Hedgehog Static Analysis and Multi-node Applications on Heterogenous Systems

Alexandre Bardakoff, Bruno Bachelet, Timothy Blattner, Walid Keyrouz, Loïc Yon
Martin Berzins, John Holmen, **Nitish Gangadhar Shingde**

NIST
National Institute of Standards and Technology
U.S. Department of Commerce

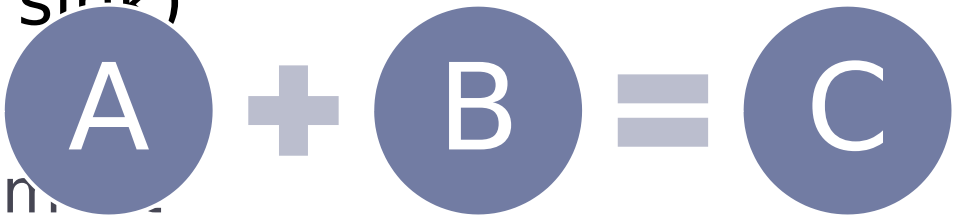
ITL
INFORMATION TECHNOLOGY LABORATORY

Overview

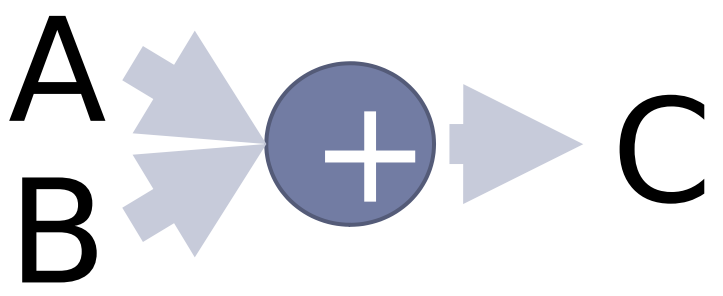
- Parallel library for **coarse-grained parallelism**
- Based on a **data-flow** graph model
- Using **data pipelining** to get performance
- **No scheduler**
  - Relies solely on OS to manage threads
  - Execution triggered only on the presence of data
- **Header only** library
  - V.1 C++17
  - V.2 C++20

Data-flow graph

- Directed graph (circles accepted!)
- entry and exit points (source and sink)
- Components:
  - Nodes: computation or state management
  - Edges: directed information flow
- When graph = node ⇒ Composability!
- Coarse-grained decomposition
  - Node = execution kernel
    - Fine-grained parallelism with specialized library



Addition algorithm

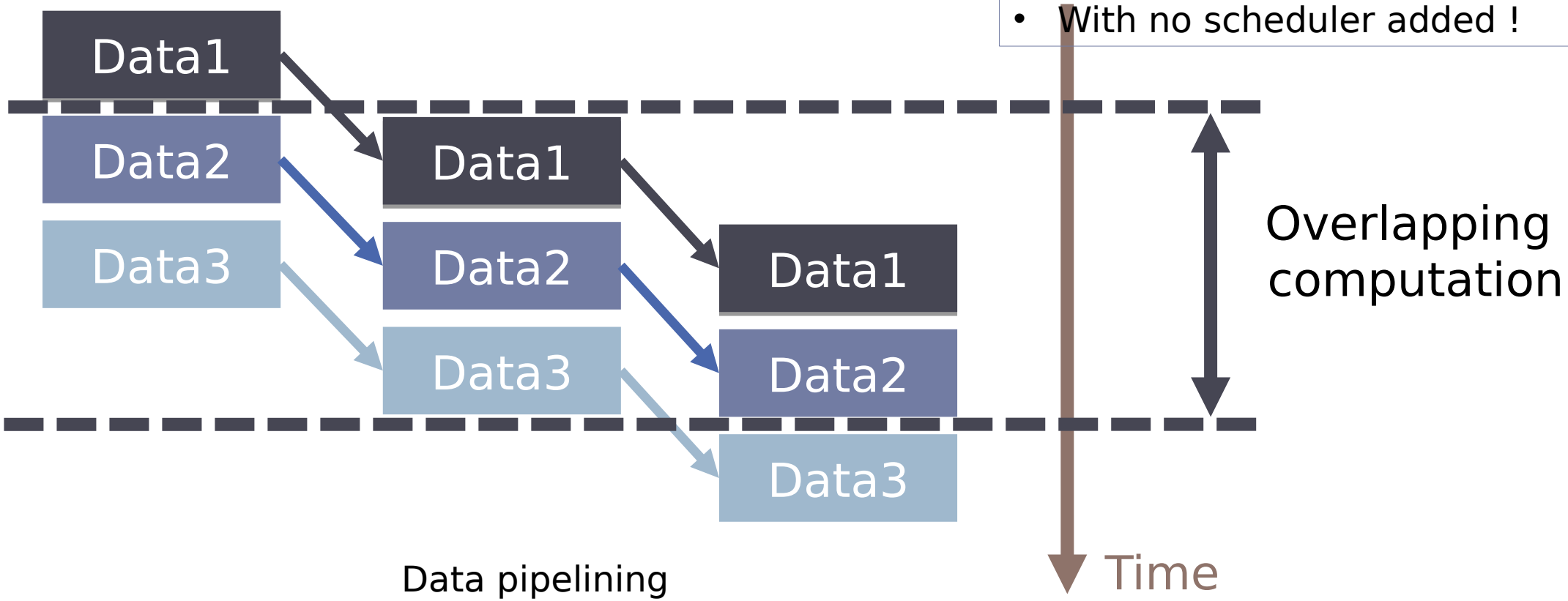


Data-flow representation

Data pipelining

Stage starts as soon as data become available

- Asynchronous behavior
- With no scheduler added !



Data pipelining representation

Securing with metaprogramming techniques

Why?

Why securing ?

- Few restrictions in the library
- Graph construction rules
- Directed cycles authorized
  - Possible deadlocks
- Data broadcast
  - Possible data races

Why static metaprogramming ?

- A way to do computation during compilation
- Better to detect problems as early as possible
  - During compilation
  - While writing code!

Goals

Conformity checks with Template Metaprogramming

- **Simplest** possible API
- **Secured** API
  - Checks **compatibility**
  - Does not allow compilation if ill-constructed graph
- To enforce method existence without virtuality

Static graph with constant expressions

- **Compile-time** (or “static”) representation of a Hedgehog graph
  - Simple
  - To hold basic information about the graph’s **structure**
  - Because of technical limitations
- Defines / applies **test(s)**
  - To check for valid & well-formed graph
  - To detect possible runtime issues
- **Converts** a compile-time graph to a **runtime** (or “dynamic”) graph

Metaprogramming techniques

Template MetaProgramming (TMP)

- Usage of templates
- Computation on types (Turing Complete)
  - Definition of metafunctions
  - Usage of traits + helper
- C++20: concepts (+constraints) added
  - Defines “type categories”

Metaprogramming with constant expressions

- Usage of (mainly) “constexpr” modifier
- Defines variable, class and method usable at compile-time (computation on values)
- Only a subset of C++ is available

Conformity check

Conformity check

Metafunction

- class / struct with 0+ template parameters & 0+ return types and values
- Idiomatic use / not part of the language
- Enable_if construct based on SFINAE

```
template <class T1, class T2>
struct CheckAddable {
  constexpr static bool value =
  std::is_arithmetic_v<T1> &&
  std::is_arithmetic_v<T2> &&
  std::is_same_v<T1, T2>;
};

template <
  class T1, class T2,
  class = std::enable_if_t<CheckAddable<T1, T2>::value>
>
T1 add(T1 const & val1, T2 const & val2){
  return val1 + val2;
}
```

Concepts

- Define “type category”
- Concepts = named requirements
- Not evaluated, only validated

```
template <class T>
concept Addable = requires(T v){
  { v + v };
  std::is_arithmetic_v<T>;
};


template <Addable T1, Addable T2>
T1 add(T1 const & val1, T2 const & val2){
  static_assert(
   std::is_same_v<T1, T2>,
   "T1 & T2 should be the same type.");
  return val1 + val2;
}
```

Real example: input method

With Template Metaprogramming (HH V.1):

```
template<
  class UserDefinedMultiReceiver,
  class InputsMR = typename UserDefinedMultiReceiver::inputs_t,
  class InputsG = typename behavior::MultiReceivers<GraphInputs...>::inputs_t,
  class isMultiReceiver = typename std::enable_if_t<
    std::is_base_of_v<typename helper::HelperMultiReceiversType<InputsMR>::type, UserDefinedMultiReceiver>>,
  class isInputCompatible = typename std::enable_if_t<traits::is_included_v<InputsMR, InputsG>>>
void input(std::shared_ptr<UserDefinedMultiReceiver> input){}
```

```
No matching member function for call to 'input'
candidate template ignored: requirement 'traits::is_included_v<std::tuple<double>, std::tuple<int, float>>' was not satisfied [with Node = TestTask, InputsN = std::tuple<double>, InputsG = std::tuple<int, float>, isMultiReceiver = void]
```

Error message produced by Clion IDE

With concepts (HH V.2):

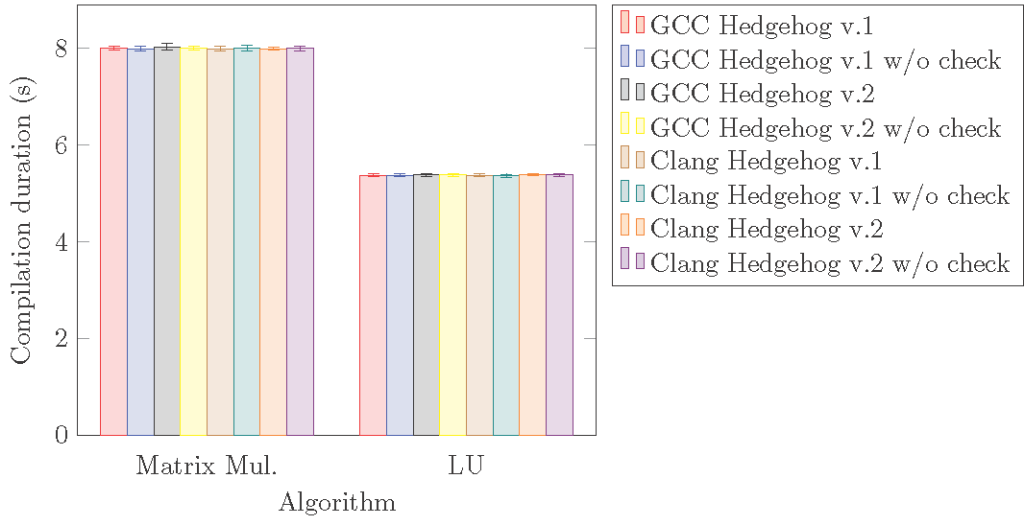
```
template<HedgehogMultiReceiver UserDefinedInput>
void input(std::shared_ptr<UserDefinedInput> input) {
  using NodeInputs = typename UserDefinedInput::inputs_t;
  using NodeMR = typename helper::HelperMultiReceiversType<NodeInputs>::type;
  static_assert(
    traits::is_included_v<NodeInputs, typename behavior::MultiReceivers<GraphInputs...>::inputs_t>,
    "The input node should share at least one input type with the graph.");
  //…}
```

```
In template: static_assert failed due to requirement 'traits::is_included_v<std::tuple<double>, std::tuple<int, float>>' "The input node should share at least one input type with the graph."
error occurred here
in instantiation of function template specialization 'hh::Graph<int, int, float>::input<TestTask>' requested here
```

Error message produced by Clion IDE

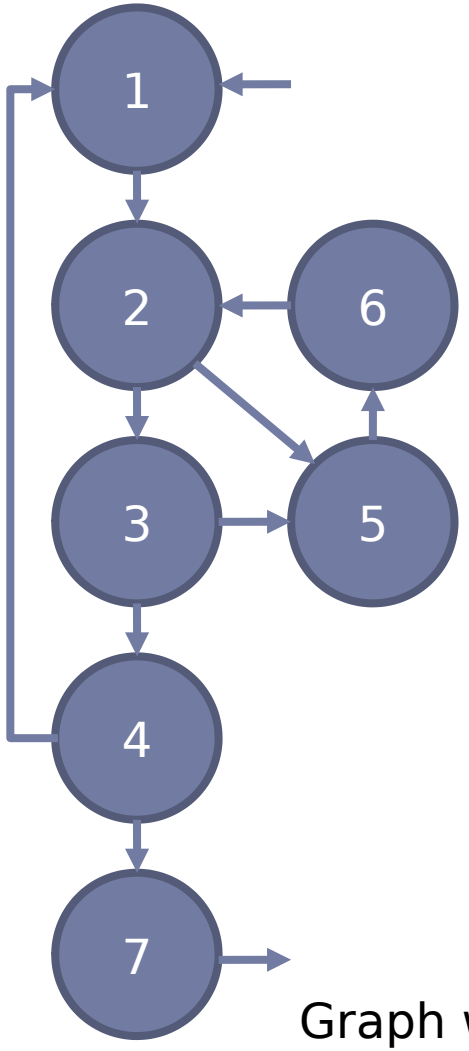
Cost of conformity checks

Compilation time with and without conformity checks



Static graph

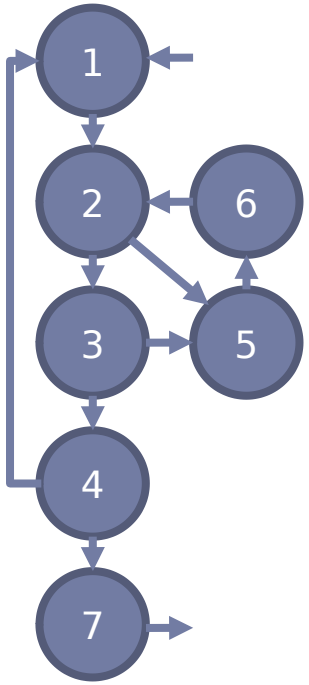
Static graph: example graph



Graph with potential problems

- Directed cycles:
- Possible data races:

Creation of a compile-time graph



raph with potential problems

```
constexpr hh::cx::CXNode<TaskIntInt> node1("Task1");
constexpr hh::cx::CXNode<TaskIntInt> node2("Task2");
constexpr hh::cx::CXNode<TaskIntInt> node3("Task3");
constexpr hh::cx::CXNode<TaskIntInt> node4("Task4");
constexpr hh::cx::CXNode<TaskIntInt> node5("Task5");
constexpr hh::cx::CXNode<TaskIntInt> node6("Task6");
constexpr hh::cx::CXNode<TaskIntInt> node7("Task7");


constexpr auto defroster = [&]() {
  hh::cx::CXGraph<GraphIntInt> g(
  "Graph with multiple cycles and data races");


  g.input(node1);
  g.addEdge(node1, node2);
  g.addEdge(node2, node3);
  g.addEdge(node3, node4);
  g.addEdge(node4, node7);
  g.addEdge(node4, node1);
  g.addEdge(node3, node5);
  g.addEdge(node5, node6);
  g.addEdge(node6, node2);
  g.addEdge(node2, node5);
  g.output(node7);
  //...
}();
```

Instantiate of constexpr nodes. They represent dynamic nodes of type “TaskIntInt”

Instantiate a constexpr graph representing a dynamic graph of type “GraphIntInt”.

Build a compile-time graph with an API similar to Hedgehog’s dynamic one.

Constexpr lambda that creates a constexpr graph.

Adding tests on the graph and Defroster creation

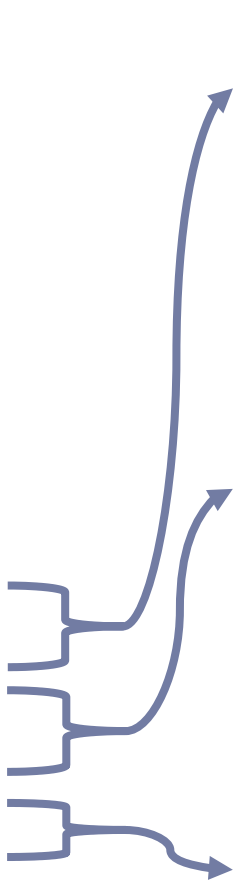
```
constexpr hh::cx::CXNode<TaskIntInt> node1("Task1");
constexpr hh::cx::CXNode<TaskIntInt> node2("Task2");
constexpr hh::cx::CXNode<TaskIntInt> node3("Task3");
constexpr hh::cx::CXNode<TaskIntInt> node4("Task4");
constexpr hh::cx::CXNode<TaskIntInt> node5("Task5");
constexpr hh::cx::CXNode<TaskIntInt> node6("Task6");
constexpr hh::cx::CXNode<TaskIntInt> node7("Task7");

constexpr auto defroster = [&]() {
  hh::cx::CXGraph<GraphIntInt> g(
    "Graph with multiple cycles and data races");
  g.input(node1);
  g.addEdge(node1, node2); g.addEdge(node2, node3);
  g.addEdge(node3, node4); g.addEdge(node4, node7);
  g.addEdge(node4, node1); g.addEdge(node3, node5);
  g.addEdge(node5, node6); g.addEdge(node6, node2);
  g.addEdge(node2, node5); g.output(node7);

  auto cycle = hh::cx::test::CycleTest<GraphIntInt>{};
  auto constTest = hh::cx::test::DataRaceTest<GraphIntInt>{};

  g.addTest(cycle);
  g.addTest(constTest);

  return hh::cx::Defroster(g);
}();
```



Instantiate graph tests.

- First test = deadlocks
- Second test = data races

Add the tests on the static graph

Defroster:

- Runs the tests at construction
- Gather static graph information (structure to create dynamic graph)

Defining a test “as usual”

```
template<class GraphType, size_t NodesNumber = 20>
class MyTest: public hh::cx::CXAbstractTest<GraphType, NodesNumber> {
 private:
  double value_ = 0;

 public:
  constexpr MyTest(double const value)
      : hh::cx::CXAbstractTest<GraphType, NodesNumber>(”My test !"), value_(value) {}

  constexpr ~MyTest() override = default;

  constexpr void test(hh::cx::CXGraph<GraphType, NodesNumber> const *graph) override {
    // […]
  }
};
```

Defroster usage

```
constexpr auto defroster = [&]() { //...  }();

static_assert(
defroster.isGraphValid(),
"The Graph is not valid.");

//OR

if constexpr(!defroster.isGraphValid()){
std::cout << defroster.report() << "\n";
} else {/*...*/}
```

Because there are cycles and data races

Compilation stops with the following error message: “The Graph is not valid.”

Compilation succeeds. Executable will only contain the defroster object and the “defroster.report()” call. The execution will only be:

```
In graph Graph with multiple cycles and data races :
      Johnson: Cycles found; the canTerminate() method needs
 to be defined for each of these cycles.
       Task1  ->  Task2  ->  Task3  ->  Task4  ->  Task1
       Task2  ->  Task3  ->  Task5  ->  Task6  ->  Task2
       Task2  ->  Task5  ->  Task6  ->  Task2
      Constness test: Potential data races found between these
 nodes:
       Task2 -> Task3 / Task5
       Task3 -> Task4 / Task5
       Task4 -> Task1 / Task7
```

Conversion of a compile-time graph

```
constexpr hh::cx::CXNode<TaskIntInt> node0(”Node0");
constexpr hh::cx::CXNode<TaskIntInt> node1(”Node1");
constexpr hh::cx::CXNode<TaskIntInt> node2(”Node2");
constexpr hh::cx::CXNode<TaskIntInt> node3(”Node3");
constexpr hh::cx::CXNode<TaskIntInt> node4(”Node4");
constexpr hh::cx::CXNode<TaskIntInt> node5(”Node5");
constexpr hh::cx::CXNode<TaskIntInt> node6("Node6");
constexpr hh::cx::CXNode<TaskIntInt> node7("Node7");

constexpr auto defroster = [&]() { […] }();

if constexpr(defroster.isValid()){
  auto task0 = std::make_shared<TaskIntInt>("Task0");
  auto task1 = std::make_shared<TaskIntInt>("Task1");
  auto task2 = std::make_shared<TaskIntInt>("Task2");
  auto task3 = std::make_shared<TaskIntInt>("Task3");
  auto task4 = std::make_shared<TaskIntInt>("Task4");
  auto task5 = std::make_shared<TaskIntInt>("Task5");
  auto task6 = std::make_shared<TaskIntInt>("Task6");
  auto task7 = std::make_shared<TaskIntInt>("Task7");

  auto graph = defroster.convert(
    node0, task0,
    node1, task1,
    node2, task2,
    node3, task3,
    node4, task4,
    node5, task5,
    node6, task6,
    node7, task7
  );

  graph->executeGraph();
  graph->finishPushingData();
} graph->waitForTermination();
```

Instantiate the CXNodes

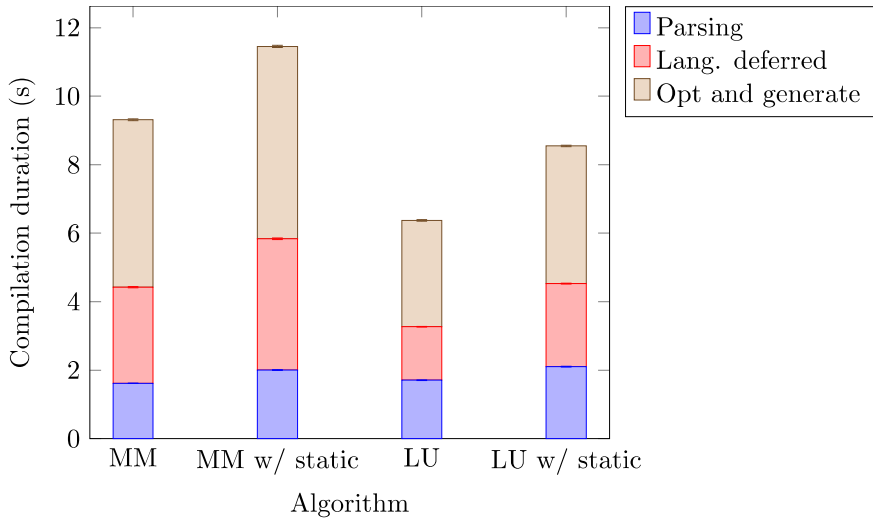
Creates a defroster from the CXGraph

Instantiate the dynamic nodes

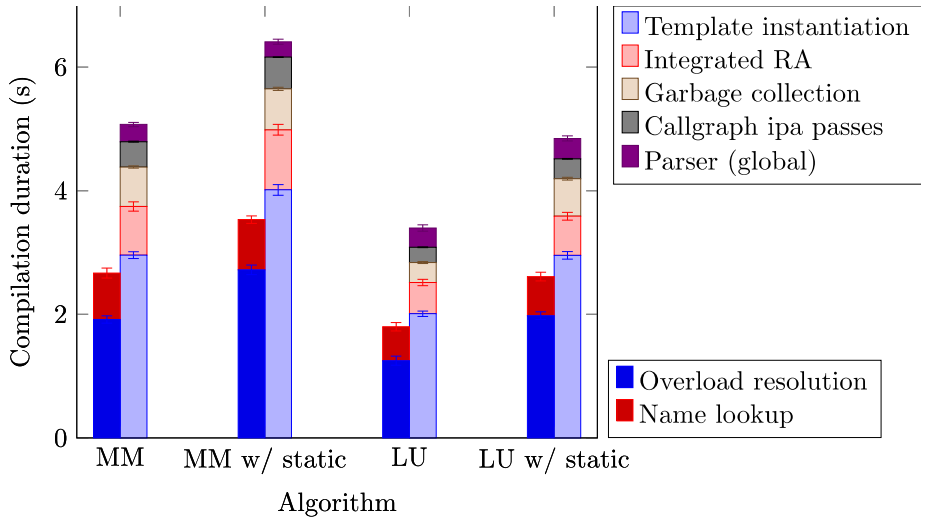
Convert the compile-time graph into a dynamic graph by mapping the CXNodes to the dynamic nodes by pairs.

Use the dynamic graph as usual

Cost of static analysis



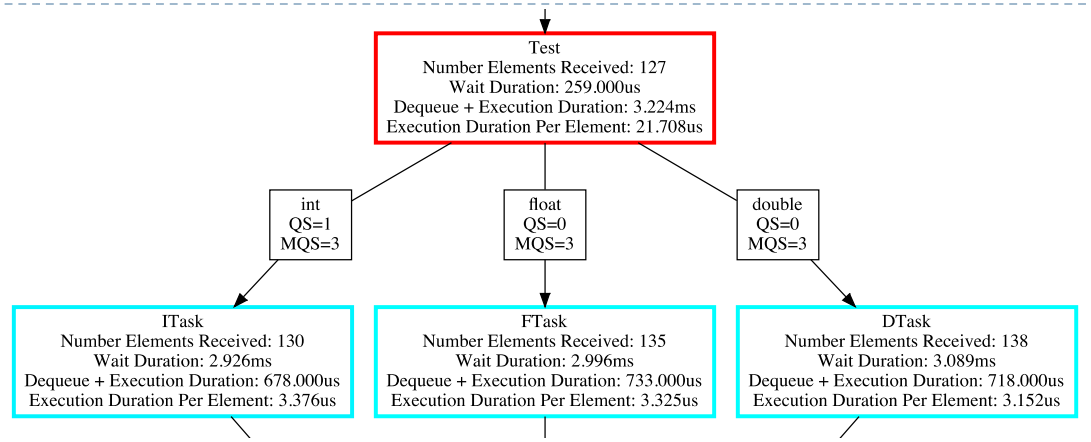
Compilation phases with and without static graph



Compilation steps with and without static graph

What's next in HH v.3 ?

- Multiple outputs
- Cleaning step
- Reworked architecture
  - Same (lower?) latency
  - Easier extensibility
  - Compile time change ?



Dot representation of a graph showcasing multiple outputs

Edges creation for double type

```
g.addEdge<int>(testNode,
intNode);
g.addEdge<float>(testNode,
floatNode);
g.addEdge<double>(testNode,
doubleNode);
```

Example of execute method

```
void execute(std::shared_ptr<int> data)
override {
  this->addResult(data);
  this-
>addResult(std::make_shared<float>(*dat
a));
  this-
```

Thank you !