Container Class Annotations in C++ Improve the Capability of Static Analysis in MLIR

Ehsan Amiri, Rouzbeh Paktinatkeleshteri, Hao Jin – Huawei Technologies Canada

Eric Wang – University of Waterloo

Jose Nelson Amaral – University of Alberta

www.huawei.com

HUAWEI

What is a Container?

- Containers are specified in the C++ standard as part of C++ standard library and have certain requirements. (Quotes from n4713).
  - *“Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, **insert** and **erase** operations.”*
- The standard is also provides very specific information on member function and member variables:
  - *“ The member function size() returns the number of elements in the container.”*
  - On erase(q) function: *“Effects: Erases the element pointed to by q”.*
- This is text of the standard, not something that can be used by the compiler.

Is it useful if compiler knows (standard library) containers?

- An MLIR compiler for C++ would be able to recognize C++ idioms and optimize them, among other things by recognizing standard library containers.
- Examples from relevant talks in LLVM Dev 2020 and LLVM Dev 2023.

C++ idioms - semantics transforms based on C++ specific dialects

- e.g. replace `std::map[k] = v` with `std::map::insert(k,v)` to avoid unnecessary default construction

```
std::map<K, V> Map;
Map[k] = v;
```

```
auto I = Map.lower_bound(k);
if ((I != Map.end()) && (k == I->first))
   I->second = v;
else
   Map.insert(I, {k, v});
```

STL Optimizations using CIL (1)

```
void func() {
  std::vector<int> vec;
  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);

  vec.insert(vec.end(),{4,5,6});
}
```

```
void func() {
  std::vector<int> vec;
  vec.insert(vec.end(),{1,2,3});
  vec.insert(vec.end(),{4,5,6});
}
```

How the code will look like?

```
template< <class T, class Allocator>
[[container, vector]] class ElementsArray{

   protected:
   [[memory]] T *elements;
   [[size]] unsigned int used_count;
   [[capacity]] unsigned allocated_mem;

   public:
   [[insert]] void insert(...)

};
```

[[container]]: This class has an exclusive data member that is allocated and deallocated only in member functions of this class.

[[vector]]: The memory is a contiguous list of elements of a specific data type. Only specific member functions of this class can add new element or remove an existing element.

[[memory]]: The memory that is allocated/deallocated by the class.

[[size]]: Number of valid elements of the specific data type in the allocated memory.

[[capacity]]: Total size of allocated memory

[[insert]]: Insertion of new elements happens exclusively in this function.

How the code will look like?

```
template< <class T, class Allocator>
class ElementsArray{

   protected:
   [[container, vector]] T *elements;
   [[size]] unsigned int used_count;
   [[capacity]] unsigned allocated_mem;

   public:
   [[insert]] void insert(...)

};
```

- Advantage: A more complex class might contain a container for its own use.

Example 1

- The example is taken from an open source DBMS.

```
for (size_t onexpr_idx = 0; onexpr_idx < added_columns.join_on_keys.size(); ++onexpr_idx)
```

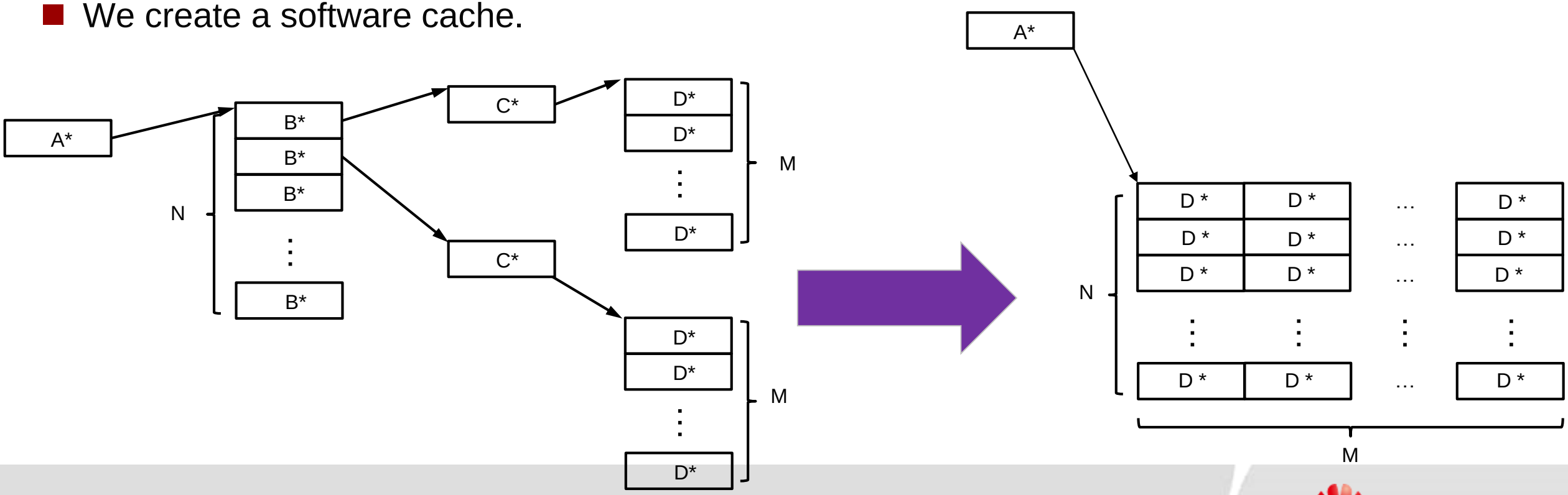
- Here we have a std::vector<..> container.
- We cannot hoist size() function because compiler cannot prove tht the following function does not modify the relevant memory.

```
PODArray::reserveForNextSize()
```

- PODArray is an internally defined data structure, similar to std::vector.
- Proper definition of container that guarantees two containers have distinct memory could be enough to resolve this issue.
- Alternatively, if it is known that this function only reallocates memory, we were fine.

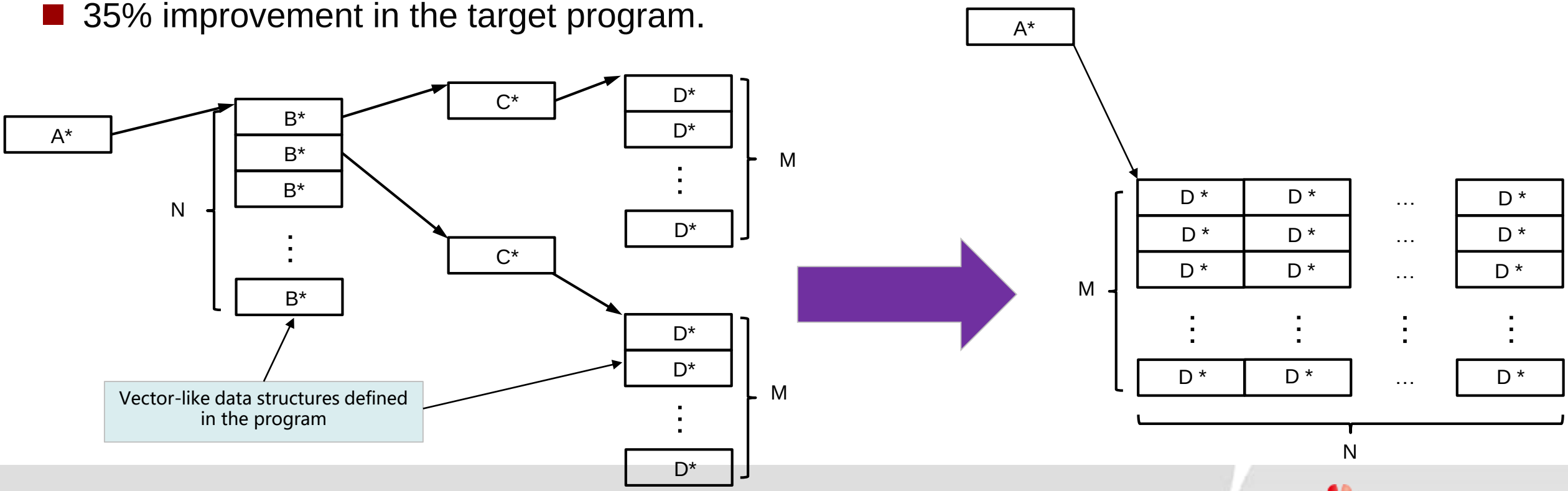
Example 2

- Temporarily create copies of pointers to D, directly reachable from A (While original objects still exist in the memory)
- The copies should be valid in a specific time interval in the program.
- We create a software cache.



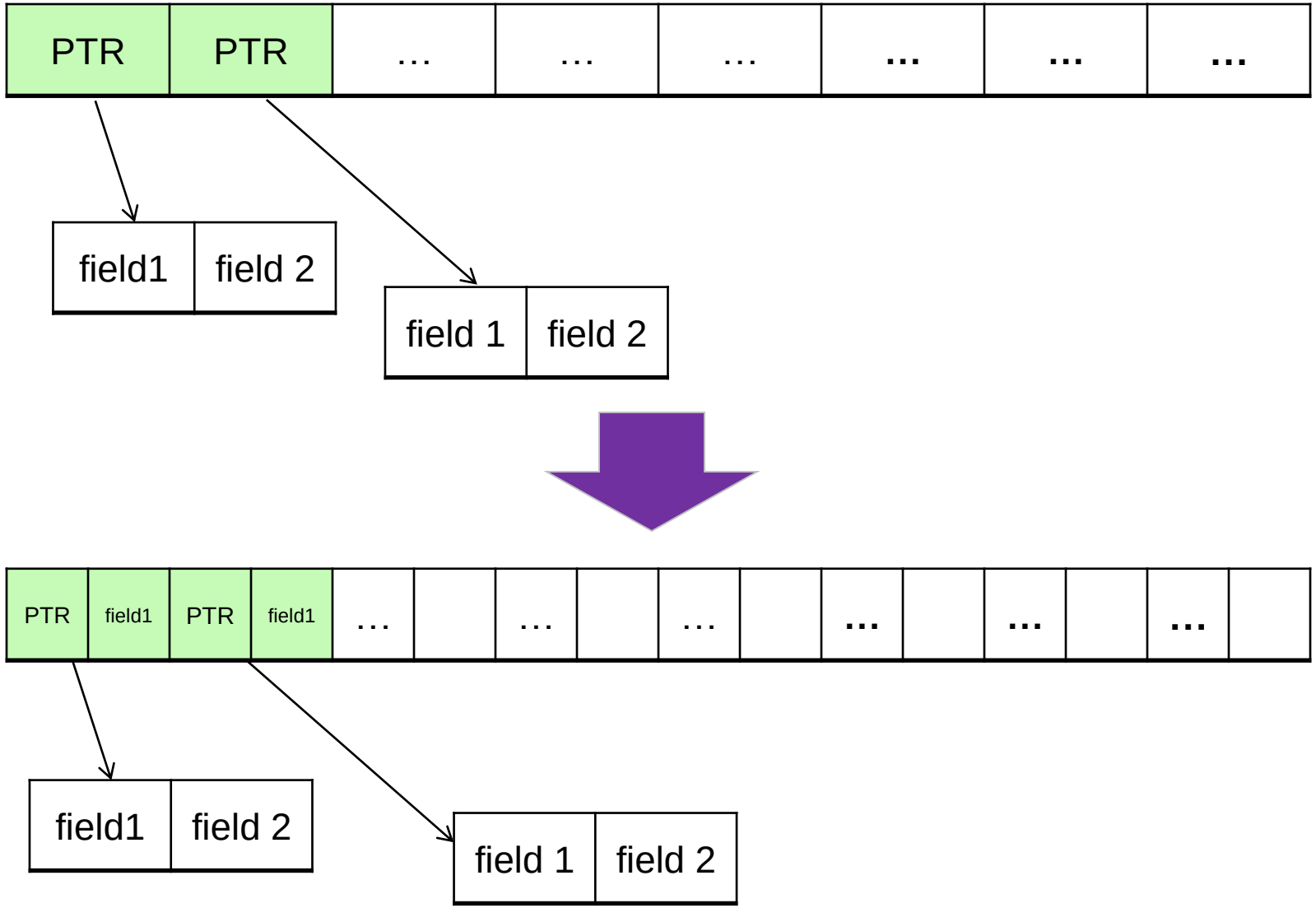
Example 2

- The challenge is to prove the cache remains valid, during the time interval of interest.
- We need to prove nothing is removed from the vector-like data structures.
- Concepts of “container”, and “insert”and “remove”to/from it are relevant.
- 35% improvement in the target program.



Example 3

- We have a sorted data structure that contains pointers to actual objects.
  - Sorting is based on “field 1”of the actual object.
  - Could be a sorted array, a priority queue, or a binary search tree.
- Hot code may include
  - Binary search over array.
  - Standard operations on BST or priority queue.
- Extra level of indirection and potential cache miss could be expensive.
- What if we keep a copy of “field 1” in the main data structure?
- Objects exists in the program, at some point they are inserted in this sorted data structure and then later on removed.
  - Challenge: How can we prove our copy of “field 1” is not invalidated?
  - In our case, even a very careful analysis of the program can not rule out that the same object with a modified “field 1” is inserted twice.
- 9% gain on an actual workload.
- The gain reduces if we put in guardrails for correctness.



A few questions

- Can we consider a couple of different optimizations in different programs a good enough motivation for a programming language change?
- What if programmer uses this attributes incorrectly?

Related work

- CGO 2024 paper [8]: “Representing data collections in an SSA form”
  - Proposes SSA representation of data collections in the compiler.
  - The authors need to modify C/C++ programs to use their library instead of the original data structures.
  - This allows the compiler to detect a data structure and translate it to proper IR.
  - Language support would be a better solution to achieve this goal.

- Knowledge of higher level semantics, helps compiler.
  - The performance gain reported in [8] depends on the compiler, understanding quick sort.

Related work: similar optimizations

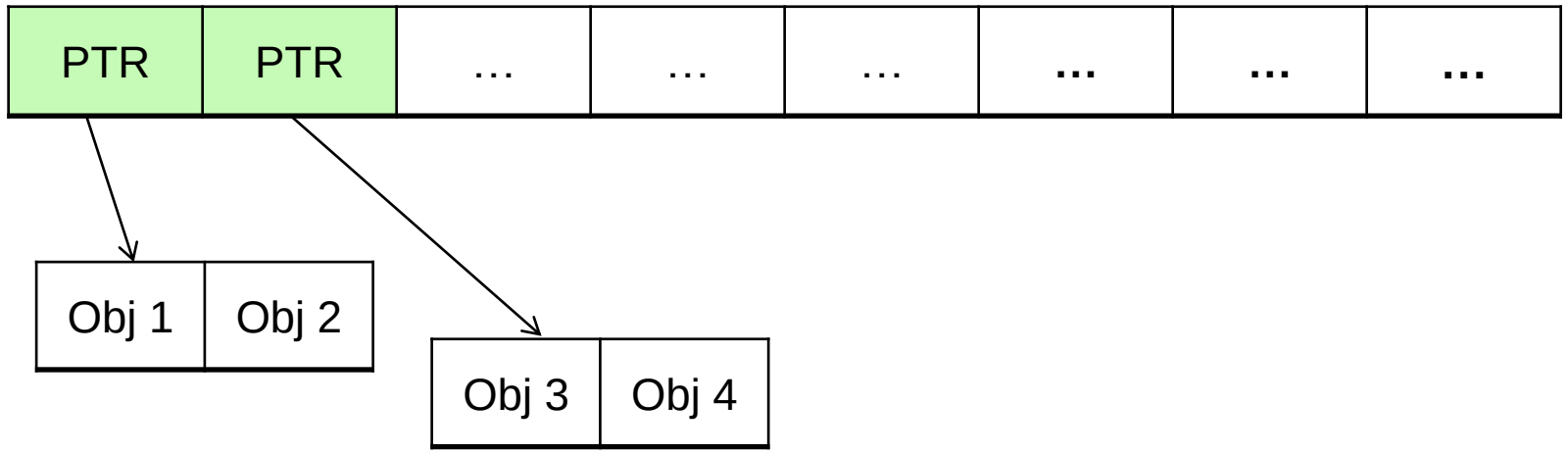
- Object inlining [6, 7]
- This is primarily an optimization for Java.

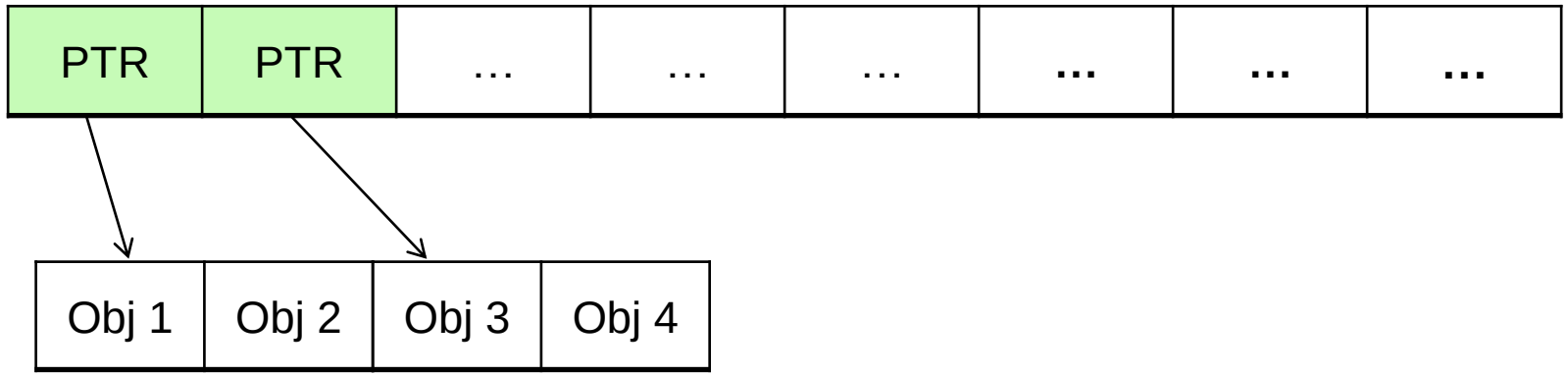
```
class Point {
  private int x;
  private int y;
}

class Line {
  private Point start;
  private Point end;
}
```

```
class LineInlined {
  private int start_x;
  private int start_y;
  private int end_x;
  private int end_y;
}
```

Related work: similar optimizations

- Array flattening [5]





References

1. N4713, Working draft, Standard for Programming Language C++
2. Evolution of Clang IR, LLVM Dev 2023
3. Common MLIR Dialect for C/C++ and Fortran, LLVM Dev 2020
4. A novel data layout optimization in BiSheng compiler, LLVM Dev 2023
5. A compiler framework for general memory layout optimizations targeting structures, INTERACT-14, 2010.
6. An automatic object inlining optimization and its evaluation, PLDI 2000
7. Compiler-assisted object inlining with value fields, PLDI 2021
8. Representing data collections in an SSA form, CGO 2024

Thank you

www.huawei.com