LLVM Data-structures overview

Marcello Maggioni

\(^1\)Codeplay Software Ltd.

EuroLLVM 2014
1 Motivation
   • Why having specific data-structures
   • LLVM Resources

2 Data-structures
   • Vectors
   • Maps
   • Sets
Outline

1 Motivation
   - Why having specific data-structures
   - LLVM Resources

2 Data-structures
   - Vectors
   - Maps
   - Sets
Why not using standard structures?

- C++ Standard data-structures have performance that is platform dependent
- C++ Standard might not have a specific kind of data structures (like HashMaps. With C++11 this problem was solved)
- Specialized data-structures can be made faster than the Standard generic ones
Outline

1 Motivation
   - Why having specific data-structures
   - LLVM Resources

2 Data-structures
   - Vectors
   - Maps
   - Sets
Resources on LLVM Data-Structures

- Look for methods in every subclass
- The most exposed interface does not expose all methods in documentation usually
Outline

1 Motivation
   • Why having specific data-structures
   • LLVM Resources

2 Data-structures
   • Vectors
   • Maps
   • Sets
Possible Choices

- LLVM SmallVector
- std::vector
SmallVector

- Vector-like data structure
- Is optimized to contain a fixed amount of elements
- It is flexible if more elements are added
- Interface similar to std::vector
```cpp
#include "llvm/ADT/SmallVector.h"

SmallVector<type, N> V;

void foo(SmallVectorImpl<type> &V) {
}
```
SmallVector<Instruction*, 10> WorkList;

for (...) {
    Instruction *I = ...;
    WorkList.push_back(I);
}

...  
while (WorkList.empty()) {
    Instruction *I = WorkList.pop_back_val();
    ...
}


SmallVector vs World

Vectors insert speed

- std::vector<int>
- SmallVector<int, 1000>

- 500 inserts
- 1000 inserts
- 1500 inserts
SmallVector vs World

Vectors insert speed

<table>
<thead>
<tr>
<th>Vector type</th>
<th>Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::vector&lt;int&gt; with reserve(1000)</td>
<td>28</td>
</tr>
<tr>
<td>SmallVector&lt;int*, 1000&gt;</td>
<td>14</td>
</tr>
</tbody>
</table>
Outline

1 Motivation
   • Why having specific data-structures
   • LLVM Resources

2 Data-structures
   • Vectors
   • Maps
   • Sets
Possible Choices

- LLVM DenseMap
- LLVM StringMap (only for strings)
- std::map
- std::unordered_map
DenseMap is a quadratically probed HashMap.

- Keeps everything in a single memory allocation
- Iterators potentially invalidated after insertion
- Matches pretty closely std::map interface
  - insert(std::pair<>)
  - find(Key&)
  - count(Key&)
  - begin(), end() (unordered)
DenseMap Keys

- Supports all pointers and integer types as keys
- Additional Key types can be specified defining a custom DenseMapInfo<> class

```c
struct KeyInfo {
    static inline T getEmptyKey() {...}
    static inline T getTombstoneKey() {...}
    static unsigned getHashValue(const T &Val) {...}
    static bool isEqual(const T &LHS, const T &RHS)
        { ... }
};

DenseMap<Key, Value, KeyInfo> M;
```
DenseMap vs World

Maps speed insert

- DenseMap\textless int*, Data\textgreater
- std::map\textless int*, Data\textgreater
- std::unordered_map\textless int*, Data\textgreater

Map type

- 500
- 1500
Data-structures
Maps

DenseMap vs World

Maps speed lookup

Map type

<table>
<thead>
<tr>
<th>Map type</th>
<th>500</th>
<th>1500</th>
</tr>
</thead>
<tbody>
<tr>
<td>DenseMap&lt;int*, Data&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>std::map&lt;int*, Data&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>std::unordered_map&lt;int*, Data&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
StringMap

- Specific implementation of an HashMap only for having strings as keys
- Strings are copied into the map. They don’t store the pointer to the map as a key.
- Similar interface to DenseMap
- Insert is different though ... it is actually called GetOrCreateValue()
const char *str = "__some_symbol";

StringMap<Data> Map;
Data D = { 10, 5 };

Map.GetOrCreateValue(str, D);
Map[str] = D;
Map.find(str);
Map.count(str);
Storing a 16 character wide random string

- **StringMap vs World**

![String maps speed chart](chart.png)

- **Map type**
  - StringMap<Data>
  - std::map<std::string, Data>
  - std::unordered_map<std::string, Data>

- **US**
  - 50 inserts
  - 50 lookups
StringMap vs World

String maps speed

<table>
<thead>
<tr>
<th>Map type</th>
<th>1500 inserts</th>
<th>1500 lookups</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringMap&lt;Data&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>std::map&lt;std::string, Data&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>std::unordered_map&lt;std::string, Data&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

1. Motivation
   - Why having specific data-structures
   - LLVM Resources

2. Data-structures
   - Vectors
   - Maps
   - Sets
Possible Choices

- Sorted vectors
- LLVM SmallSet
- std::set
- std::unordered_set
SmallSet

- Replacement for set in LLVM
- It is implemented as a small vector of fixed size that is not sorted
- Searches are linear in time
- When exceeding the specified size switches to a quadratically probed set for some keys and std::set for others
- Cannot be iterated, only for querying
SmallSet

SmallSet<int *, 10> S
int a;
S.insert(&a);
S.count(&a);
SmallSet vs World

Sets speed

Set type

- SmallVector<int> SA
- SmallVector<int> SD
- SmallSet<int*, 1500>
- SmallSet<int*, 10>
- set<int*>
- unordered_set<int*>

US

0 50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850

1500 inserts
1500 lookups

Marcello Maggioni (Codeplay Software Ltd.)
Summary

- Covered basic data structures and their performance comparisons
- Other data structures are available for specific needs (BitVectors, SparseSet, ValueMap ...)
- Using LLVM data-structures can give performance portability
- LVM data-structures are not always faster and may require parameter tuning
Marcello Maggioni

- marcello.maggioni@gmail.com
- marcello@codeplay.com