Legion Bootcamp: Debugging & Profiling Tools

Wonchan Lee

Debugging Tools:
Runtime Checks & LegionSpy

December 4, 2014

http://legion.stanford.edu

Runtime Checks

- Provide warning or error messages when the application breaks runtime's assumptions

Runtime Checks

- Provide warning or error messages when the application breaks runtime's assumptions

1. Disjointness checks

- Verify the disjointness of the index partitioning claimed to be disjoint

```
IndexPartition create_index_partition(Context ctx, IndexSpace parent,
                                      const Coloring &coloring,
                                      bool disjoint,
                                      int part_color = -1);
```

- Enabled by passing the `-hl:disjointness` flag on the command line

```
$ ./partitioning -hl:disjointness
Running daxpy for 1024 elements...
Partitioning data into 3 sub-regions...
[0 - 1] {ERROR}{runtime}: ERROR: colors 0 and 1 of partition 1 are not disjoint when they
are claimed to be!
Assertion failed: (false), function create_index_partition, file /Users/wclee/Workspace/
stanford/projects/legion//runtime/runtime.cc, line 5348.
```

2. Privilege checks
3. Bounds checks

Runtime Checks

- Provide warning or error messages when the application breaks runtime's assumptions

1. Disjointness checks

2. **Privilege Checks**
   - **Dynamically verify if all memory accesses abide by the privileges stated in task's region requirements**
   - **Compensate lack of static checking from the compiler**
     (However, Legion language has a type system and a compiler.
     See the next talk!)

3. Bounds checks

Runtime Checks

- Provide warning or error messages when the application breaks runtime's assumptions

1. Disjointness checks
2. **Privilege Checks**
   - **Enabled by compiling runtime source with the flag** `-DPRIVILEGE_CHECKS`
   - **E.g.**

```
TaskLauncher init_launcher(INIT_FIELD_TASK_ID, TaskArgument(NULL, 0));
init_launcher.add_region_requirement(
  RegionRequirement(input_lr, READ_ONLY, EXCLUSIVE, input_lr));
init_launcher.add_field(0, FID_X); runtime->execute_task(ctx, init_launcher);

void init_field_task(const Task *task, const vector<PhysicalRegion> &regions,
                     Context ctx, HighLevelRuntime *runtime)
{
  ...
  RegionAccessor<AccessorType::Generic, double> acc =
    regions[0].get_field_accessor(FID_X).typeify<double>();

  for (GenericPointInRectIterator<1> pir(rect); pir; pir++)
    acc.write(DomainPoint::from_point<1>(pir.p), drand48());
```

3. Bounds checks

Runtime Checks

- Provide warning or error messages when the application breaks runtime's assumptions

1. Disjointness checks

2. Privilege Checks

- Enabled by compiling runtime source with the flag `-DPRIVILEGE_CHECKS`
- E.g.

```
$ ./privileges
Running daxpy for 1024 elements...
Initializing field 0...
PRIVILEGE CHECK ERROR IN TASK init_field_task: Need WRITE-DISCARD privileges but only hold
READ-ONLY privileges
Assertion failed: (false), function check_privileges, file /Users/wclee/Workspace/stanford/
projects/legion//runtime/accessor.h, line 160.
```

3. Bounds checks

Runtime Checks

- Provide warning or error messages when the application breaks runtime's assumptions

1. Disjointness checks
2. Privilege checks
3. **Bounds Checks**
   - **Check if all memory accesses fall within the logical region's bounds**
   - **Enabled by compiling runtime source with the flag `-DBOUNDS_CHECKS`**
   - **E.g.**

```
$ ./bounds
Running daxpy for 1024 elements...
Initializing field 0...
Initializing field 1...
Running daxpy computation with alpha 0.39646477...
BOUNDS CHECK ERROR IN TASK 3: Accessing invalid 1D point (1024)
Assertion failed: (false), function check_bounds, file /Users/wclee/Workspace/stanford/
projects/legion//runtime/runtime.cc, line 12368.
```

LegionSpy

- Provides various visual aids for better program understanding
  - Runtime inserts copies for data movement between tasks, which are not entirely transparent to users
  - Helpful to detect the default mapper's sub-optimal decision
    (will come to this point later with a simple example)
- Also able to check logical and physical data flows (mostly useful for runtime developers)

Steps to Run LegionSpy

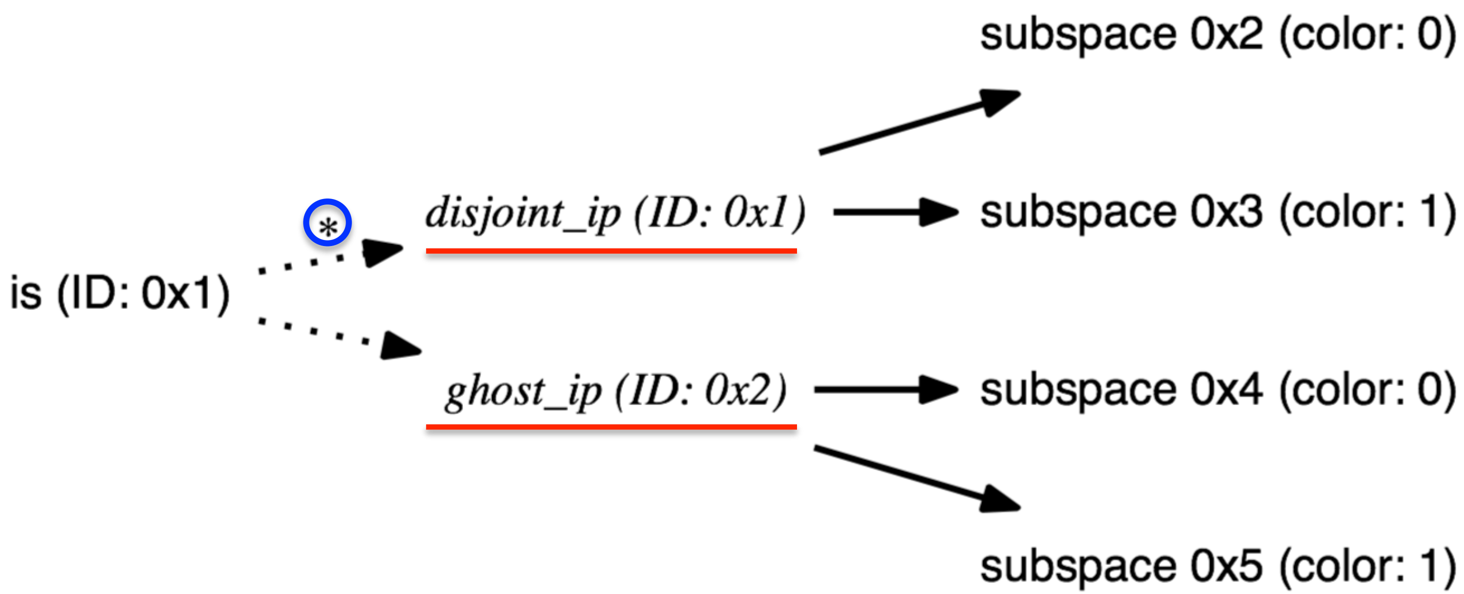
- Compile a Legion application with the `-DLEGION_SPY` `flag` added to the `CC_FLAGS` variable
- Run the application with the following flags passed on the command line: `-cat legion_spy -level 2`
  - The standard error output should be redirect to a file
- Pass the resulting log file to `legion_spy.py`

Logical and Physical Analyses

- Logical Analysis (with the `–l` flag)
  - Compares dependencies computed by the runtime (mdep) with all possible dependencies between operations (adep)
  - Warnings when mdep – adep ≠ 0 (possibly due to runtime optimizations)
  - Errors when adep – mdep ≠ 0 (possible runtime bugs, please report us!)
- Physical Analysis (with the `–c` flag)
  - Checks if each logical dependence is substantiated by actual data flow between physical instances
  - Reports missing data flows (also possible runtime bugs, please report us!)

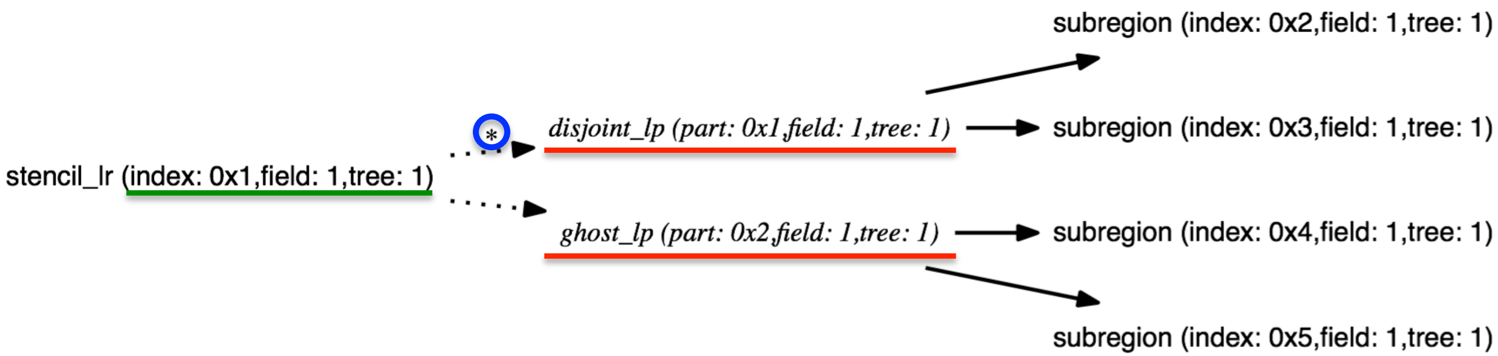
Partitioning Graphs

- Generated by passing the `-P` flag to LegionSpy
- Show index spaces and their partitions
  - Index partitions are in italics
  - Label * on edges means the index partition is disjoint



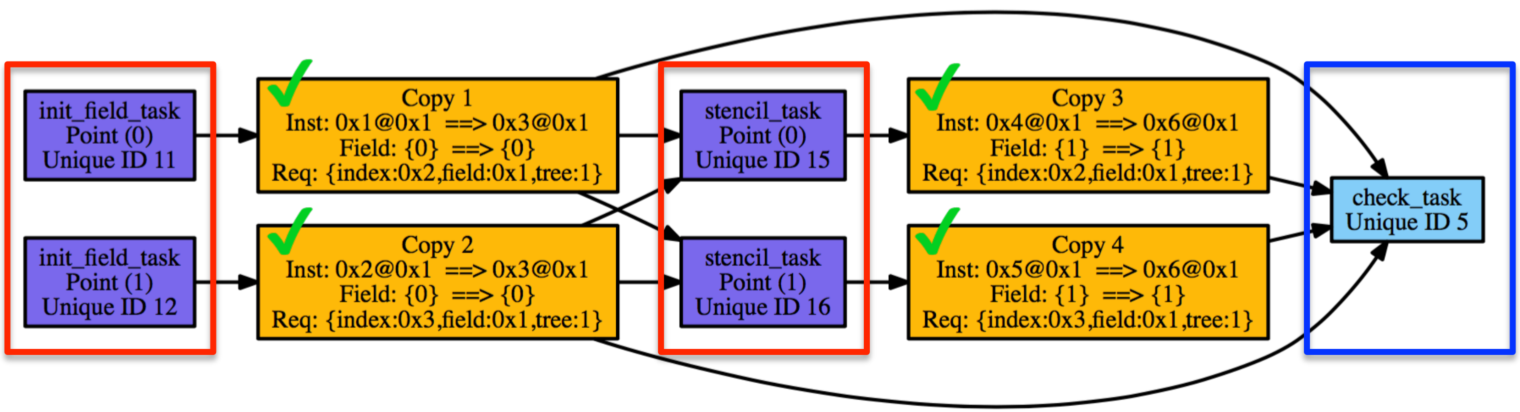
Partitioning Graphs (cont'd)

- Also show logical regions and their partitions
  - Logical partitions are in italics
  - Label * on edges means the logical partition is disjoint
  - Each region/partition is specified by index space id, field space id, and tree id



Event Graphs

- Generated by the –p flag
- Visualize operations and their dependences
  - Each box represents an operation



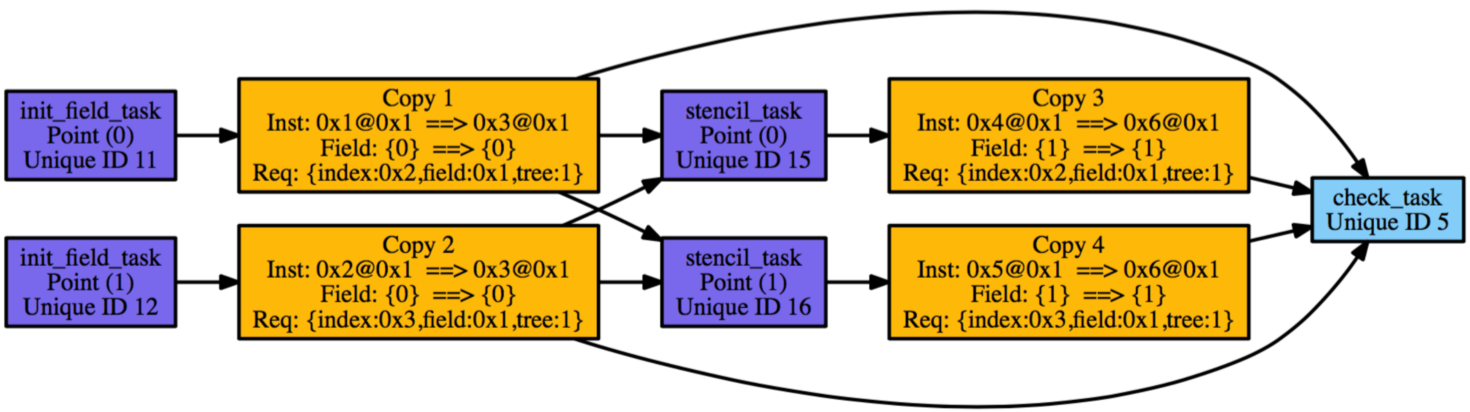
instances of index space tasks

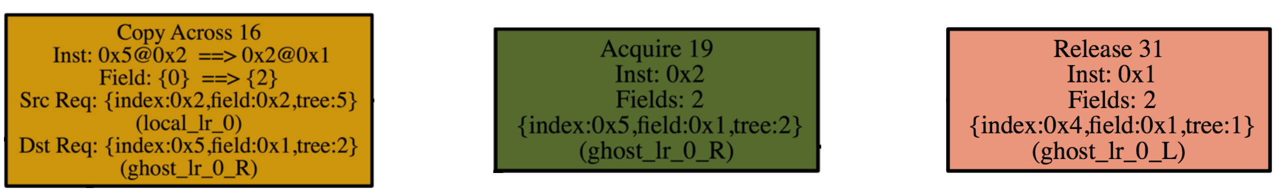
instances of single tasks

copy operations (runtime task)

Event Graphs

- Generated by the –p flag
- Visualize operations and their dependences
  - Each box represents an operation





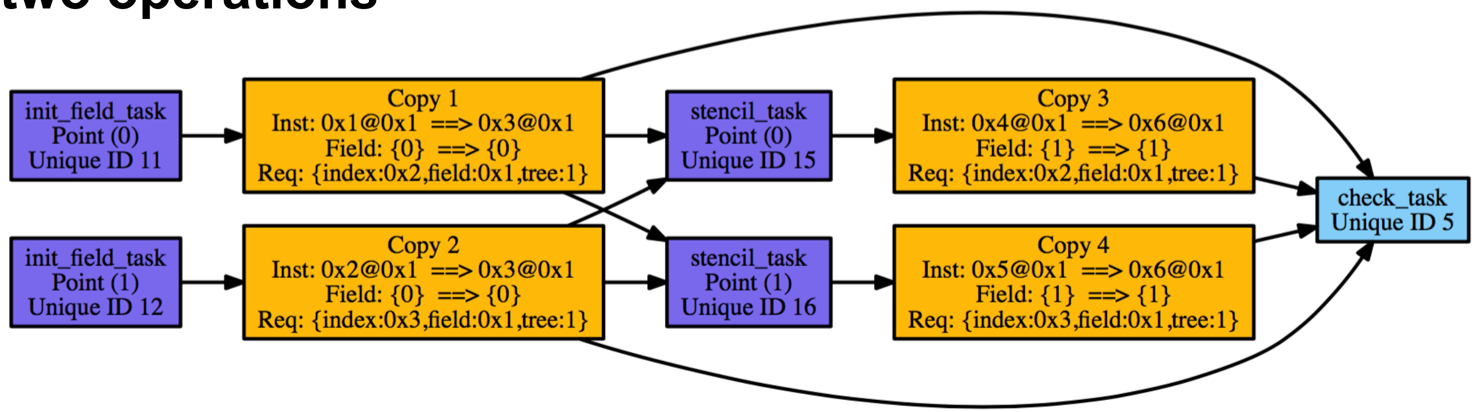
user-triggered copy operations

acquire operations

release operations

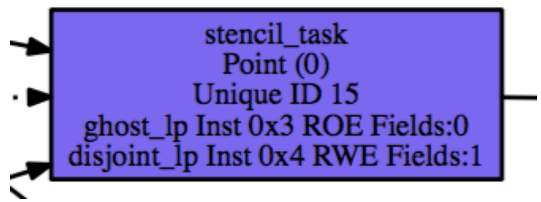
Event Graphs

- Generated by the –p flag
- Visualize operations and their dependences
  - Each box represents an operation
  - Each edge corresponds to explicit dependence between two operations



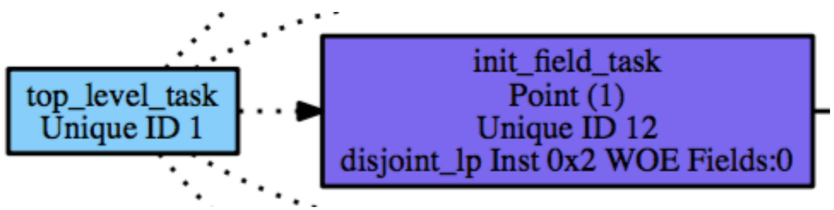
Event Graphs (cont'd)

- Passing the –v flag makes the graph also show
  - A list of accessed physical instances with the access privilege and coherence mode



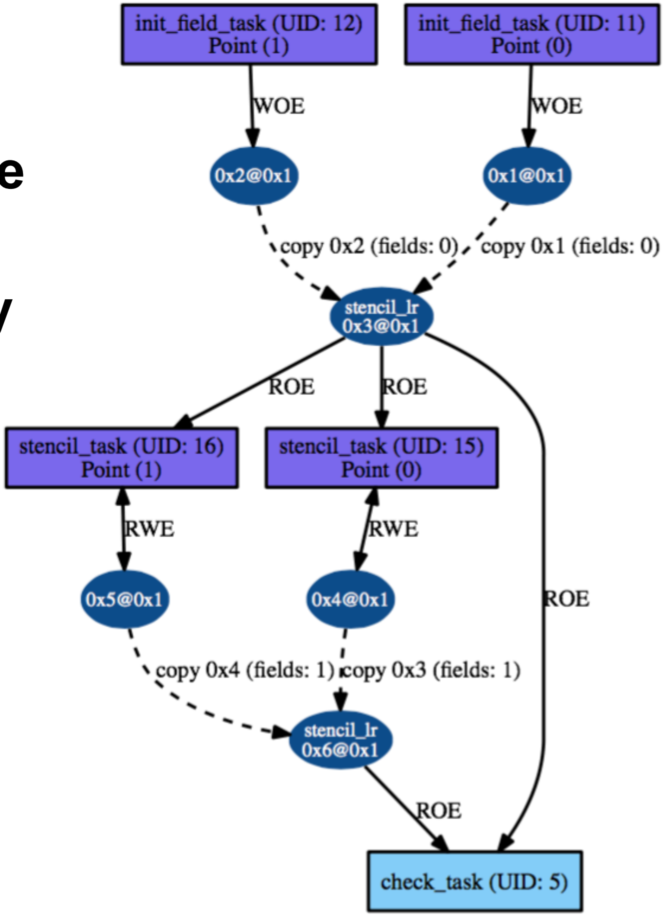
Privileges	Coherence
RO: Read-Only	E: Exclusive
WO: Write-Only	S: Simultaneous
RW: Read-Write	A: Atomic
Red: Reduction	R: Relaxed

  - Connections between parent tasks and their child operations



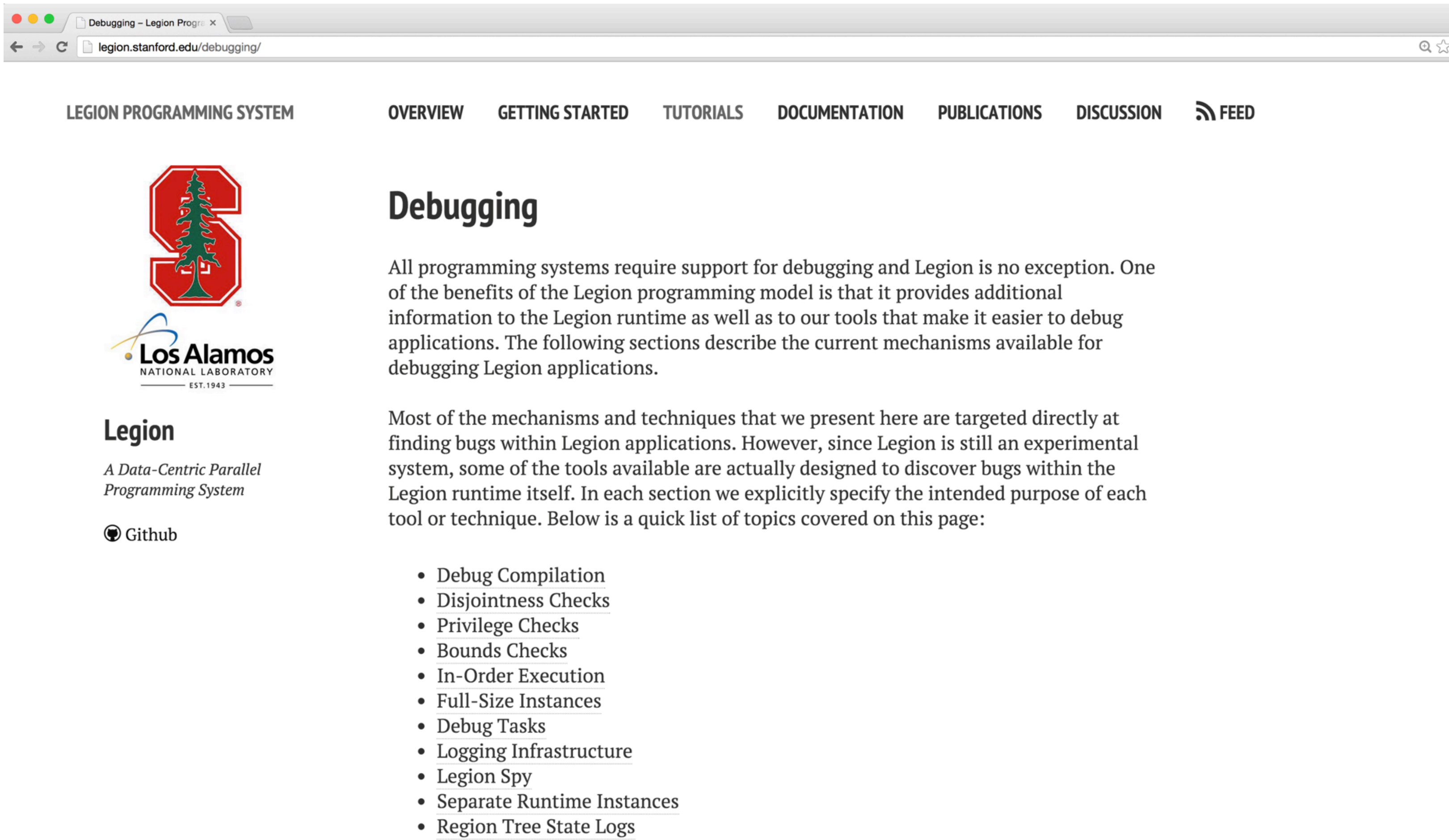
Instance Graphs (experimental)

- Generated by the `–i` flag (add `–v` for a verbose graph)
- Provide an instance-centric view of operations
  - Boxes are tasks and ovals are physical instances
  - Solid edges connect tasks with the physical instances they access
  - Dashed edges correspond to copy operations



More about Debugging

- Please visit http://legion.stanford.edu/debugging



Debugging – Legion Progr

legion.stanford.edu/debugging/

LEGION PROGRAMMING SYSTEM

OVERVIEW GETTING STARTED TUTORIALS DOCUMENTATION PUBLICATIONS DISCUSSION FEED

Los Alamos NATIONAL LABORATORY EST. 1943

Legion

A Data-Centric Parallel Programming System

Github

Debugging

All programming systems require support for debugging and Legion is no exception. One of the benefits of the Legion programming model is that it provides additional information to the Legion runtime as well as to our tools that make it easier to debug applications. The following sections describe the current mechanisms available for debugging Legion applications.

Most of the mechanisms and techniques that we present here are targeted directly at finding bugs within Legion applications. However, since Legion is still an experimental system, some of the tools available are actually designed to discover bugs within the Legion runtime itself. In each section we explicitly specify the intended purpose of each tool or technique. Below is a quick list of topics covered on this page:

- Debug Compilation
- Disjointness Checks
- Privilege Checks
- Bounds Checks
- In-Order Execution
- Full-Size Instances
- Debug Tasks
- Logging Infrastructure
- Legion Spy
- Separate Runtime Instances
- Region Tree State Logs

Profiling Tool: LegionProf

Steps to Run LegionProf

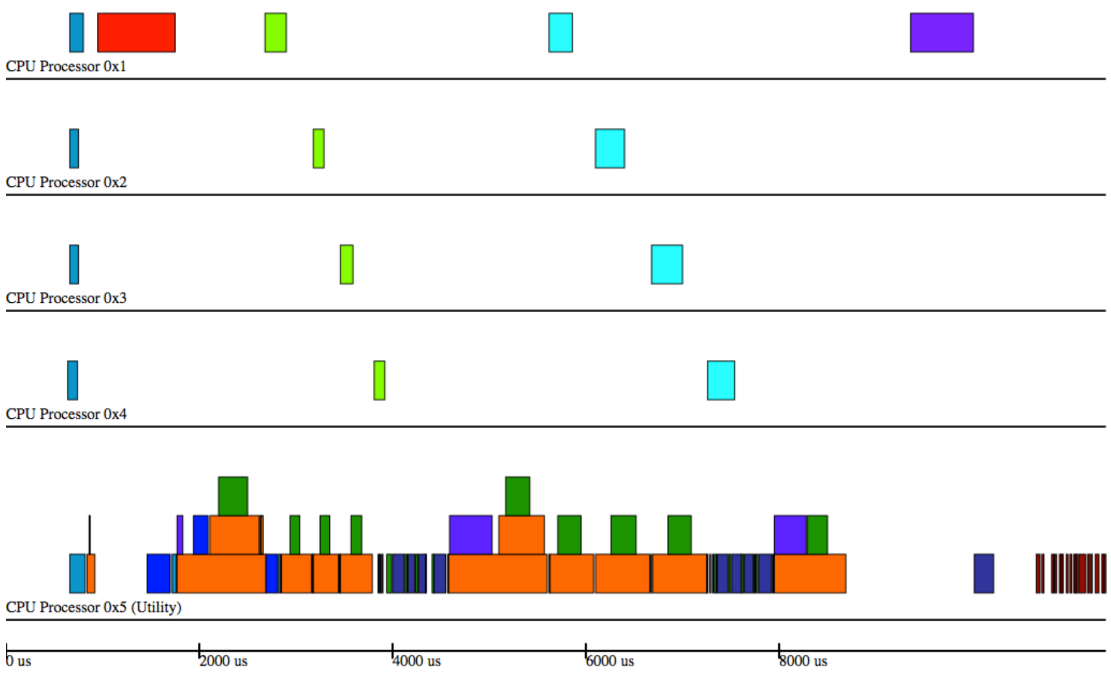
- Compile a Legion application with `-DLEGION_PROF` added to the `CC_FLAGS` variable

- Run the application with the following flags passed on the command line: `-cat legion_prof -level 2`
  - The standard error output should be redirected to a file
  - You can also pass `-hl:prof` <int> to specify a node to get the profile result

- Pass the resulting log file to `legion_prof.py`

Timeline Output

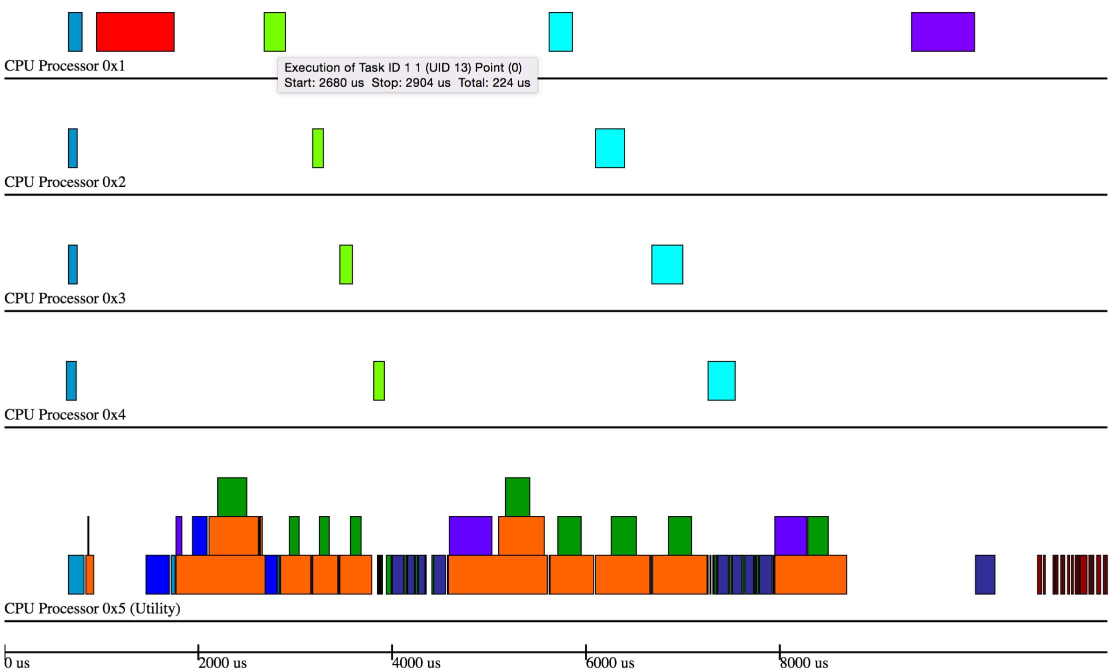
- Generated by passing the –p flag to LegionProf
- Shows which tasks ran on which processors at what time and for how long



http://legion.stanford.edu

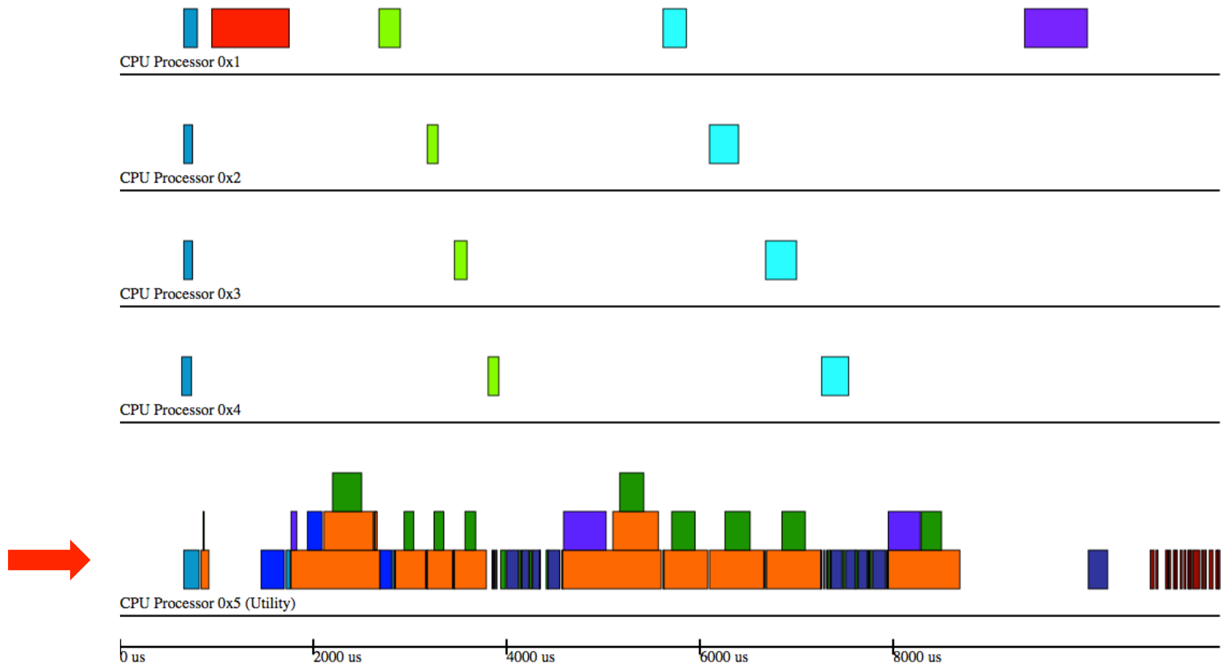
Timeline Output

- Generated by passing the –p flag to LegionProf
- Shows which tasks ran on which processors at what time and for how long
  - Task id and timestamps appear when hovering a bar



Timeline Output

- Generated by passing the –p flag to LegionProf
- Shows which tasks ran on which processors at what time and for how long
  - The last few lines are for utility processors where runtime tasks run



Runtime Statistics

- Additionally LegionProf prints out the statistics of
  - How long each processor was active

```
*************************************************
   PROCESSOR STATS
*************************************************
CPU Processor 0x1
    Total time: 11388 us
    Active time: 2078 us (18.247%)
    Application time: 1932 us (16.965%)
    Meta time: 146 us (1.282%)
```

  - How many instances were created on each memory

```
*************************************************
   MEMORY STATS
*************************************************
Memory 0x1
    Total Instances: 2
Memory 0x2
    Total Instances: 5
```

  - How often a task was invoked and how long it was running

```
 -------------------------
  Task Statistics
  -------------------------
    Task ID 2 2                             4532 us (7.959%)
          Executions (APP):
                Total Invocations: 4
                Cummulative Time: 1160 us (2.037%)
                Non-Cummulative Time: 1160 us (2.037%)
                Average Cum Time: 290.000 us
                Average Non-Cum Time: 290.000 us
          Meta Execution Time (META):
                Cummulative Time: 5315 us (9.334%)
                Non-Cummulative Time: 3372 us (5.922%)
```

Performance Tuning

- Configuring runtime parameters
- Writing a custom mapper
  - Point of mapping interface is to decouple the program correctness from the performance
- Changing the actual application code

More about Profiling and Tuning

- **Please visit http://legion.stanford.edu/profiling**

Performance Profiling and ×

legion.stanford.edu/profiling/

LEGION PROGRAMMING SYSTEM OVERVIEW GETTING STARTED TUTORIALS DOCUMENTATION PUBLICATIONS DISCUSSION FEED

Legion

A Data-Centric Parallel Programming System

Github

Performance Profiling and Tuning

After developing a functional Legion application, it is usually necessary to performance profile and tune the application for high performance. This page covers many of the techniques required for achieving high performance for Legion applications. Below is a list of topics covered on this page.

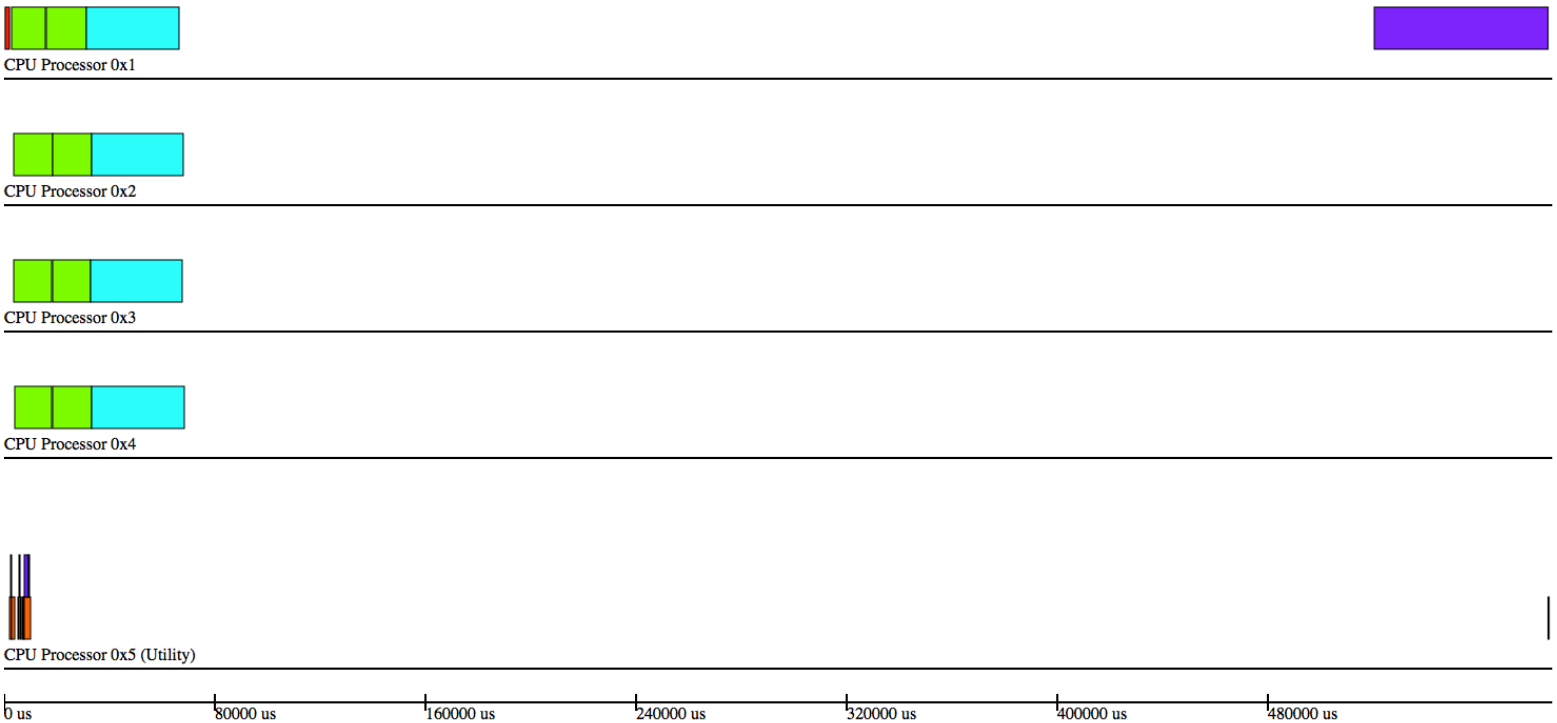
High Performance Low-Level Runtime

All Legion applications which are going to be run for performance should use the *general* low-level runtime which is capable of running on large clusters. This version of our low-level runtime is the only one that has been tuned for performance. The shared-memory-only version of the low-level runtime has **not** been performance tuned and therefore should never be used for performance experiments. When using our normal Legion Makefiles, the general low-level runtime is selected by setting the Makefile variable `SHARED_LOWLEVEL=0` . When using the general low-level runtime, users should also modify

Performance Tuning Example: Eliminating unnecessary copies

Profiling DAXPY Application

- DAXPY: Double precision A times X Plus Y



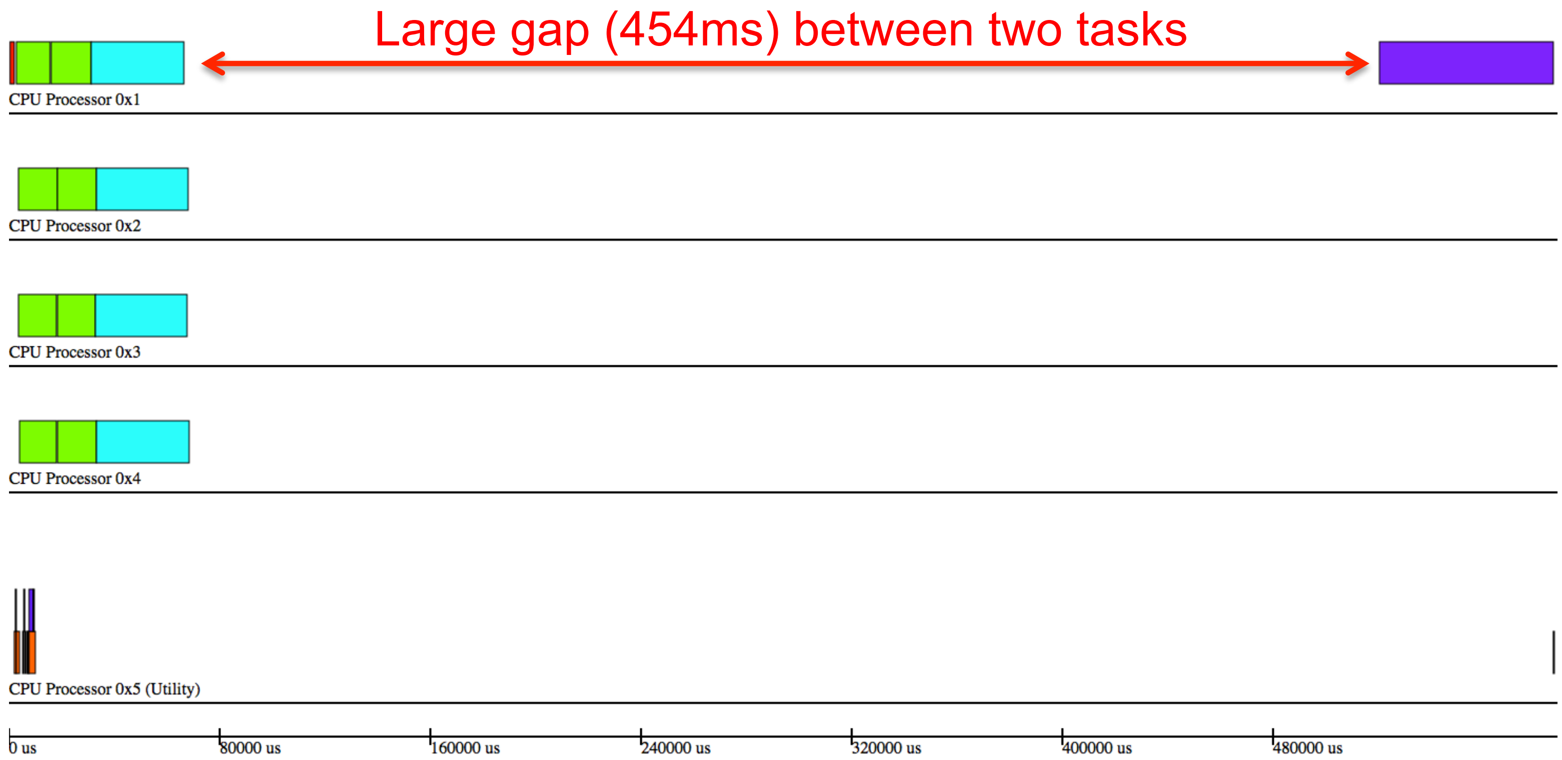
init_field task: initializes X and Y

daxpy task: calculates AXPY

check_task: verifies the calculation result

Profiling DAXPY Application

- DAXPY: Double precision A times X Plus Y

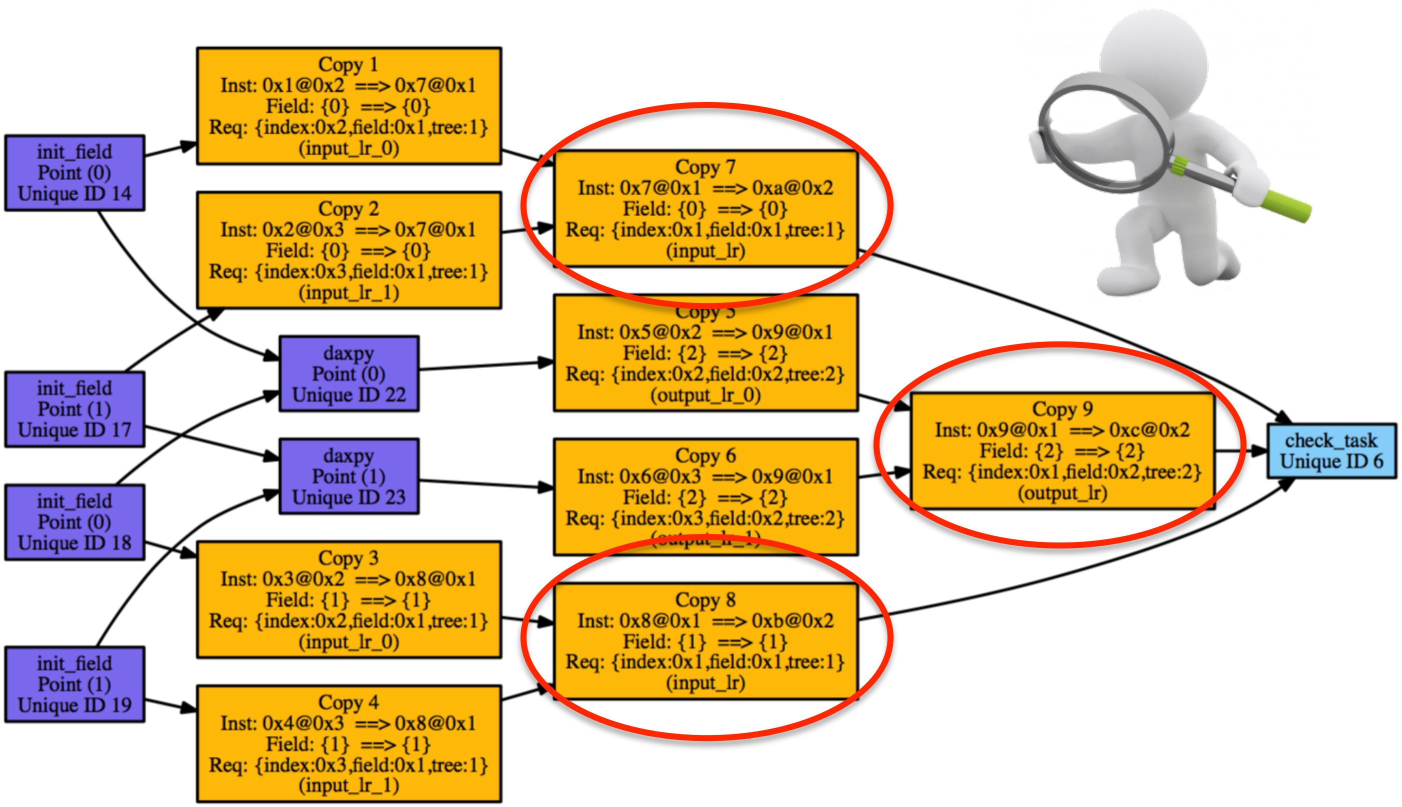


init_field task: initializes X and Y

daxpy task: calculates AXPY

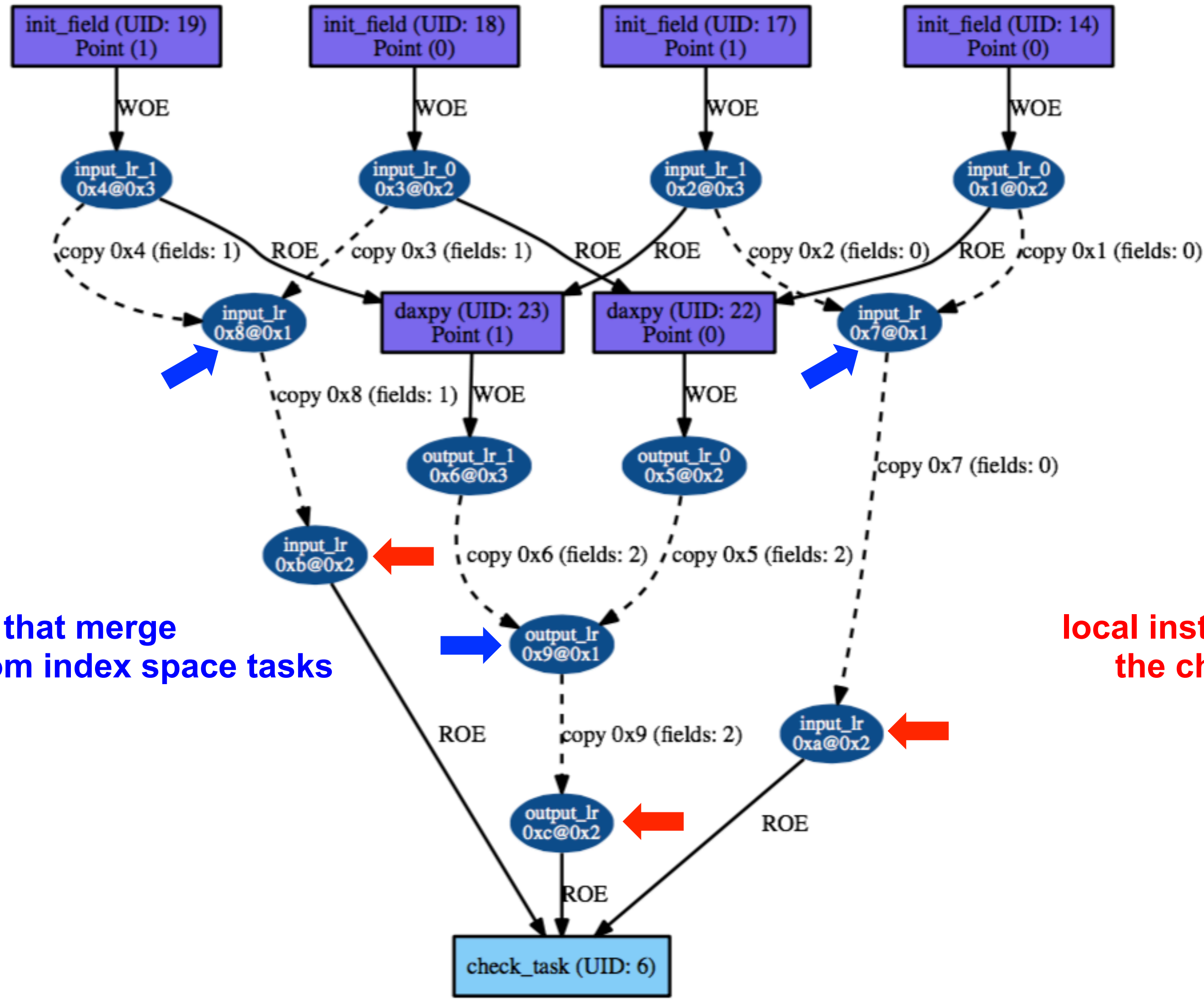
check_task: verifies the calculation result

Detecting Unnecessary Copies from Event Graph



Copy operation 7, 8, and 9 are unnecessarily making local copies to memory 0x2

Instance Graph of DAXPY



instances that merge results from index space tasks

local instances for the check_task

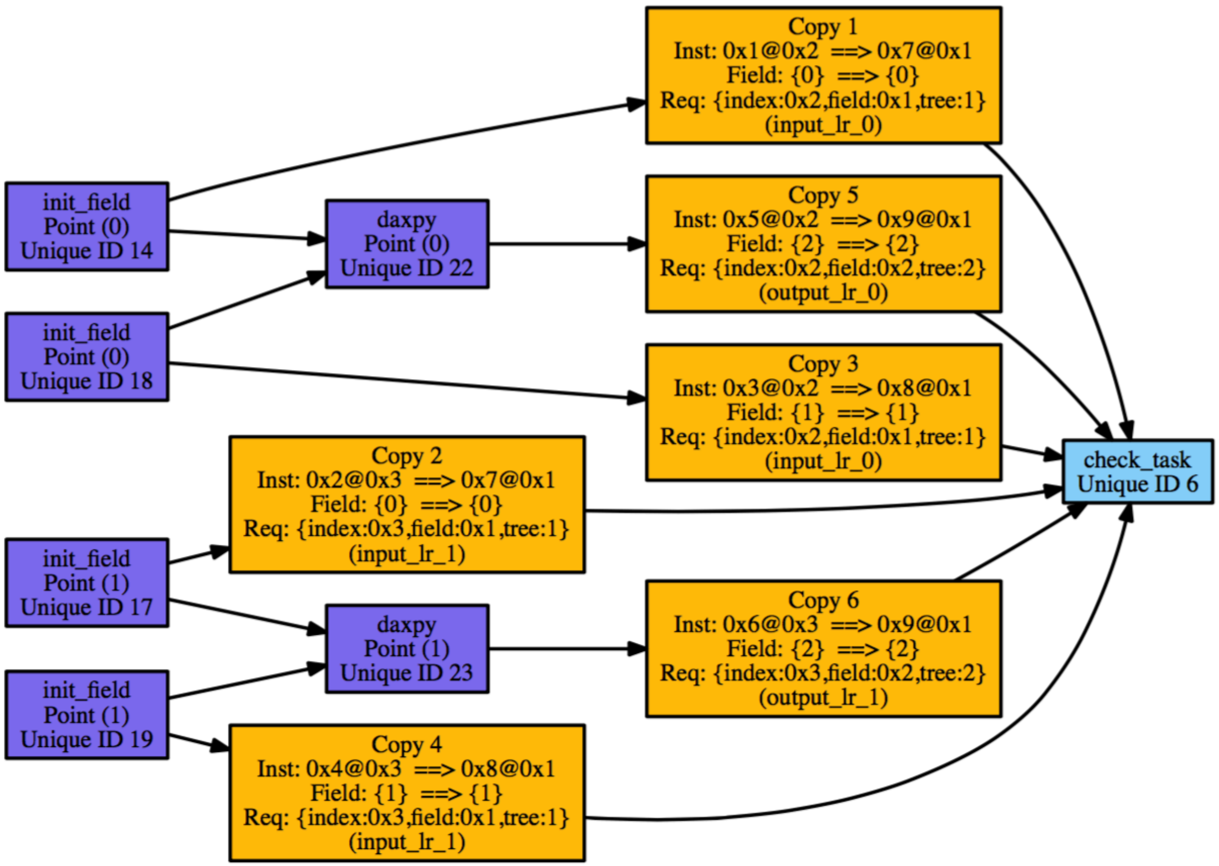
Example Mapper for Avoiding Unnecessary Copies

- **Make the check_task use directly the instances on memory 0x1 (`Memory::SYSTEM_MEM`)**

```
bool DAXPYMapper::map_task(Task *task)
{
  if (task->task_id == CHECK_TASK_ID) {
    const set<Memory> &vis_mems =
      machine->get_visible_memories(task->target_proc);
    for (unsigned idx = 0; idx < task->regions.size(); idx++) {
      for(set<Memory>::iterator it = vis_mems.begin();
          it != vis_mems.end(); ++it) {
        if (machine->get_memory_kind(*it) == Memory::SYSTEM_MEM)
        {
          task->regions[idx].target_ranking.push_back(*it);
          break;
  ...
```

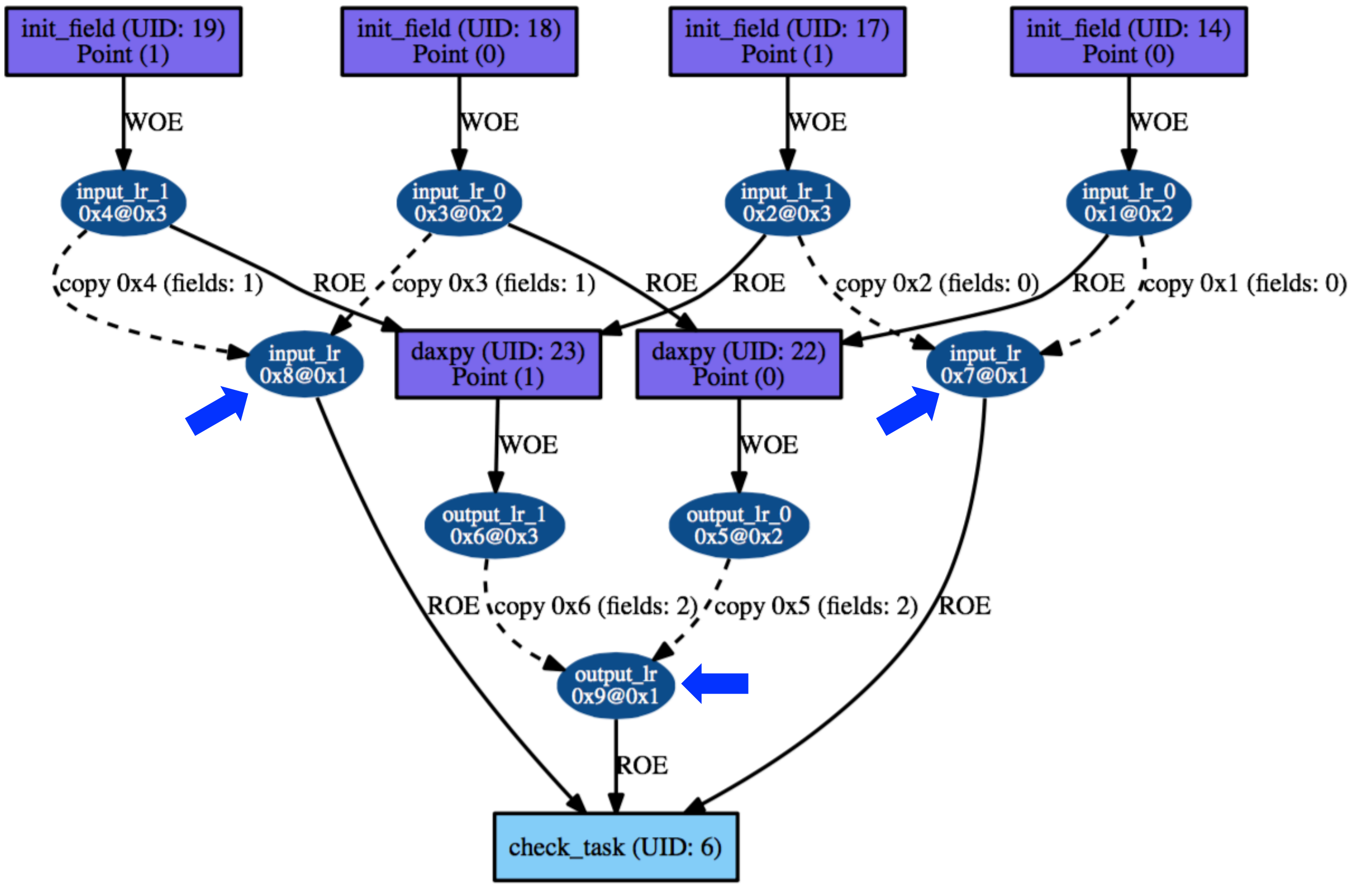
Changes in Event Graph

- Copy operation 7, 8, and 9 disappeared



Changes in Instance Graph

- **We can verify that the check_task is using directly the instances on the memory 0x1**



Changes in Timeline

Questions?