Fine-grained Compilation Caching using llvm-cas

Shubham Rastogi

LLVM Dev Meeting 2024

Background

- This talk is about compilation caching, using a Content Addressable Storage (CAS)

Background

- This talk is about compilation caching, using a Content Addressable Storage (CAS)

Past LLVM Dev Meeting Talks

LLVM Dev 2023: Representing Debug Info in LLVM CAS
https://www.youtube.com/watch?v=VPqZ8LoM5Z8

LLVM Dev 2022: Using Content-Addressable Storage in Clang for Caching Computations and Eliminating Redundancy
https://www.youtube.com/watch?v=E9GdNKjGZ7Y

Agenda

- Content-Addressable Storage (CAS) recap
- Improvements to `.debug_info` section representation in fine-grained object storage
- Improvements to replay speed in fine-grained object storage
- Fine-grained object storage support for Swift

Introducing MCCAS!

- One thing we are doing with a CAS is to create a build cache, comparable to ccache
- Split object files CASObjects for finer-grained object storage

ccache vs MCCAS

ccache

- Granularity: Object File level
- Higher rate of growth over incremental builds

MCCAS

- Granularity: Below Function Level
- Lower rate of growth over incremental builds

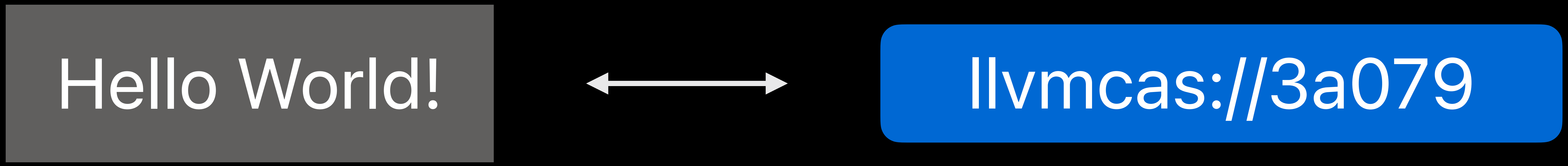
CAS Object Store refresher

CAS object address = hash of contents

CAS Object Store

CAS object address = hash of contents

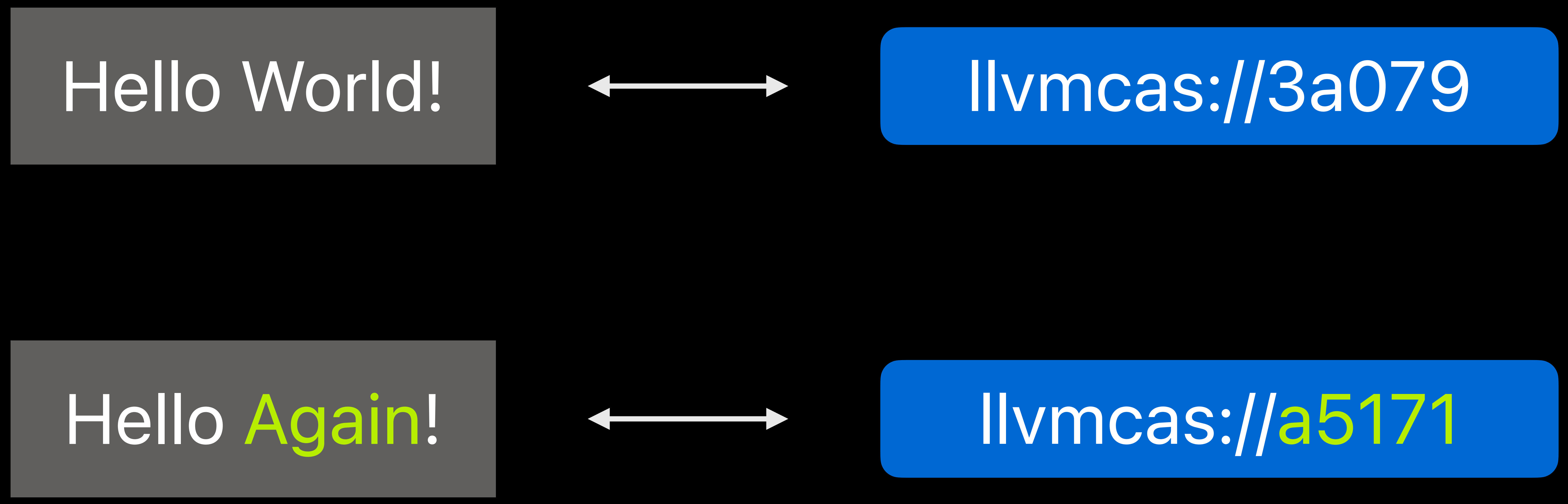
1:1 mapping



CAS Object Store

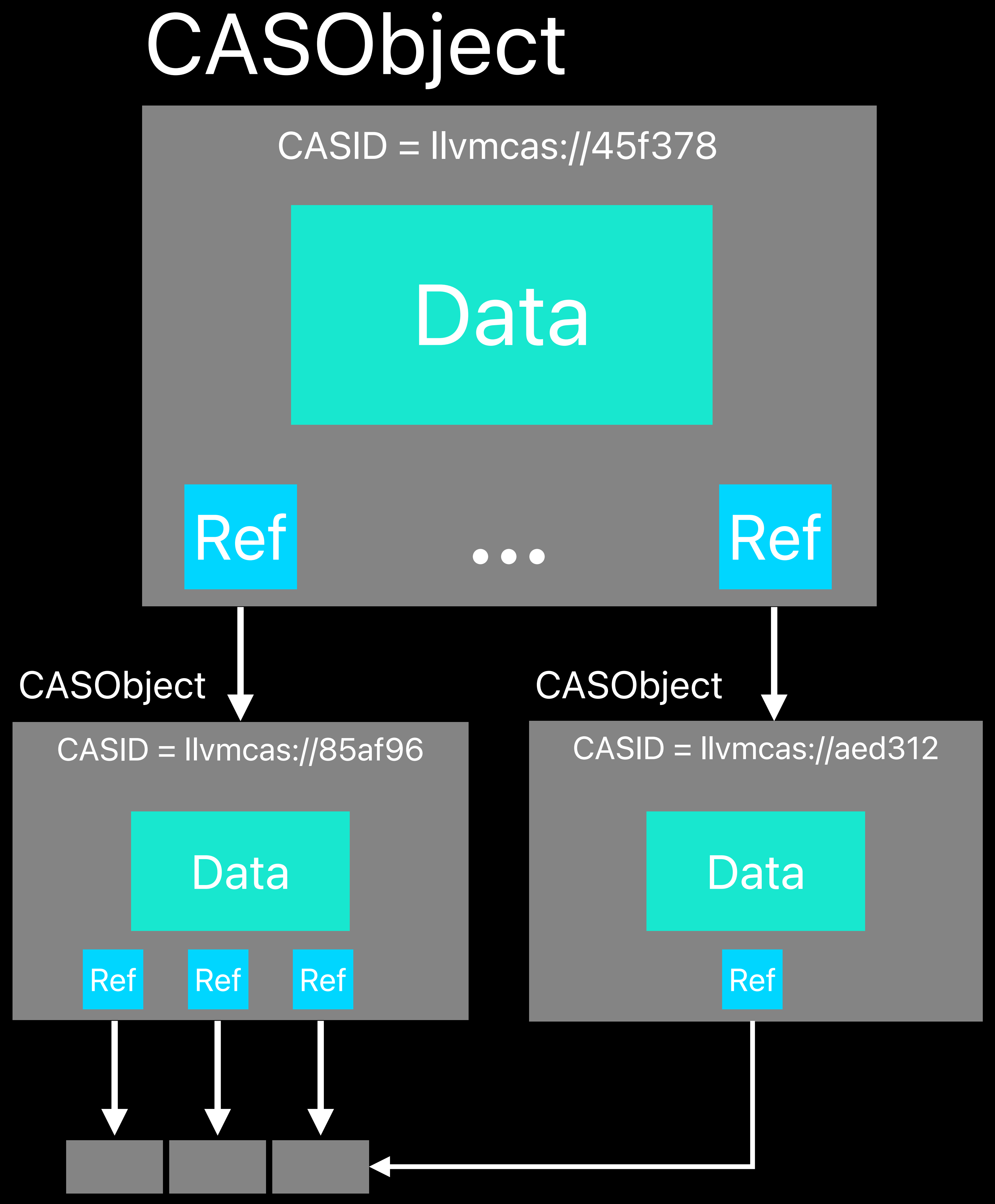
CAS object address = hash of contents

1:1 mapping



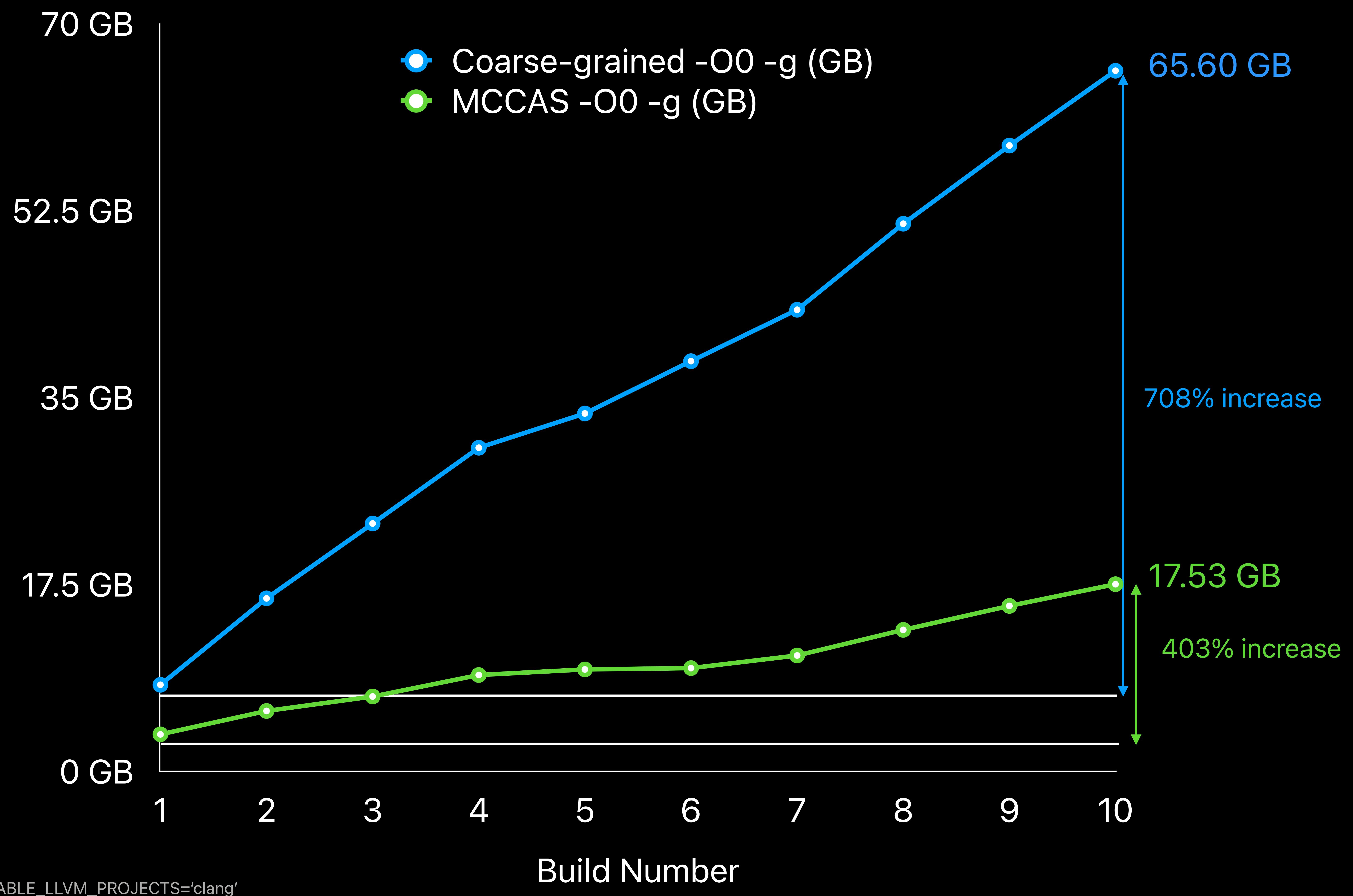
Representation of Content in the CAS

- Content is a DAG of CASObjects
- Each CASObject has data and a list of references to other CASObjects



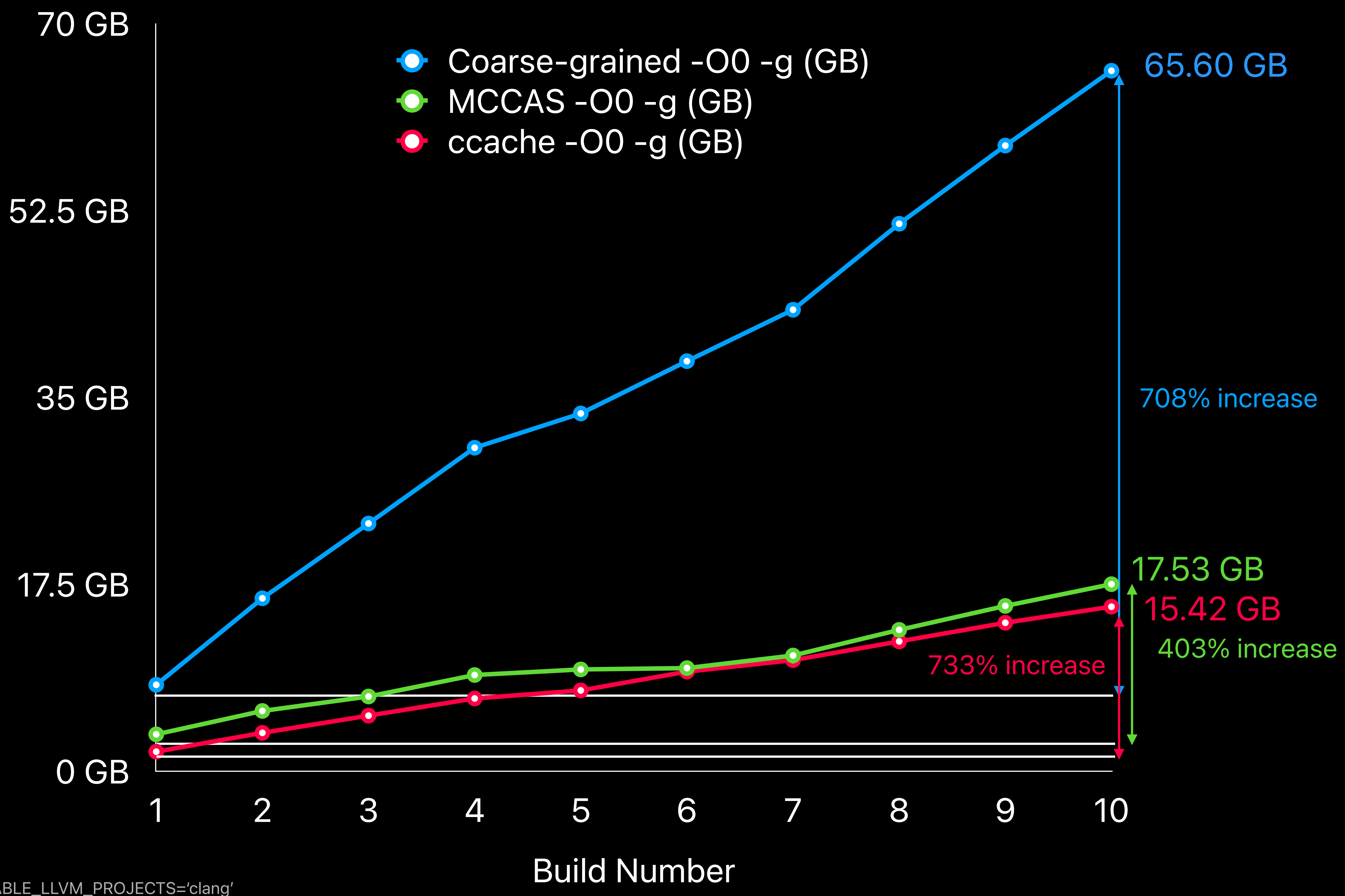
Where we left off

Where we left off



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

Where we left off

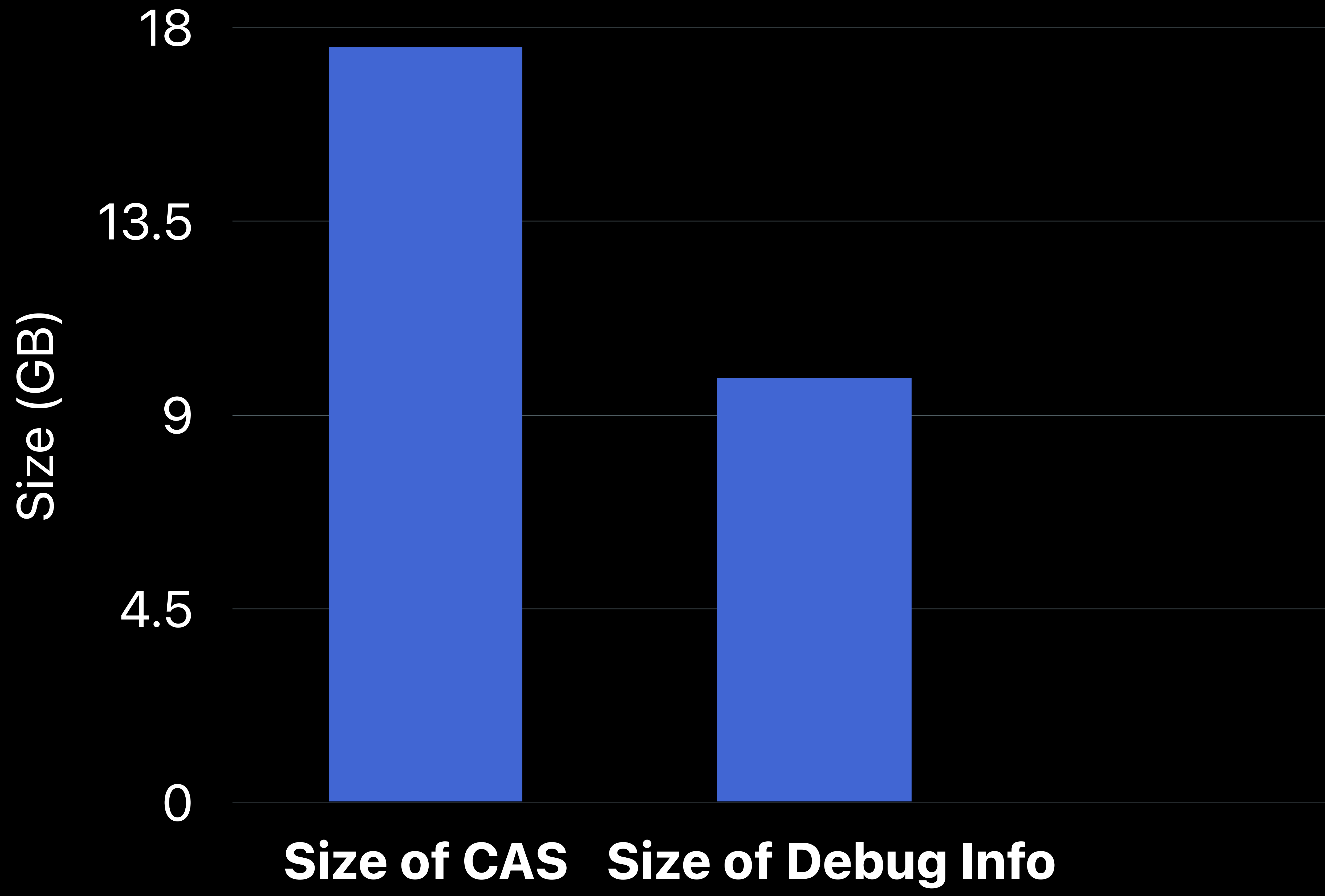


llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

Improvements since last year

Improvements since last year

- `.debug_info` section
  >50% of the total CAS size



.debug_info representation

```
int func(int x) {
  return x+1;
}

int func2(int x) {
  return x+1;
}
```

.debug_info representation

```
dwarfdump a.o —debug-info -f func

0x25: DW_TAG_subprogram
        DW_AT_low_pc(0x…)
        DW_AT_high_pc(0x…)
        DW_AT_linkage_name("_Z4funci")
        DW_AT_name("func")
        DW_AT_decl_file("a.cpp")
        DW_AT_decl_line(1)
        DW_AT_type(“int”)
```

```
dwarfdump a.o —debug-info -f func2

0x41: DW_TAG_subprogram
        DW_AT_low_pc(0x…)
        DW_AT_high_pc(0x…)
        DW_AT_linkage_name("_Z5func2i")
        DW_AT_name("func2")
        DW_AT_decl_file("a.cpp")
        DW_AT_decl_line(5)
        DW_AT_type(“int”)
```

.debug_info representation

```
dwarfdump a.o —debug-info -f func

0x25: DW_TAG_subprogram
        DW_AT_low_pc(0x…)
        DW_AT_high_pc(0x…)
        DW_AT_linkage_name("_Z4funci")
        DW_AT_name("func")
        DW_AT_decl_file("a.cpp")
        DW_AT_decl_line(1)
        DW_AT_type(“int”)
```

```
dwarfdump a.o —debug-info -f func2

0x41: DW_TAG_subprogram
        DW_AT_low_pc(0x…)
        DW_AT_high_pc(0x…)
        DW_AT_linkage_name("_Z5func2i")
        DW_AT_name("func2")
        DW_AT_decl_file("a.cpp")
        DW_AT_decl_line(5)
        DW_AT_type(“int”)
```

Debug information is represented by Debug Information Entries or DIEs

.debug_info representation

```
dwarfdump a.o —debug-info -f func

0x25: DW_TAG_subprogram
        DW_AT_low_pc(0x…)
        DW_AT_high_pc(0x…)
        DW_AT_linkage_name("_Z4funci")
        DW_AT_name("func")
        DW_AT_decl_file("a.cpp")
        DW_AT_decl_line(1)
        DW_AT_type(“int”)
```

```
dwarfdump a.o —debug-info -f func2

0x41: DW_TAG_subprogram
        DW_AT_low_pc(0x…)
        DW_AT_high_pc(0x…)
        DW_AT_linkage_name("_Z5func2i")
        DW_AT_name("func2")
        DW_AT_decl_file("a.cpp")
        DW_AT_decl_line(5)
        DW_AT_type(“int”)
```

Some data in a DIE does not deduplicate, this goes into a separate CAS block called DistinctData

.debug_info representation

```
dwarfdump a.o —debug-info —f func —c

0x25: DW_TAG_subprogram
        DW_AT_low_pc(0x…)
        …
0x35:   DW_TAG_formal_parameter
          DW_AT_location(…)
          DW_AT_name(“x”)
          DW_AT_decl_file(“a.cpp”)
          DW_AT_decl_line(1)
          DW_AT_type(“int”)
```

```
dwarfdump a.o —debug-info —f func2 —c

0x41: DW_TAG_subprogram
        DW_AT_low_pc(0x…)
        …
0x51:   DW_TAG_formal_parameter
          DW_AT_location(…)
          DW_AT_name(“x”)
          DW_AT_decl_file(“a.cpp”)
          DW_AT_decl_line(1)
          DW_AT_type(“int”)
```

DIEs can have children DIEs

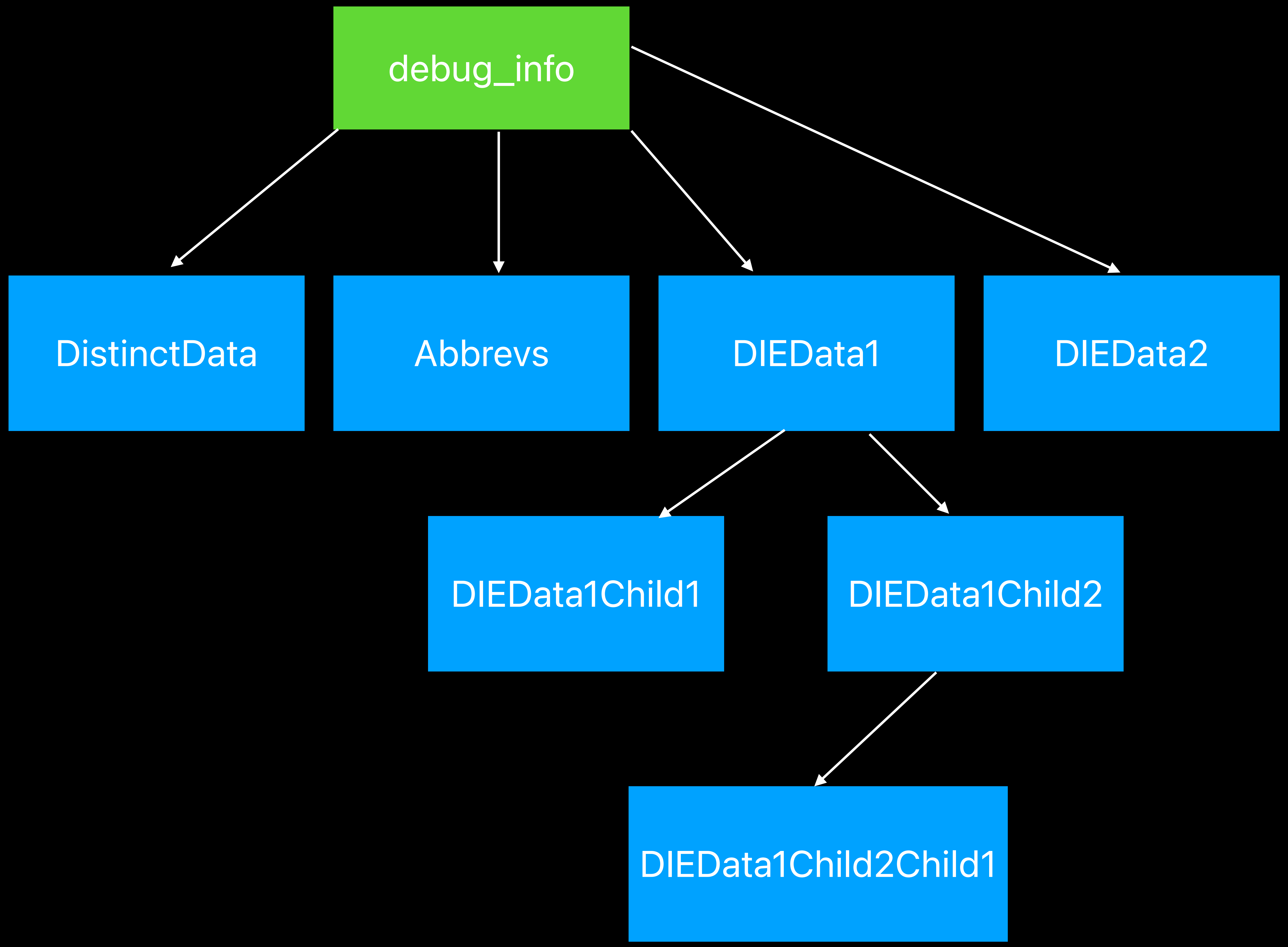
.debug_abbrev

.debug_abbrev can be thought of as the "type" of a DIE

```
dwarfdump a.o —debug-abbrev

[2] DW_TAG_subprogram
DW_CHILDREN_yes
       DW_AT_low_pc          DW_FORM_addrx
       DW_AT_high_pc         DW_FORM_data4
       DW_AT_linkage_name    DW_FORM_strx1
       DW_AT_name            DW_FORM_strx1
       DW_AT_decl_file       DW_FORM_data1
       DW_AT_decl_line       DW_FORM_data1
       DW_AT_type            DW_FORM_ref4
```

Debug Information representation during LLVM Dev Meeting 2023



`.debug_info` representation improvements

Two main improvements in `.debug_info` representation

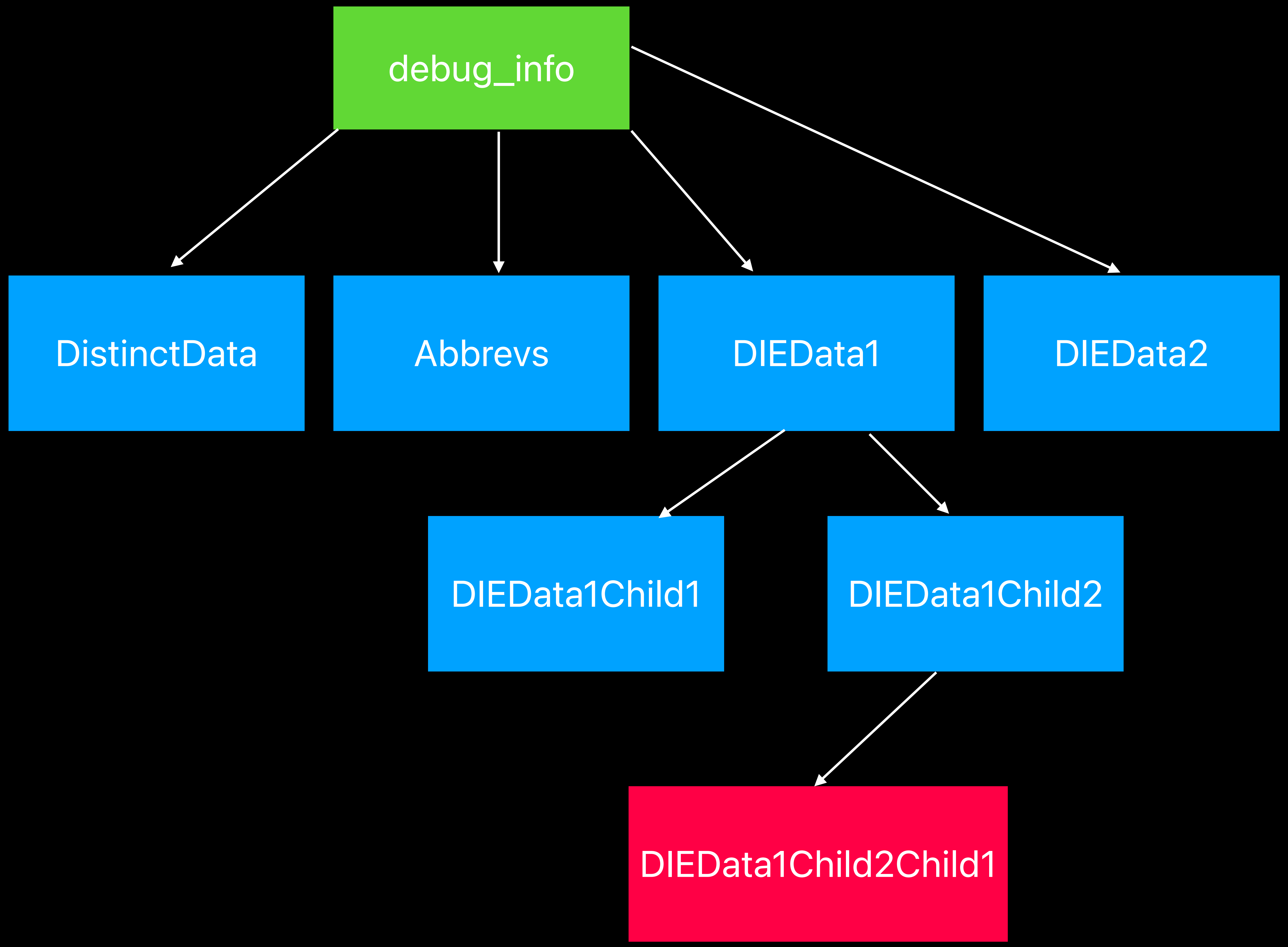
- Flattening of the `.debug_info` section CAS layout
- Reduction of the size of the *DistinctData* CAS Object, via compression

Debug Information representation during LLVM Dev Meeting 2023

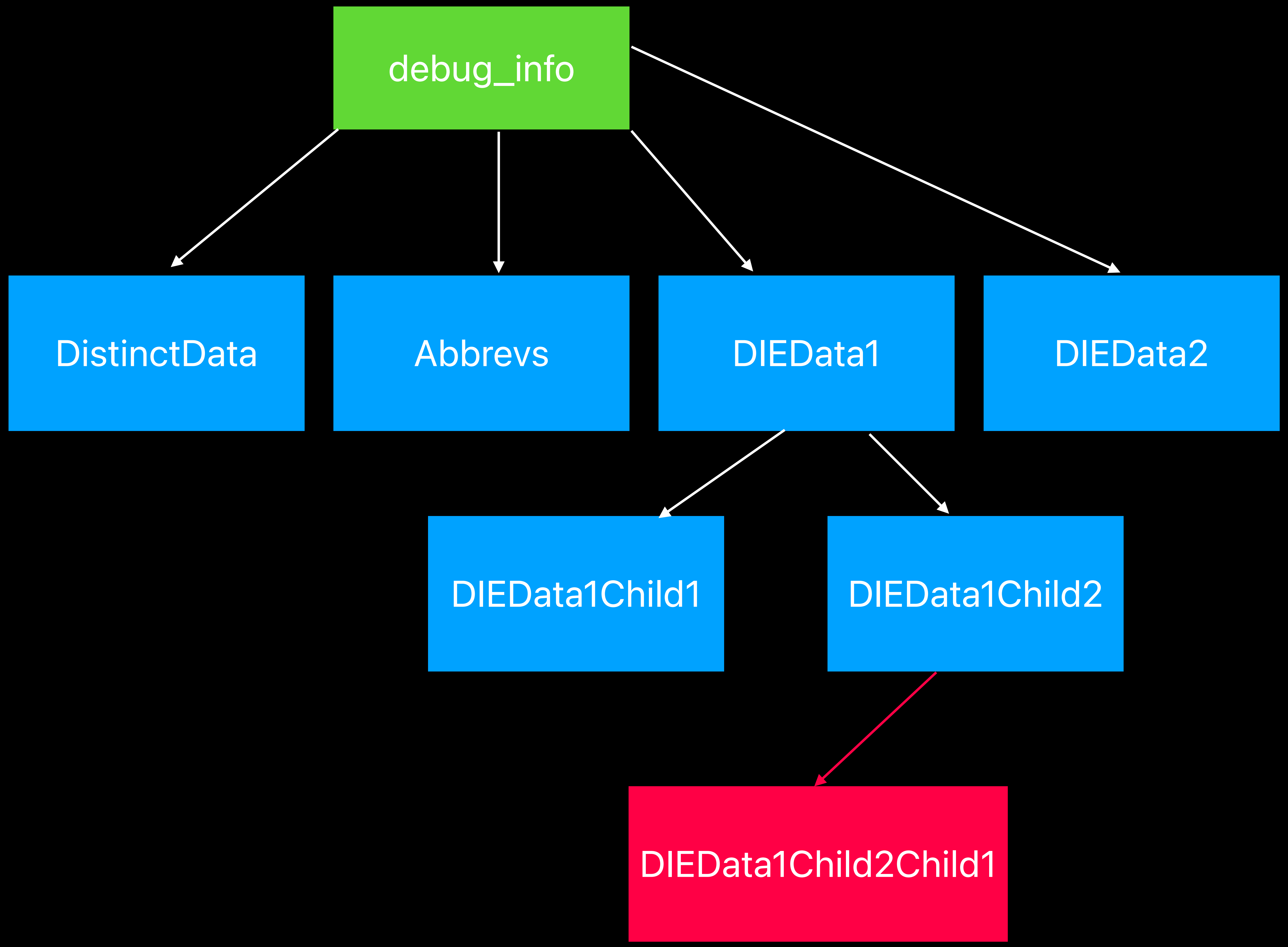
Flattening of the Debug Information section representation

- CAS Object’s Address = Hash of its contents
- CAS Object’s contents is the data, ***and*** the list of references to other CAS Objects
- Also, CAS Blocks are always ordered

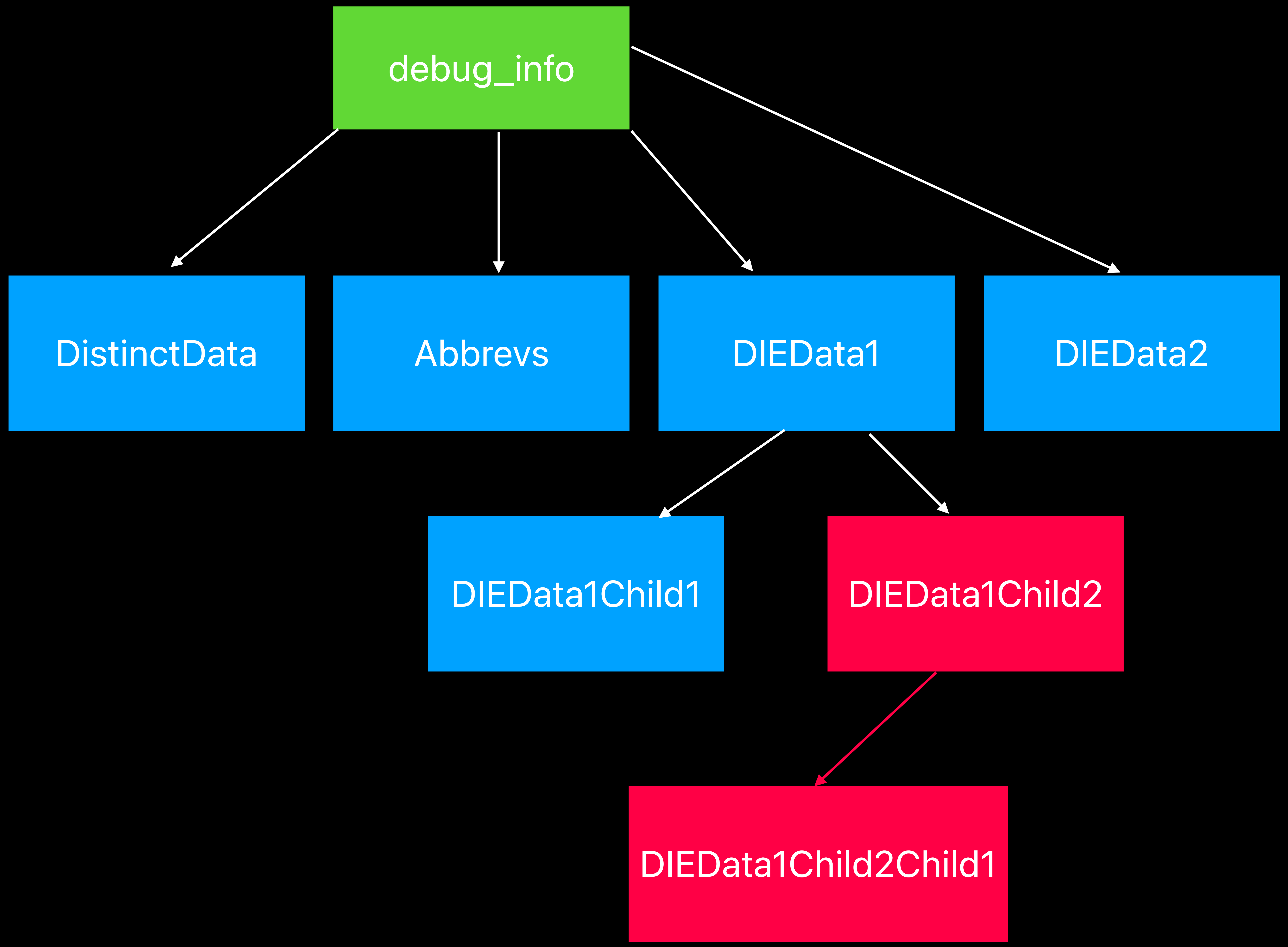
Debug Information representation during LLVM Dev Meeting 2023



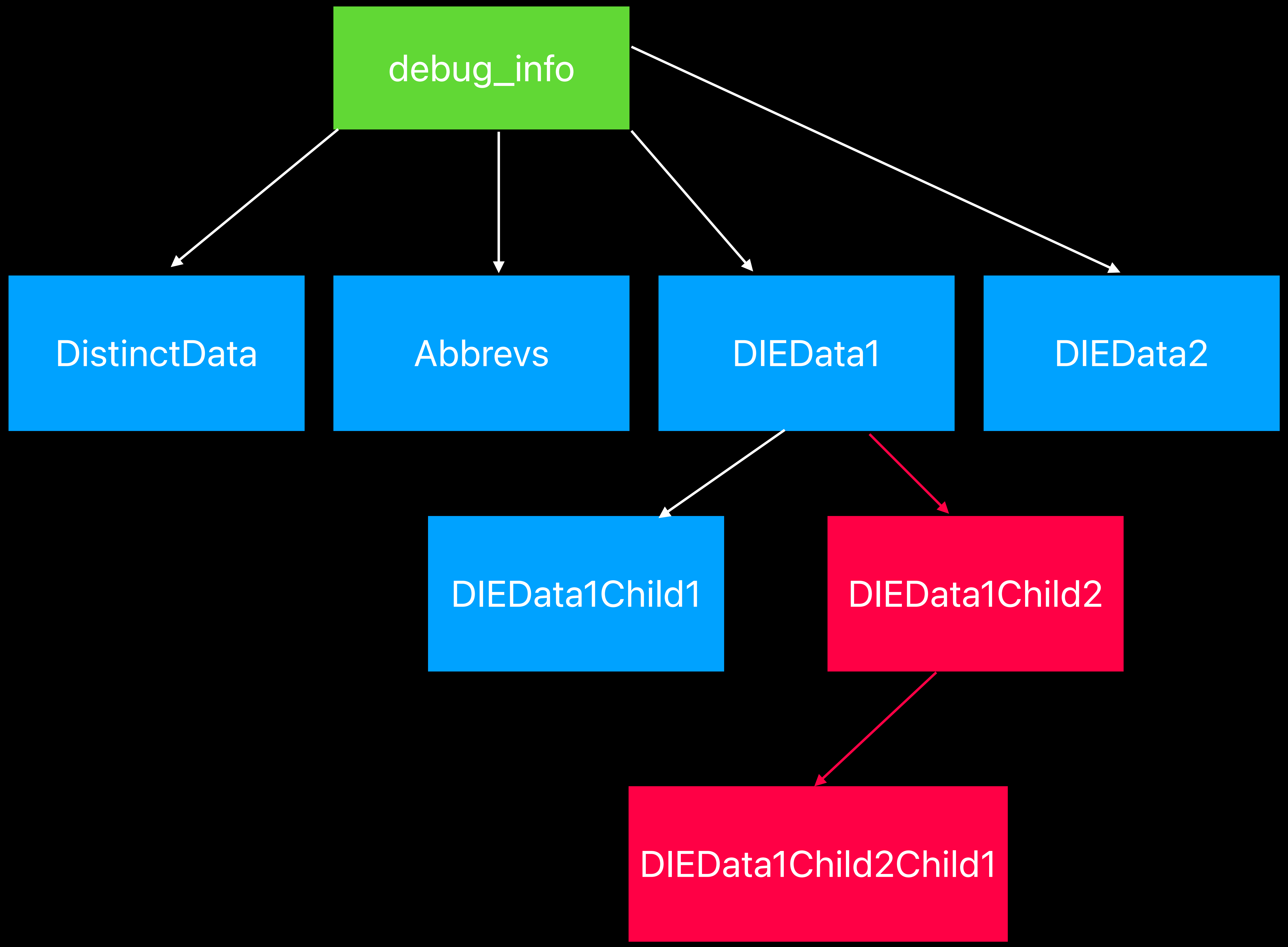
Debug Information representation during LLVM Dev Meeting 2023



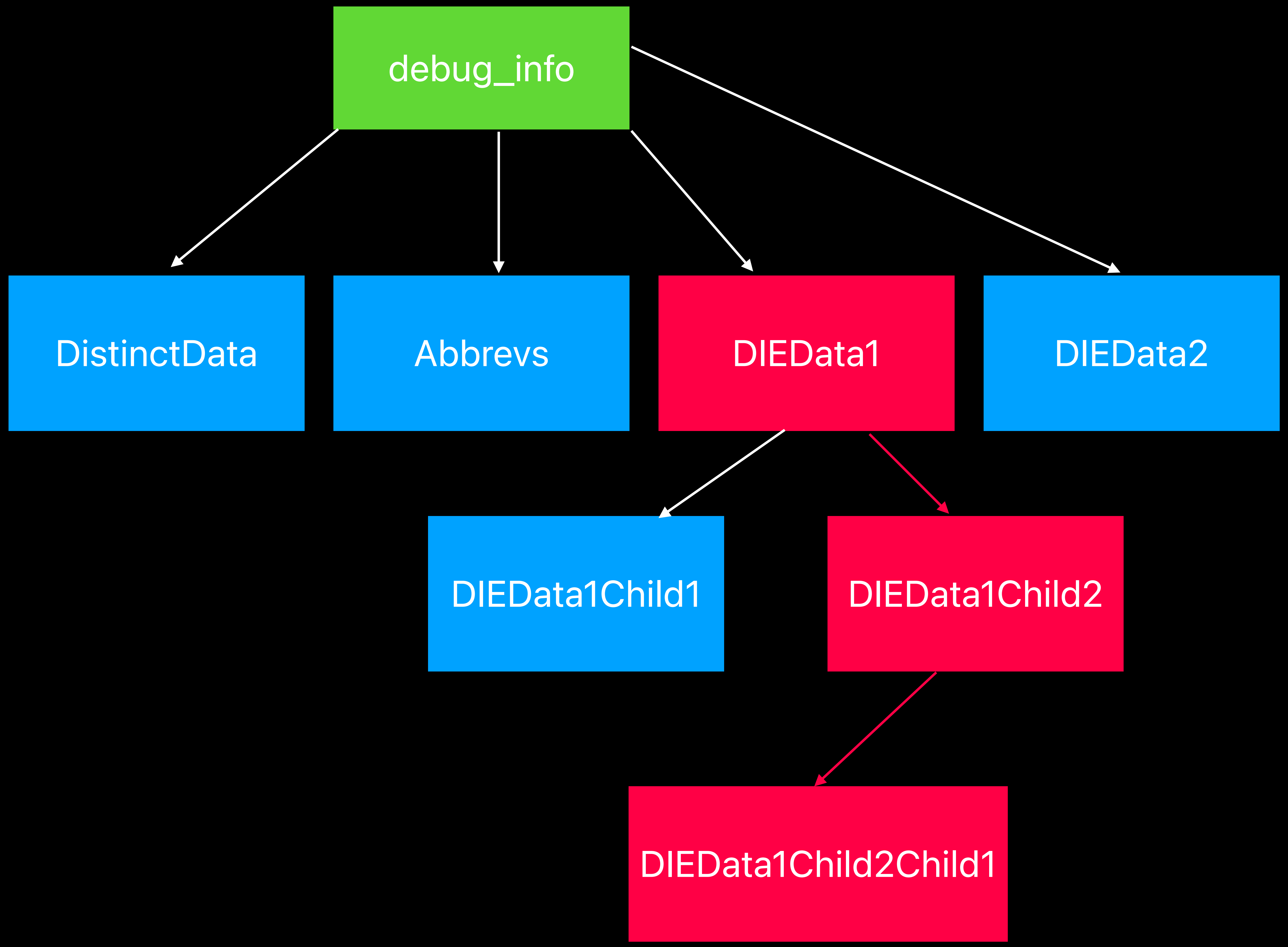
Debug Information representation during LLVM Dev Meeting 2023



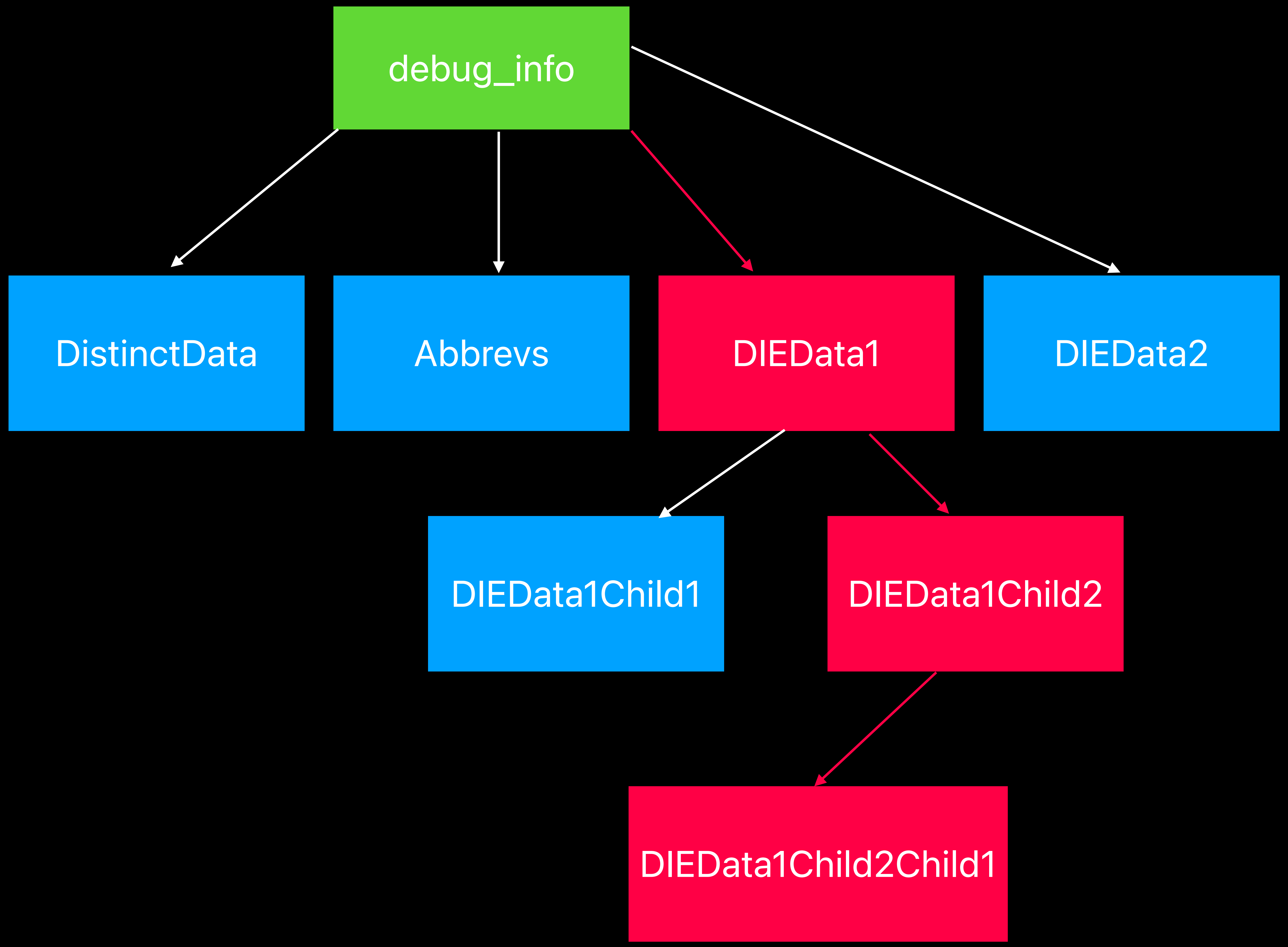
Debug Information representation during LLVM Dev Meeting 2023



Debug Information representation during LLVM Dev Meeting 2023

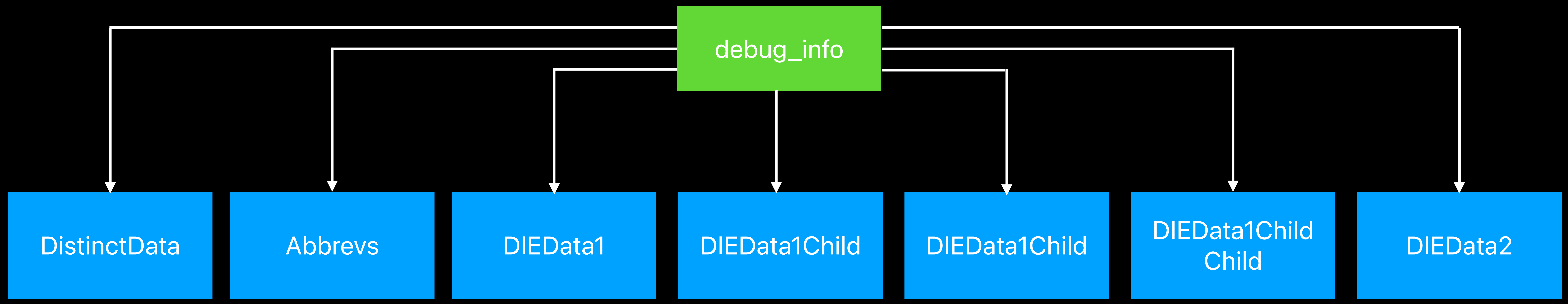


Debug Information representation during LLVM Dev Meeting 2023



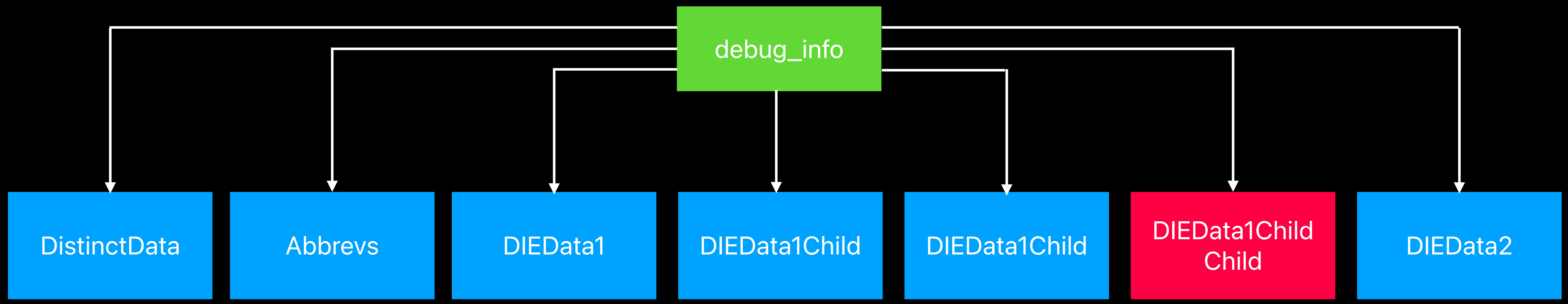
New representation

Flattening of the Debug Information section representation



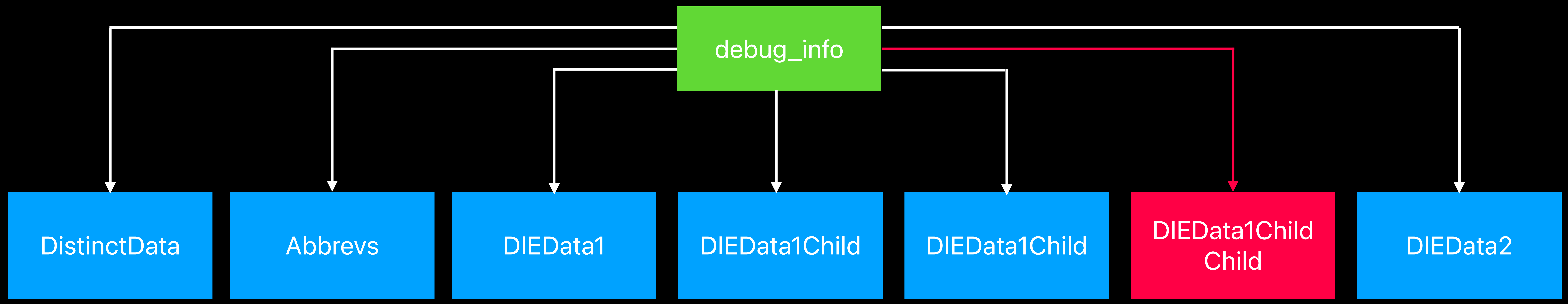
New representation

Flattening of the Debug Information section representation



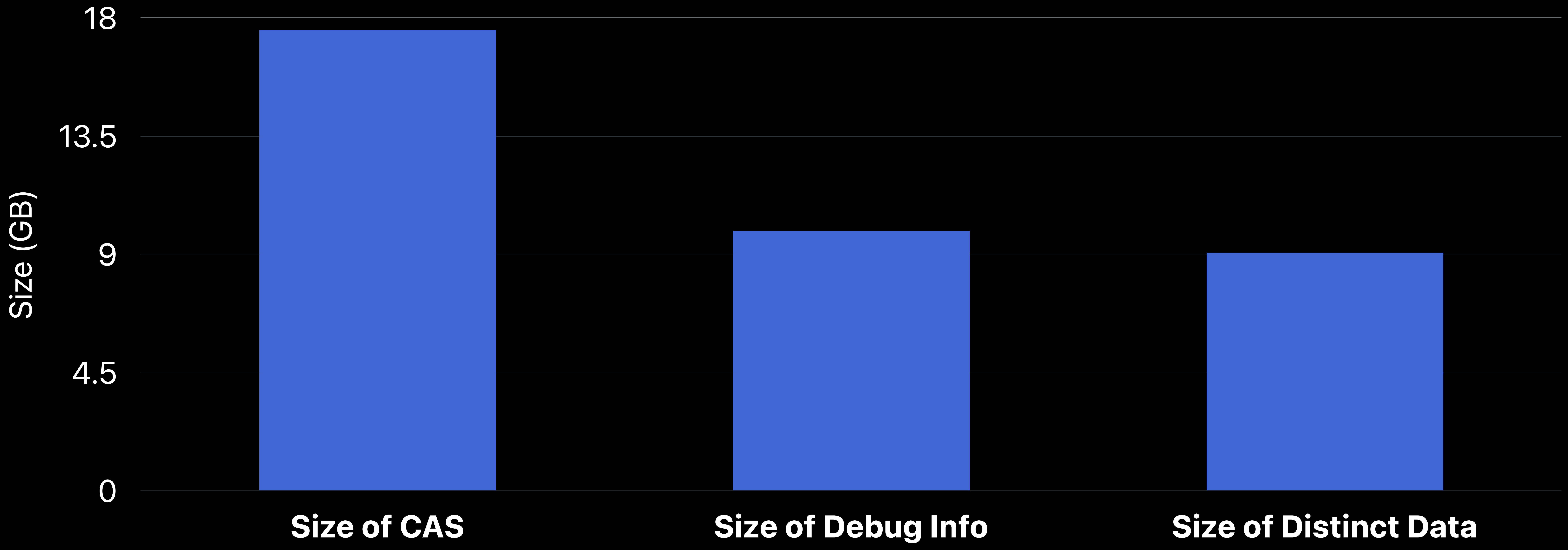
New representation

Flattening of the Debug Information section representation



Adding Compression

Reduction of the size of the DistinctData CAS Object block via compression



Debug Information representation improvements

Reduction of the size of the DistinctData CAS Object block via compression

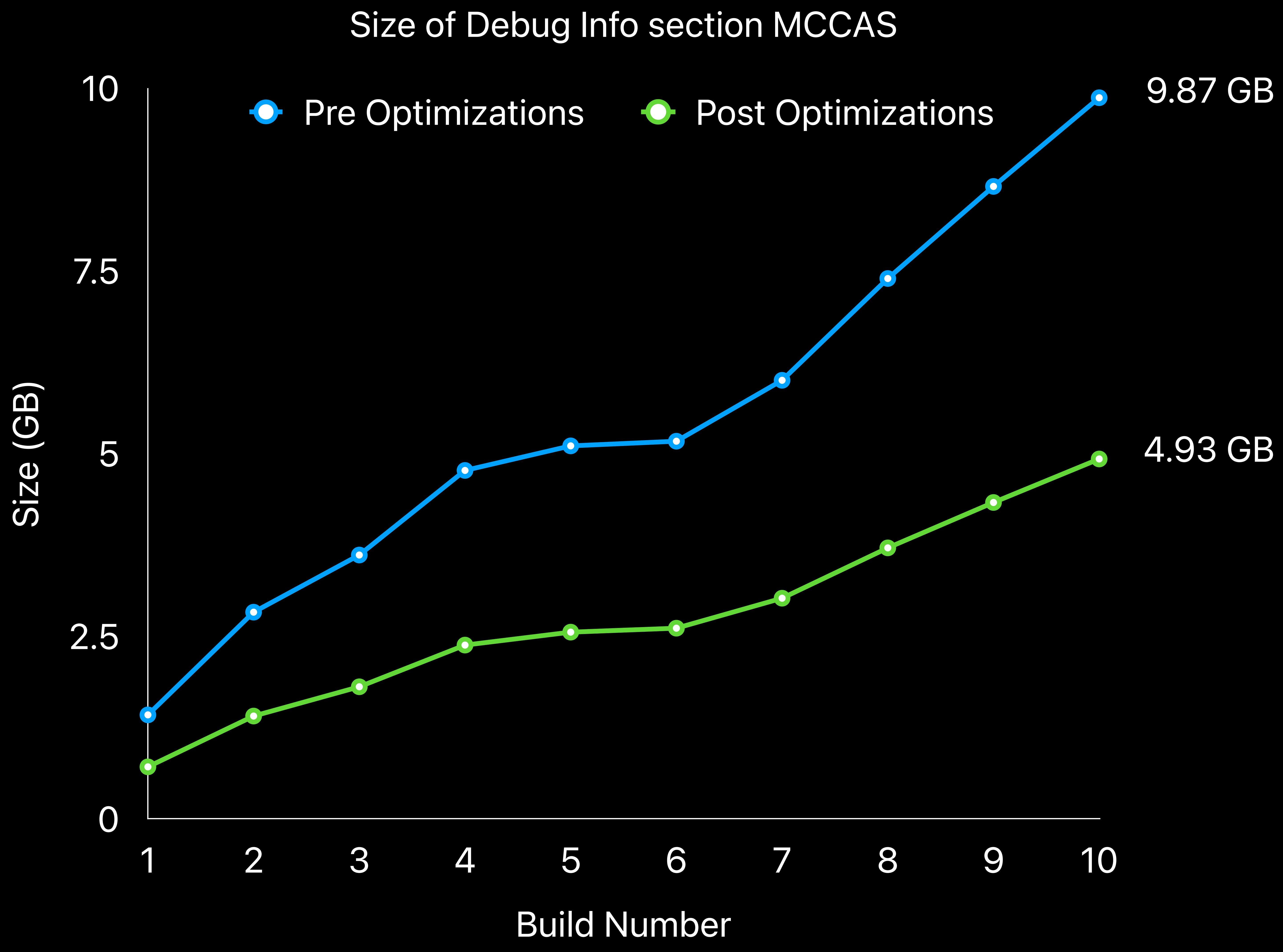
- *DistinctData* block stores all the data that doesn't deduplicate
- Accounts for 90% of `.debug_info` in CAS

Debug Information representation improvements

Reduction of the size of the DistinctData CAS Object block via compression

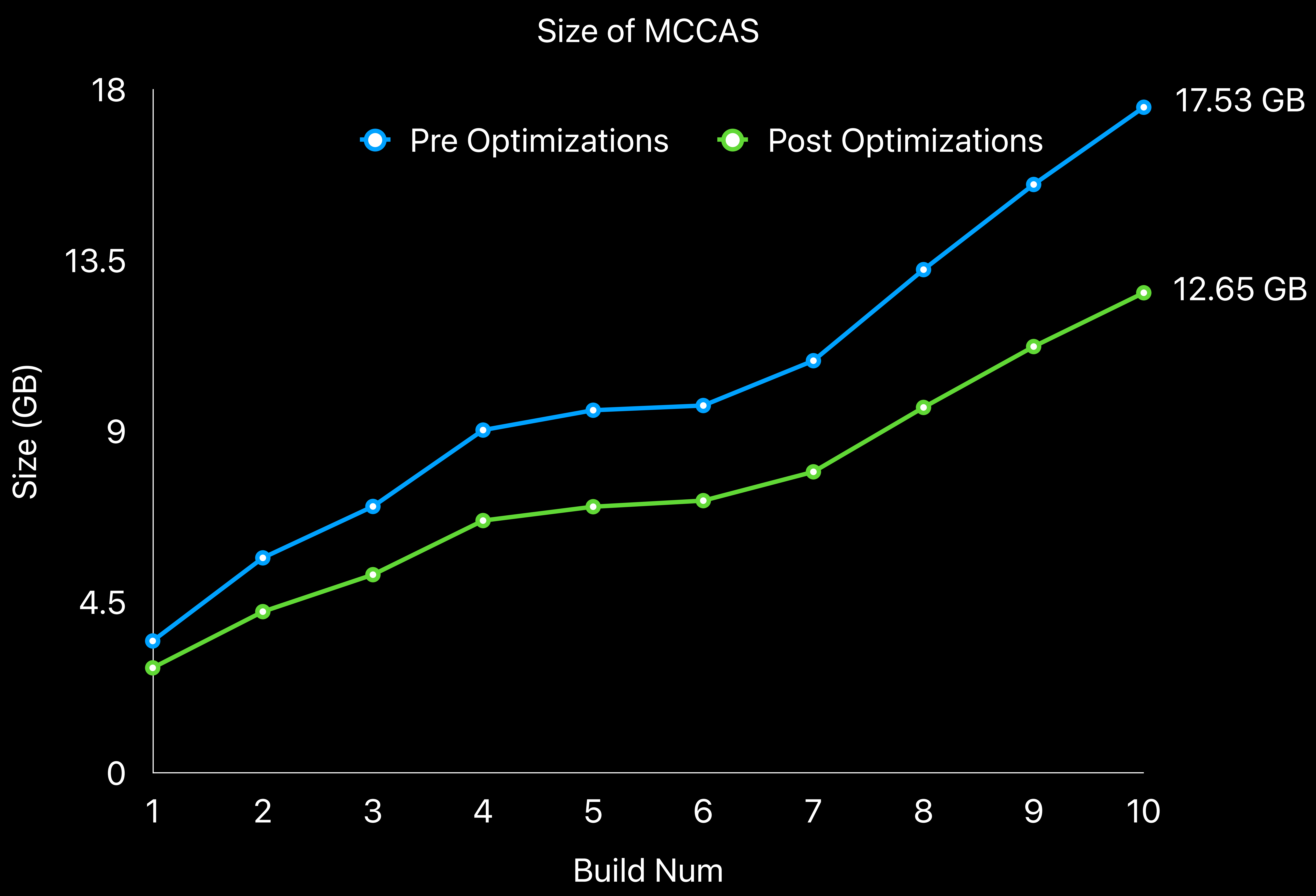
- *DistinctData* block stores all the data that doesn’t deduplicate
- Accounts for 90% of `.debug_info` in CAS
- Only one *DistinctData* block per object file
  - 9.07 GB = 14370 CAS Blocks, or 630 KB per Block

Debug Information representation improvements: Results



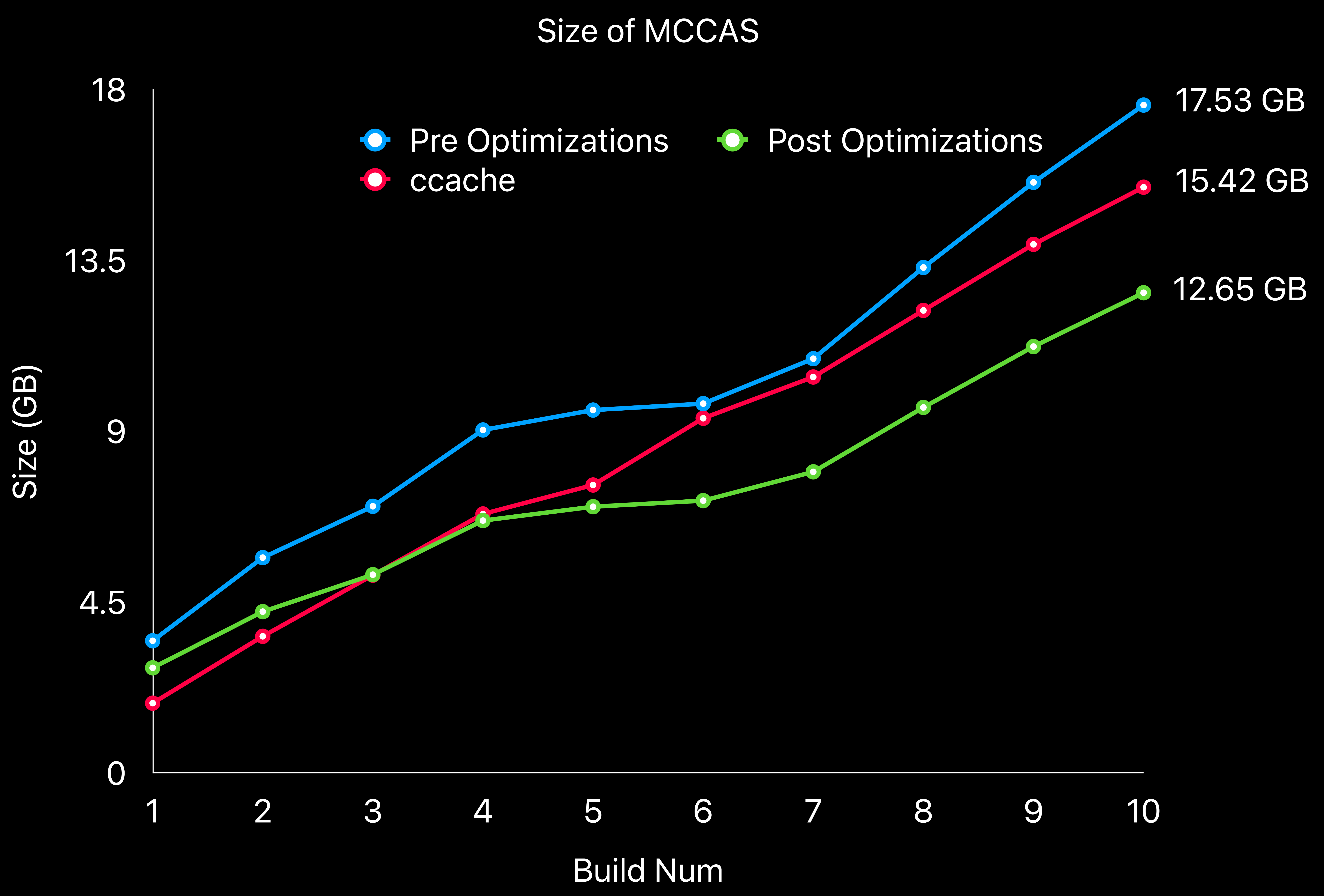
llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

Debug Information representation improvements: Results



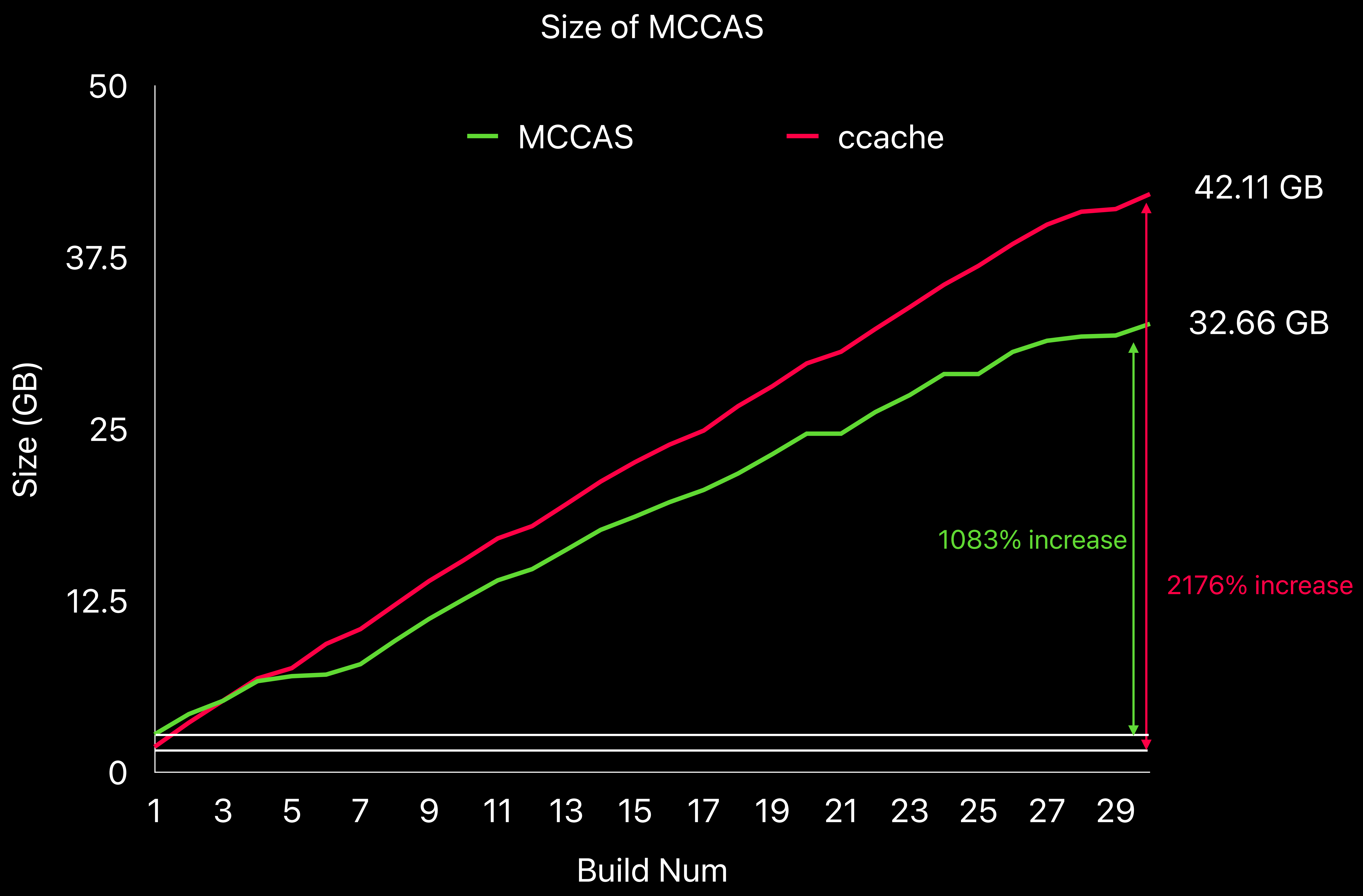
llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

Debug Information representation improvements: Results



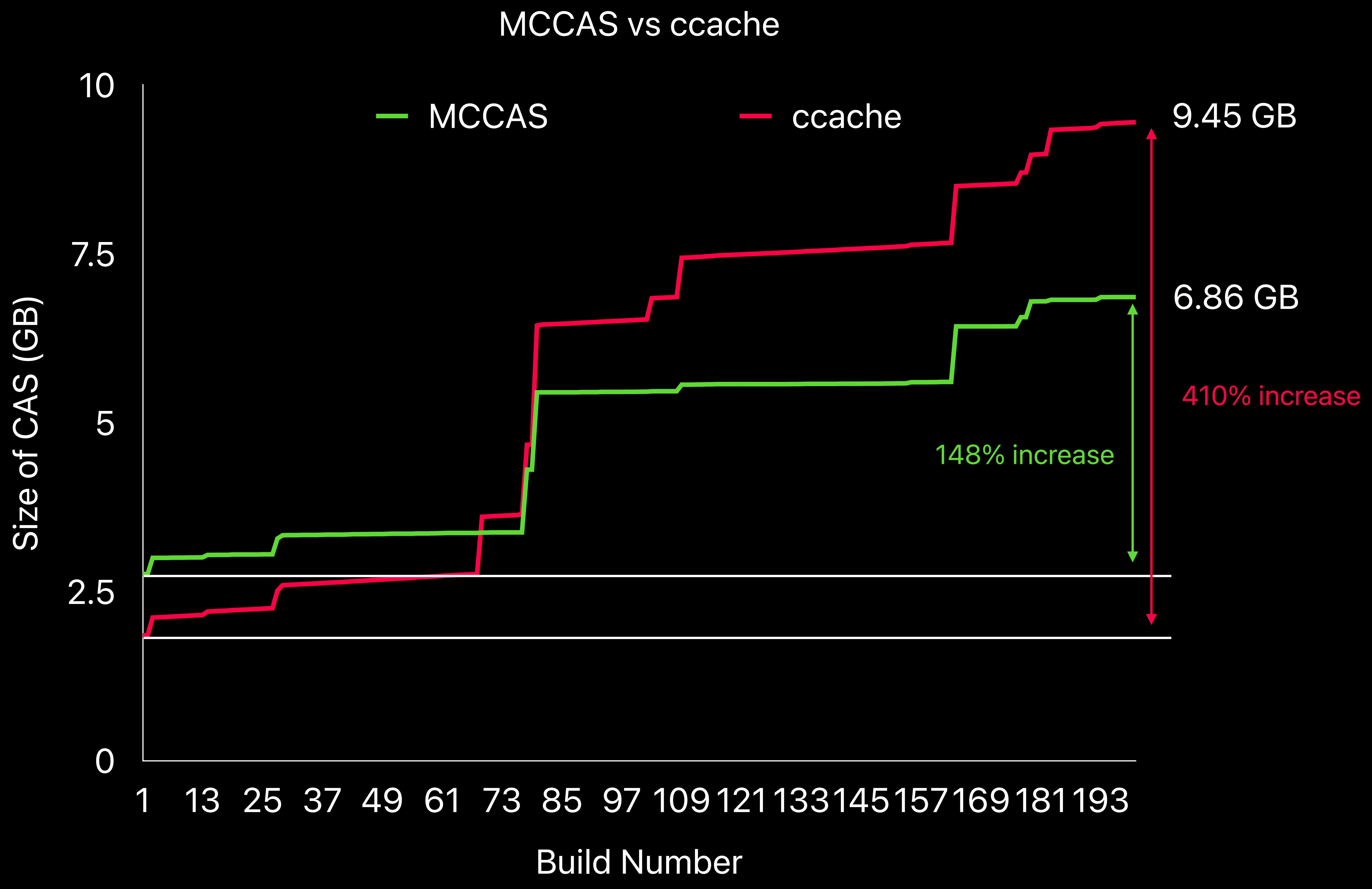
llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

Debug Information representation improvements: Results



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

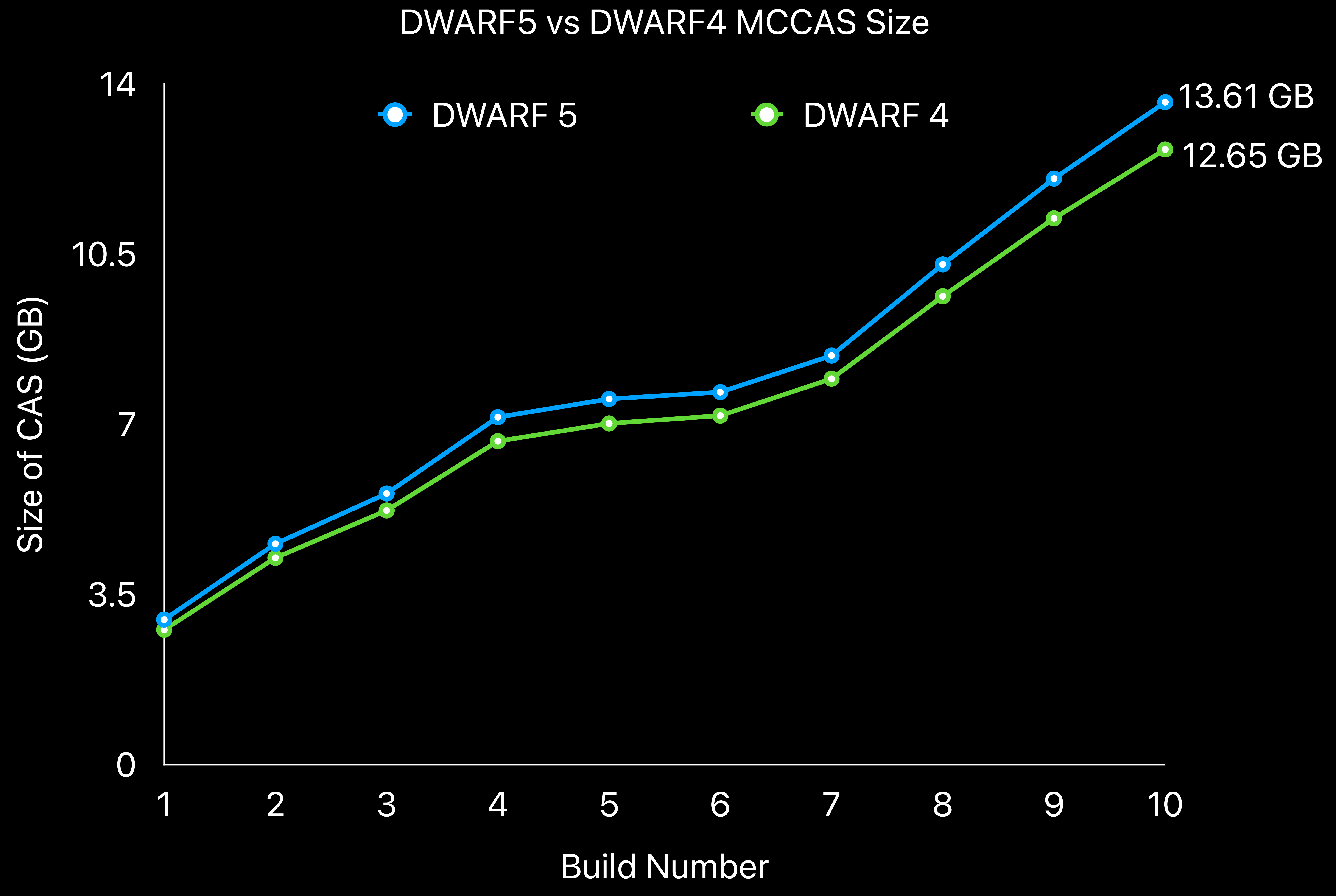
Debug Information representation improvements: Results



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

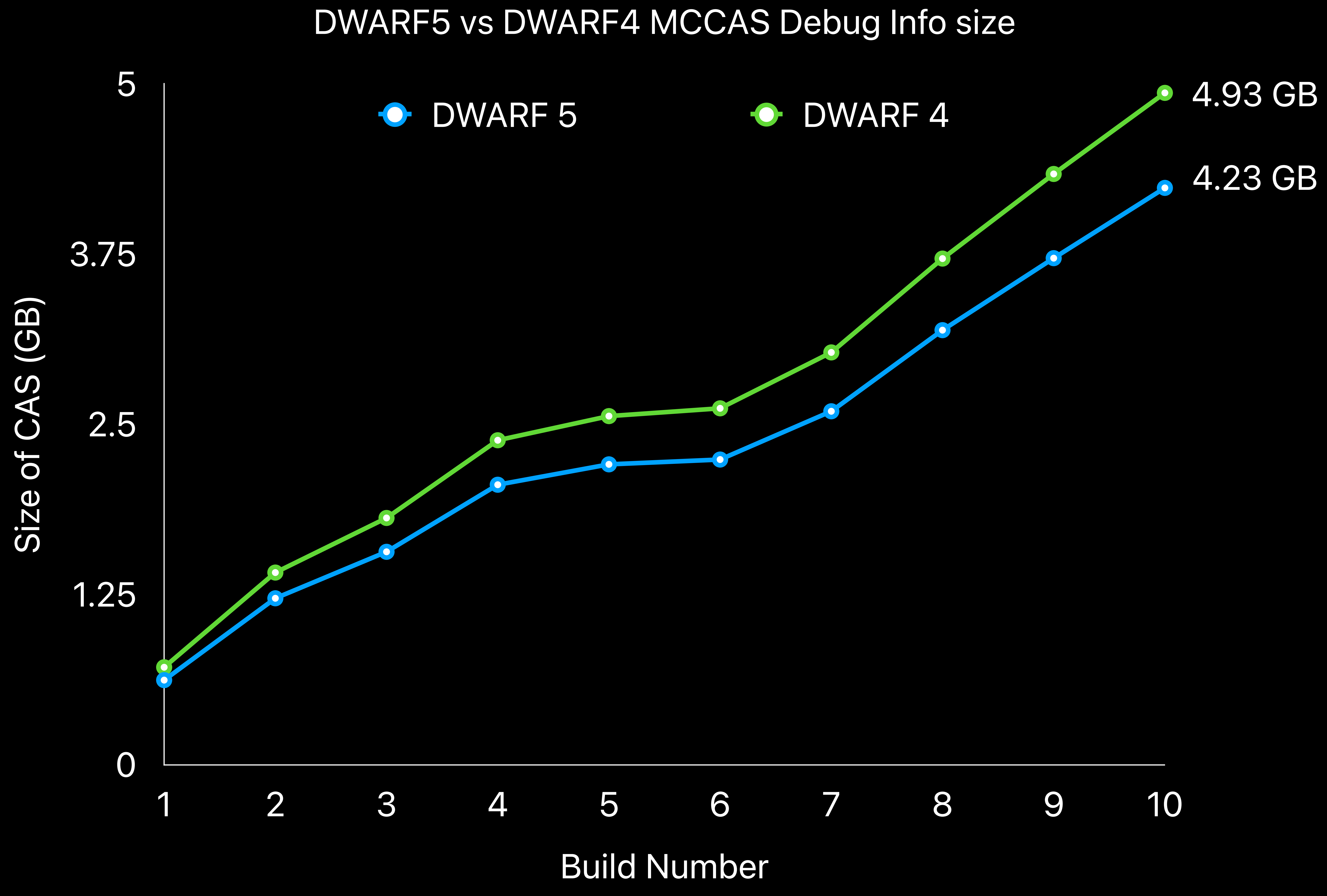
Support for DWARF5 in MCCAS

Support for DWARF5 in MCCAS: Results



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

Support for DWARF5 in MCCAS: Results



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

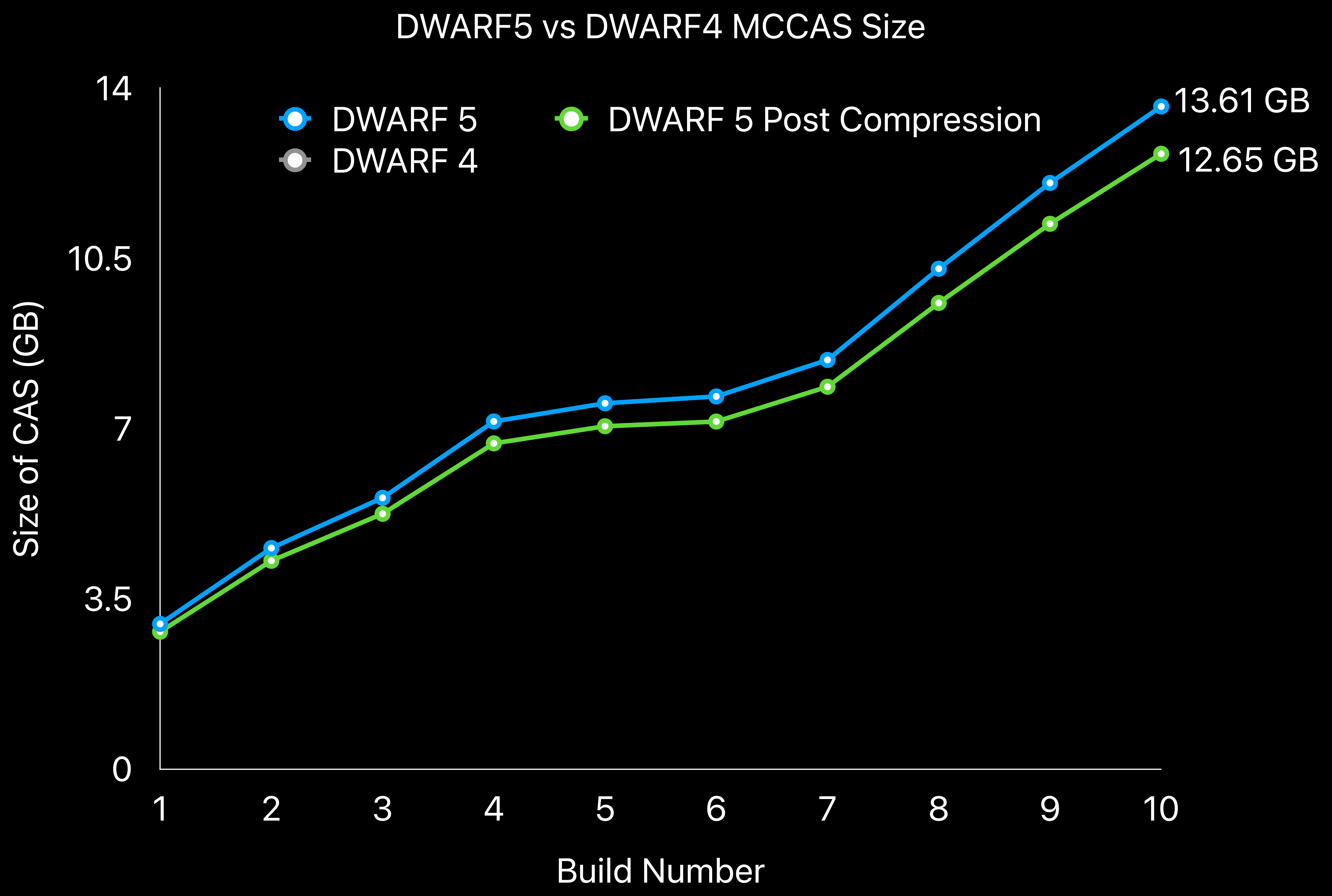
Support for DWARF5 in MCCAS: Results

- DWARF5 CAS 7% > DWARF4 CAS
- Reason is `.debug_str_offsets` section in DWARF5

Support for DWARF5 in MCCAS: Results

- DWARF5 CAS 7% > DWARF4 CAS
- Reason is `.debug_str_offsets` section in DWARF5
- Zlib compression brings size down to DWARF4 levels

Support for DWARF5 in MCCAS: Results

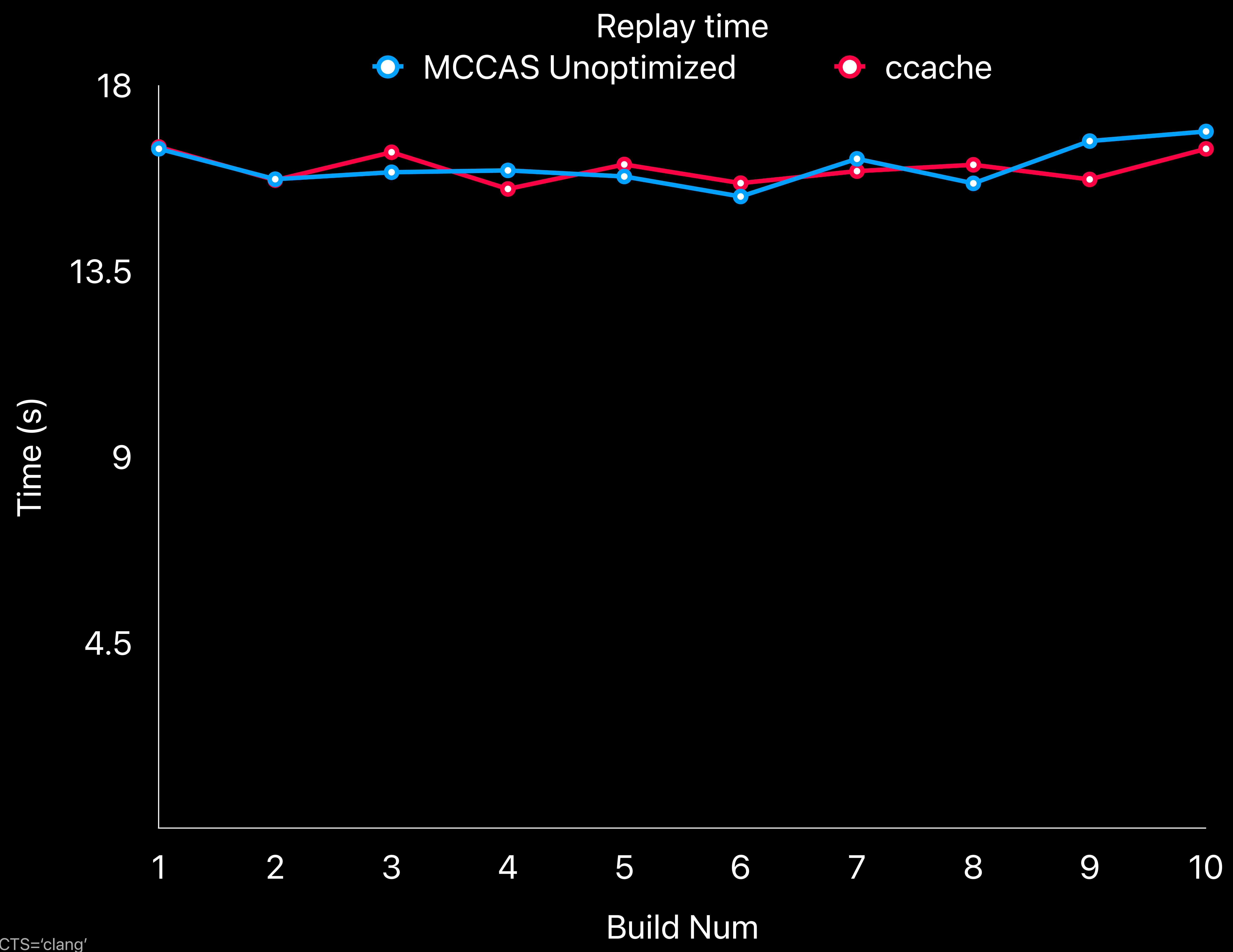


llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

Improvements to replay speed in a MCCAS

- Replay refers to rebuilding a previously cached build

Improvements to replay speed in MCCAS: Results



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

Improvements to replay speed in MCCAS

There are two issues with replay speed that we identified

- Materializing the same abbreviations multiple times
- The ULEB decoder was not optimal

Improvements to replay speed in MCCAS

Materializing the same abbreviations multiple times

- Debug abbreviations describe the DIEs in the .debug_info section
- Multiple DIEs can be described by one abbreviation
- Number of abbreviations is always ≤ Number of DIEs

Improvements to replay speed in MCCAS

Materializing the same abbreviations multiple times

- The problem: We were materializing an abbreviation for a DIE every time we wanted to materialize the DIE
- Materialization is expensive, it requires lots of ULEB decoding

Improvements to replay speed in MCCAS

Materializing the same abbreviations multiple times

- Solution: Materialize all abbreviations once, and memoize them
- Cuts down on materialization time for the object file significantly

Improvements to replay speed in MCCAS

The ULEB decoder was not optimal

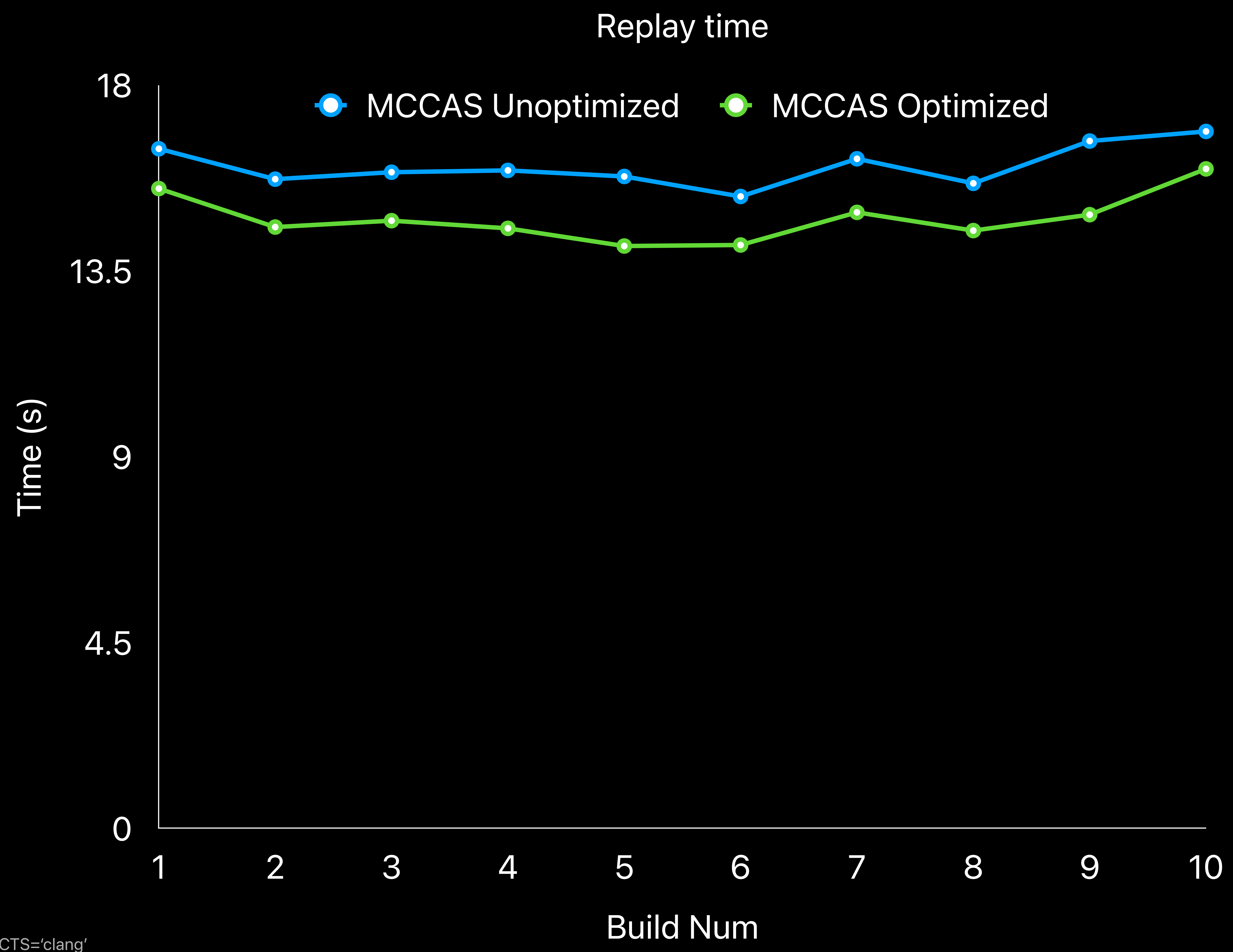
- Materializing any debug info or abbreviations requires ULEB decoding
- The ULEB decoder being used was part of BinaryStreamReader
- BinaryStreamReader is not optimal because it doesn’t guarantee that it’s stream is contiguous
- However, all the CAS Objects that we are reading from, are contiguous

Improvements to replay speed in MCCAS

The ULEB decoder was not optimal

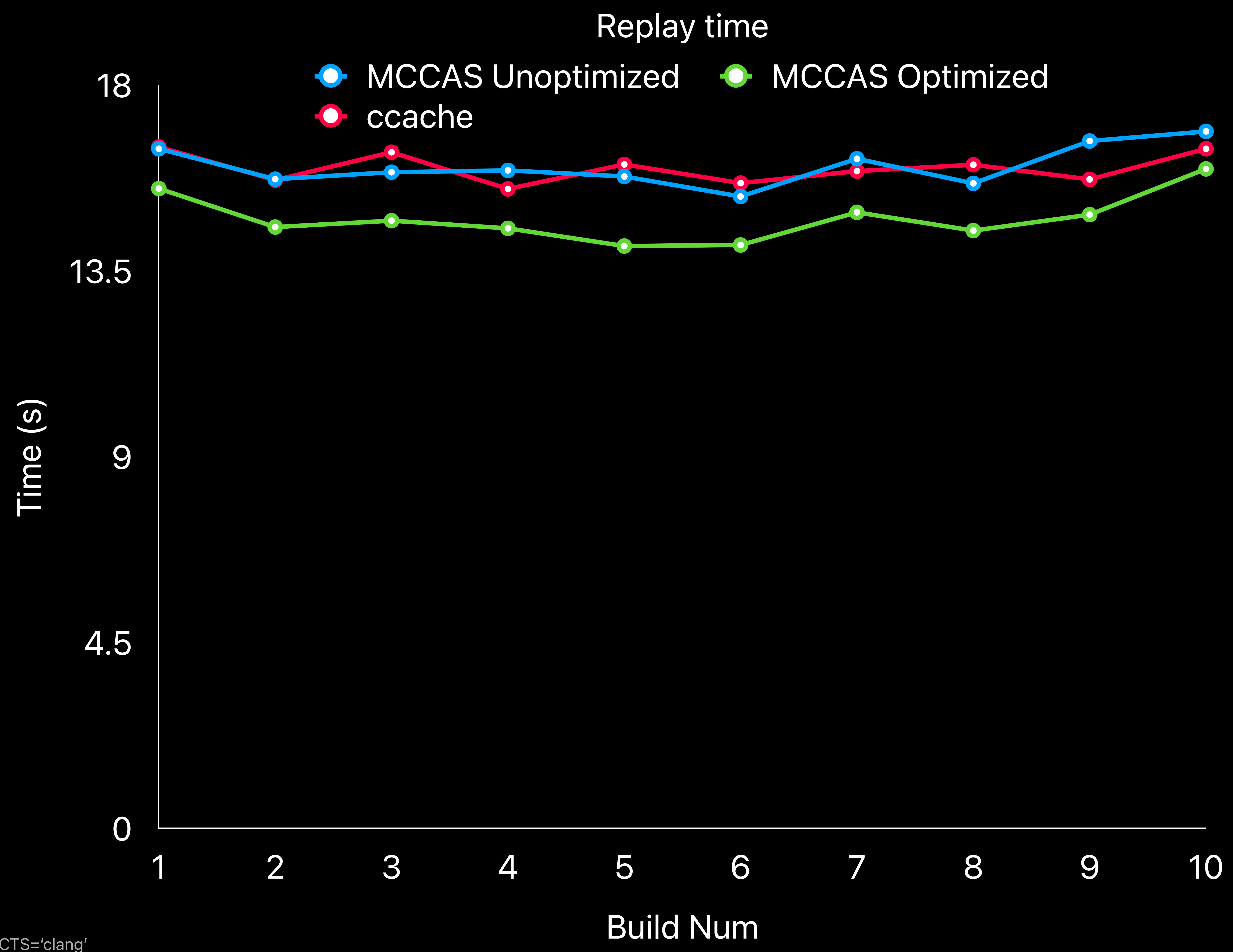
- Solution: Replace BinaryStreamReader with DataExtractor

Improvements to replay speed in MCCAS: Results



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

Improvements to replay speed in MCCAS: Results



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='clang'

MCCAS Support for Swift

- The Swift compiler also supports MCCAS
- Currently, it has been tested with a small open-source project called AlamoFire and works fine
- Further testing is needed to ensure it works correctly

Conclusions

- MCCAS demonstrates a real world use-case for having a CAS-library in LLVM

Conclusions

- MCCAS demonstrates a real world use-case for having a CAS-library in LLVM
- Having a CAS library built into the compiler is advantageous, we can cache clang-modules

LLVM Dev 2023: Caching Explicit Clang Modules with Content-Addressable Storage
https://www.youtube.com/watch?v=6P9787H_SlQ

Future work

- Test and Benchmark MCCAS for Swift
- Implement CAS-specific optimizations for other DWARF sections, such as:
  - `.debug_loc`
  - `.debug_ranges`

Want to contribute?

- llvm-cas initial patch
  - LLVMCAS Implementation: https://github.com/llvm/llvm-project/pull/68448