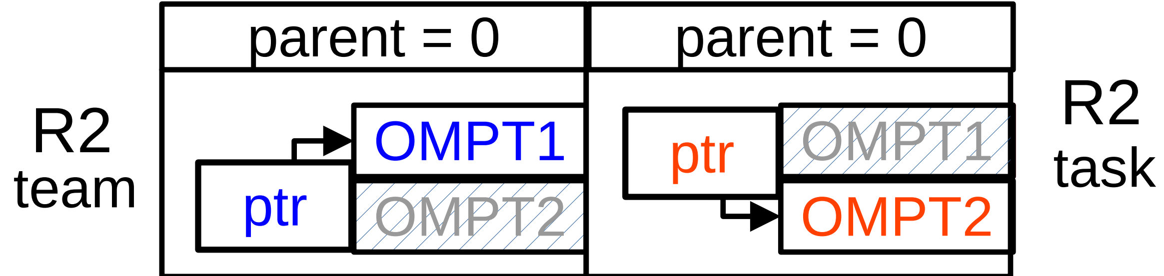




Primary thread's HEAP

Primary thread's STACK

OMPT Under the Hood - Serialized Regions



Primary thread's HEAP

ptr | parent = 0
team OMPT1
task OMPT1
team OMPT2
task OMPT2

R1
LWT

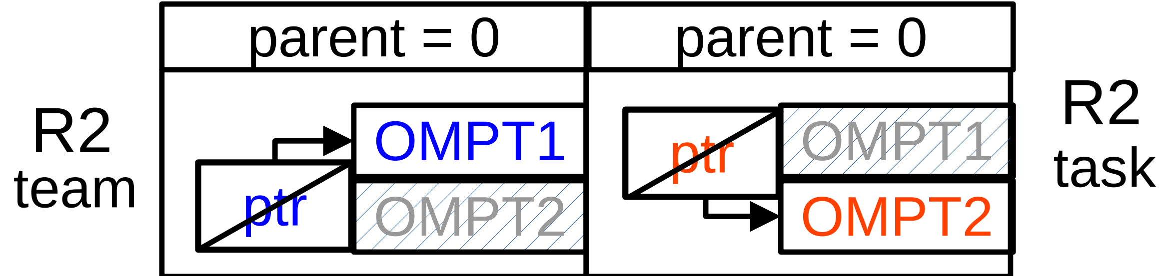
lwt_list

ptr=0 | parent
???
???

R2
LWT

Primary thread's STACK

OMPT Under the Hood - Serialized Regions

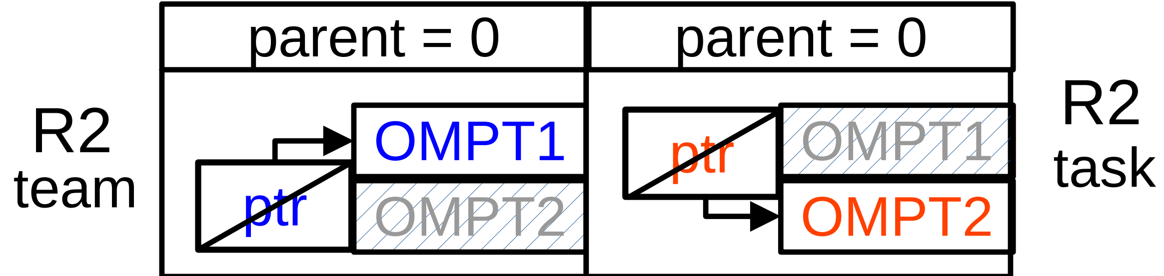


Primary thread's HEAP

ptr | parent = 0
team OMPT1
task OMPT1
team OMPT2
task OMPT2
R1
LWT
lwt_list
ptr=0 | parent
???
???
R2
LWT

Primary thread's STACK

OMPT Under the Hood - Serialized Regions

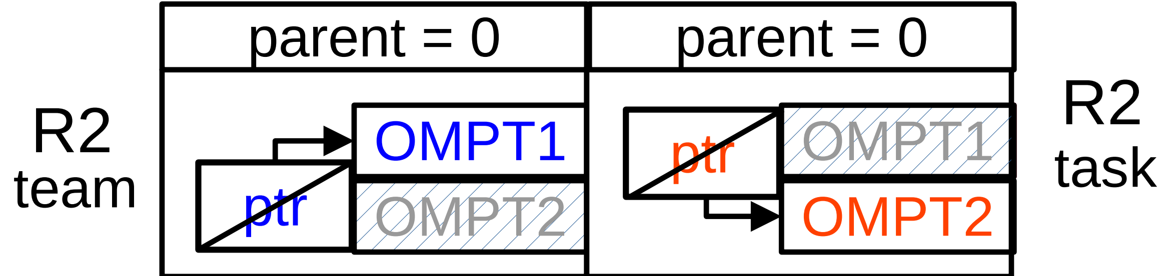


Primary thread's HEAP

ptr
parent = 0
team OMPT1
task OMPT1
team OMPT2
task OMPT2
R1
LWT
ptr=0
parent
???
???
R2
LWT
lwt_list

Primary thread's STACK

OMPT Under the Hood - Serialized Regions



Primary thread’s HEAP

ptr | parent = 0
team OMPT1
task OMPT1
team OMPT2
task OMPT2

R1
LWT

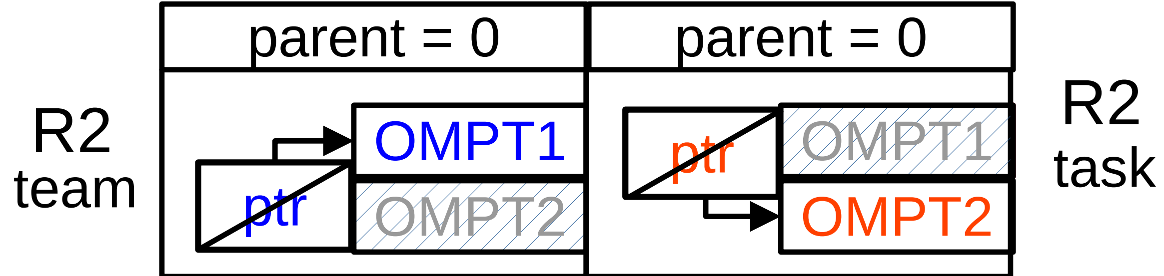
ptr=0 | parent
team OMPT1
task OMPT2
???
???

R2
LWT

lwt_list

Primary thread’s STACK

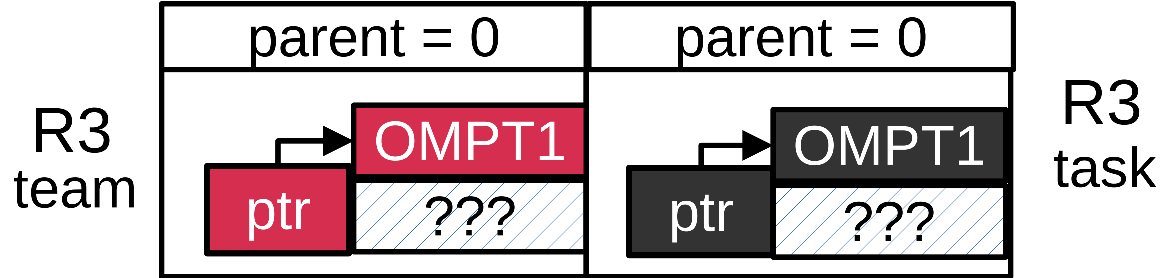
OMPT Under the Hood - Serialized Regions



Primary thread's HEAP

Primary thread's STACK

OMPT Under the Hood - Serialized Regions



Primary thread's HEAP

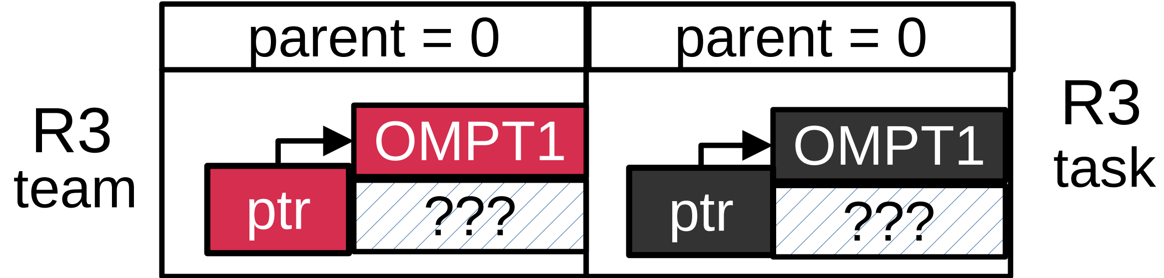
ptr | parent = 0
team OMPT1
task OMPT1
team OMPT2
task OMPT2
R1 LWT

ptr | parent
team OMPT1
task OMPT2
???
???
R2 LWT

lwt_list

Primary thread's STACK

OMPT Under the Hood - Serialized Regions



Primary thread's HEAP

ptr | parent = 0
team OMPT1
task OMPT1
team OMPT2
task OMPT2
R1
LWT

ptr | parent
team OMPT1
task OMPT2
???
???
R2
LWT
lwt_list

Primary thread's STACK

Wait-Free Coordination Protocol: IR Perspective

- **if lwt_list is tagged (the process in progress)**
  - **finish region creation/destruction!**
    - **Do the same steps as the runtime:**
      - **Use regular store registers**
        - Cannot be interrupted by the runtime
        - Store the second element’s address (&pair[2])
      - **Always makes the decision if the runtime hasn’t**
- **If not**
  - **reads from current_team/task or lwt_list entries**

Evaluation

- **Question: Will providing introspection consistency with our wait-free protocol hurt overall runtime performance?**
  - The descriptors are slightly bigger
  - We introduce an additional level of indirection when accessing the OMPT state
  - The implementation of the introspection routines is more complex

Evaluation

- **We compare three runtime implementations**
  - **U: Parent task and team information sometimes unavailable**
    - Improved version of LLVM upstream
  - **W: Wait-free implementation of introspection consistency**
  - **F: Introspection consistency using full region descriptors**
    - No lightweight support for serialized regions

Experimental Setup - System

- **Experiments performed on an idle system**
  - Intel Knights Landing (68 4-way SMT cores) running Linux
- **Linux Address Space Layout Randomization disabled to avoid unnecessary changes in code and data layout**
- **Three runtime implementations (W, U, F):**
  - built as shared libraries with Clang 12.0.0 -O3
  - OMPT_SUPPORT=ON

Experimental Setup - MicroBenchmarks

```
#pragma omp parallel num_threads(1)
  for (int i = 0; i < 16000000; i++)
    #pragma omp parallel num_threads(1)
      volatile int x = 0;
```

Serialized micro-benchmark

```
for (int i = 0; i < 4000000; i++)
  #pragma omp parallel num_threads(4)
    volatile int x = 0;
```

Parallel micro-benchmark

- Also built with Clang with -O3 -g,
- linked to one of the .so runtime libraries (U, W, F) corresponding the following experiments

Cost of Maintaining Nested Serialized Regions

- Running Serialized region benchmark instances standalone for 30 times.
- OMPT_SUPPORT compiled, but not used
- W and F overhead relative to U

Code	Time(s)	Overhead(%)
Serialized_**U**	8.9846 ± 0.0006	-
Serialized_**W**	8.8508 ± 0.0009	**-1.49**
Serialized_**F**	16.077 ± 0.007	**78.94**

- W and U are comparable
- ~80% overhead induced by heap allocation in F

Cost of Maintaining OMPT Information about Nested Serialized Regions

- Trivial tool attached to activate OMPT info maintenance
  - ompt_start_tool, an initializer, and a finalizer; nothing else
- W and F overhead relative to U

Code	Time(s)	Overhead(%)
Serialized_**U**	10.926 ± 0.004	**-**
Serialized_**W**	12.477 ± 0.007	**14.20**
Serialized_**F**	16.241 ± 0.012	**48.65**

- W introduces additional 2/3 costs
- **Still, W delivers introspection consistency 3x cheaper than F**

Nested Serialized Regions and Statistical Sampling

- Simple proxy tool:
  - Linux CPUTIME timer generates 200 samples/sec.
  - signal handler calls ompt_get_task_info for available parallel regions
- Overhead relative to trivial tool times (previous slide)

Code	Time(s)	Overhead(%)
Serialized_**U**	10.959 ± 0.001	**0.30**
Serialized_**W**	12.513 ± 0.008	**0.29**
Serialized_**F**	16.280 ± 0.002	**0.24**

- Similar sampling overhead for U, W and F

Cost of Maintaining Regular Parallel Regions

- Running P benchmark instances standalone for 30
- OMPT_SUPPORT compiled, but not used
- **OMPT_PROC_BIND=close** to reduce the deviation
- W and F overhead relative to U

Code	Time(s)	Overhead(%)
Parallel_**U**	23.588 ± 0.013	-
Parallel_**W**	23.610 ± 0.033	**0.09**
Parallel_**F**	23.377 ± 0.016	**-0.90**

- W and U are comparable
- Perf observation of F - less branch and icache load misses
  - Removing LWT support reduces the code size.

Something Unusual on Broadwell

- **Running Parallel microbenchmark on Broadwell architecture:**
  - significant standard deviation using the same approach
    - Additionally, disable dynamic CPU frequency adaptation
      - Maximal CPU frequency set to minimal

Conclusions

- **Practical experience with LLVM OpenMP revealed the shortcomings of the weak specification and problematic implementations of OMPT introspection routines**
- **We proposed introspection consistency as a firm foundation for reliable OpenMP tools**
- **We developed novel implementation of wait-free coordination protocol that provides introspection consistency at reasonable cost**
- **We found repeatable experiments to be very challenging and we would loved to discuss this problem with folks at the Workshop**