Representing Debug Information in LLVM CAS

Shubham Rastogi

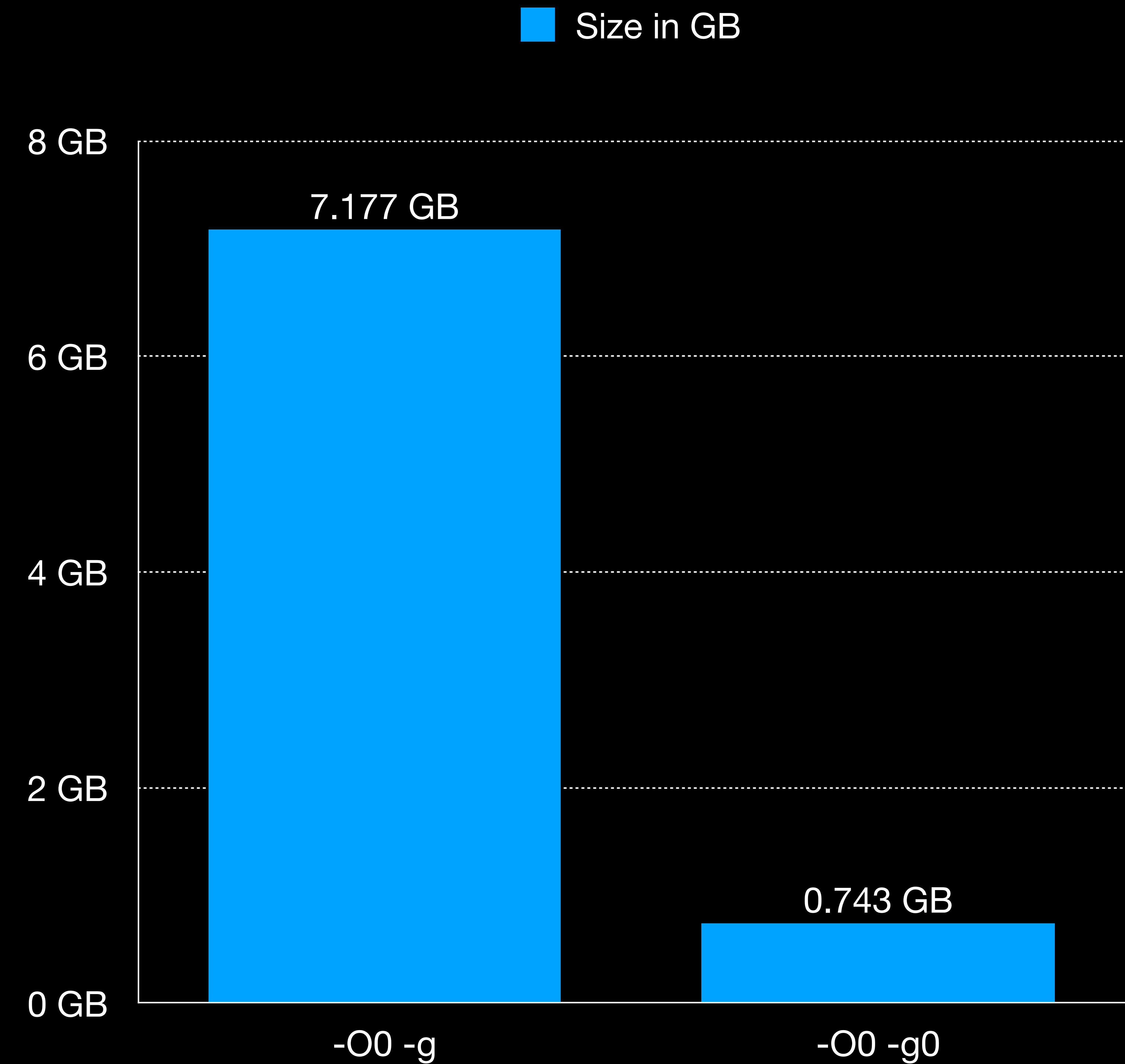
September 14, 2023

Agenda

- Caching builds efficiently using a Content-Addressable Storage (CAS)
- Specifically, how to make DWARF debug info cache-able

Current Problems with Debug Info

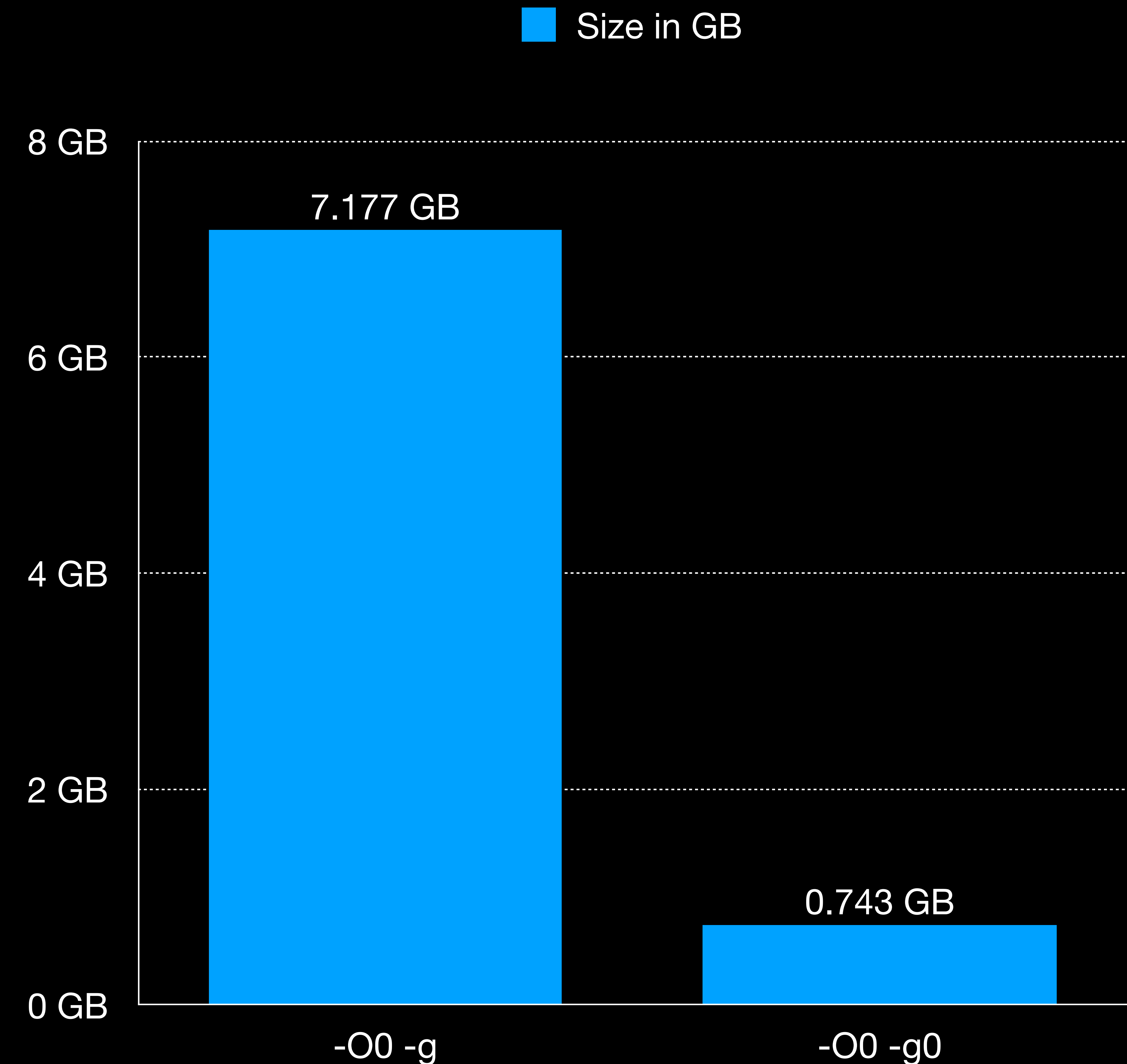
- Debug info is the bulk of an object file's content



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='llvm', ninja

Current Problems with Debug Info

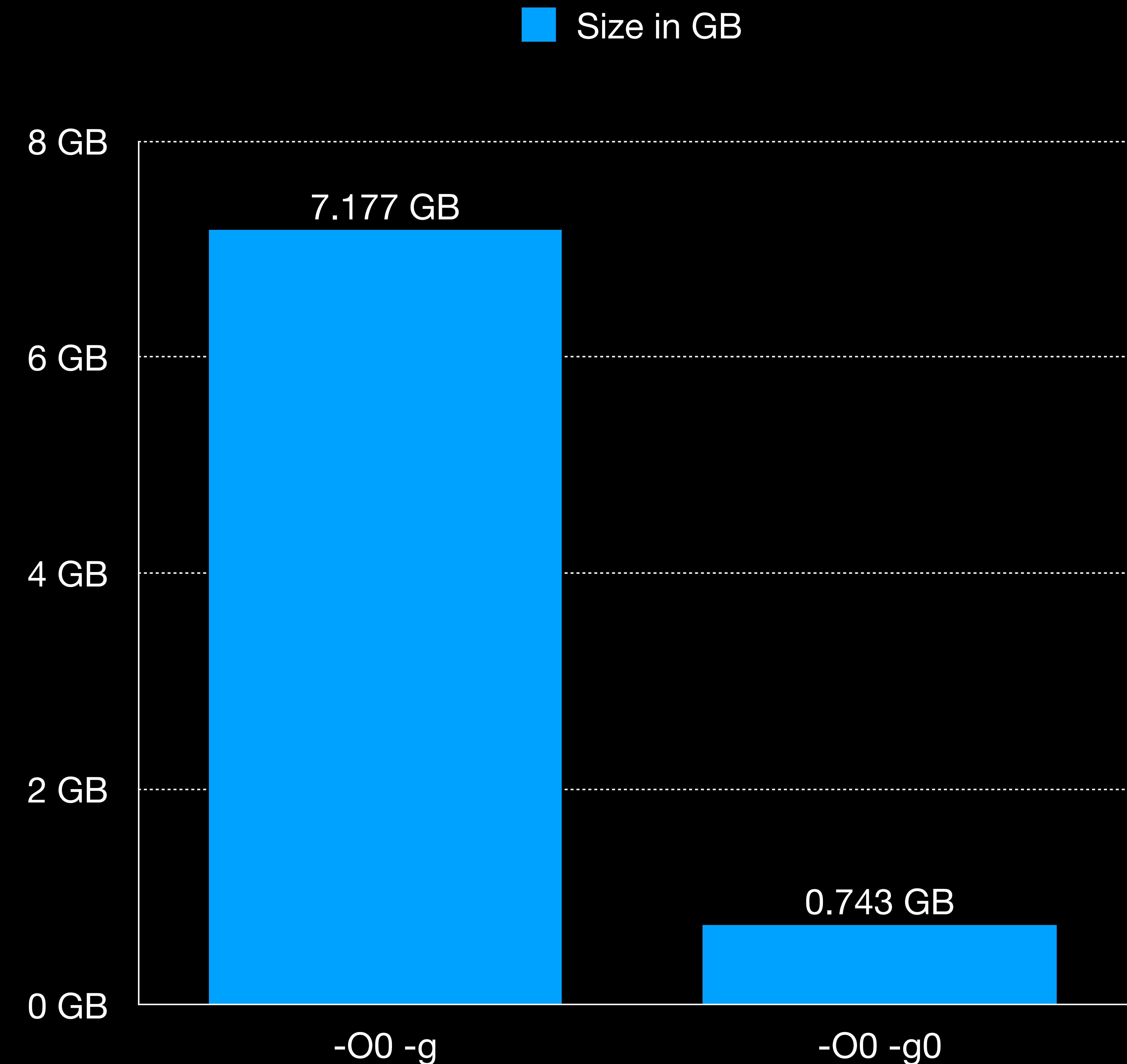
- Debug info is the bulk of an object file's content
- It accounts for ~90% of object file content



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='llvm', ninja

Current Problems with Debug Info

- Debug info is the bulk of an object file's content
- It accounts for ~90% of object file content
- It is redundant, i.e. there is a lot of the same information stored in multiple object files



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='llvm', ninja

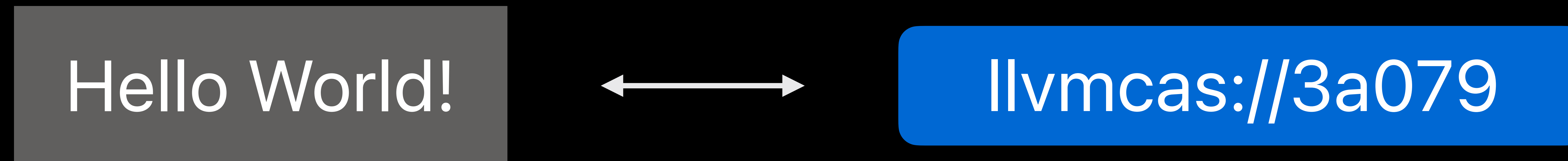
CAS Object Store

CAS object address = hash of contents

CAS Object Store

CAS object address = hash of contents

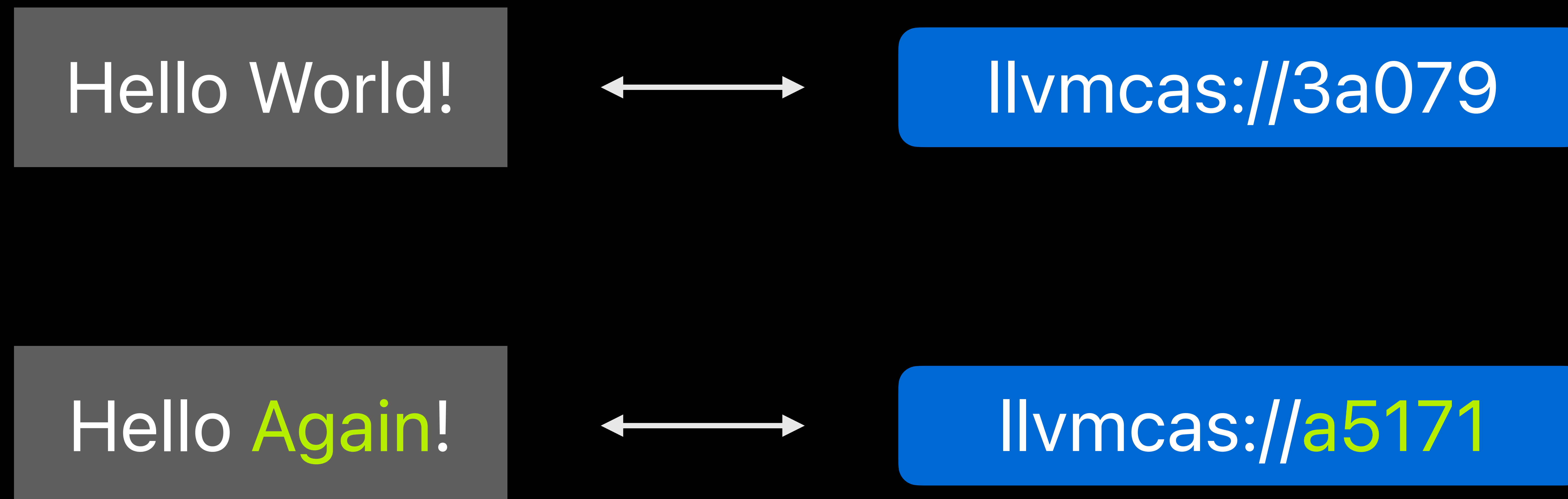
1:1 mapping



CAS Object Store

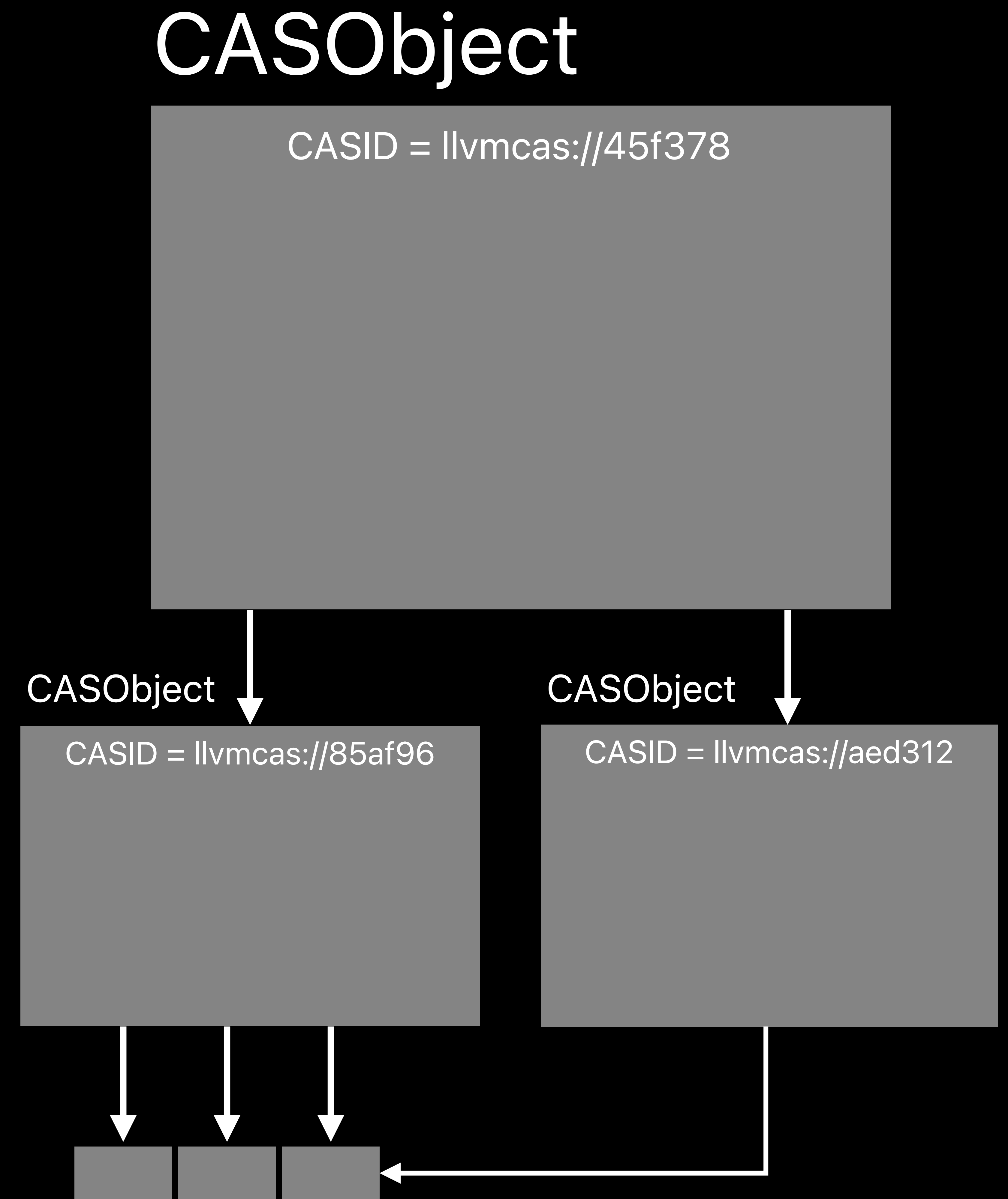
CAS object address = hash of contents

1:1 mapping



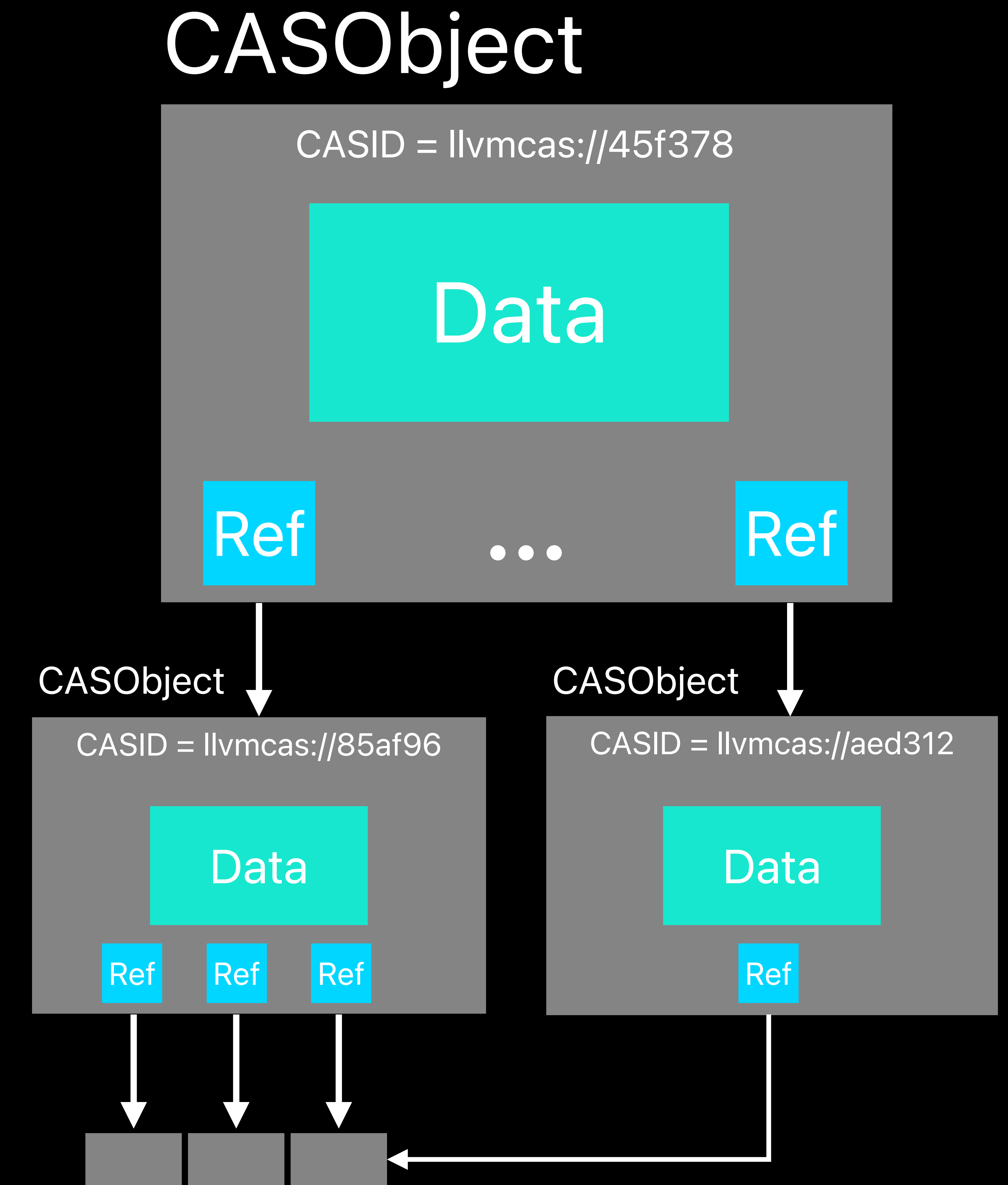
Representation of Content in the CAS

- Content is a DAG of CASObjects



Representation of Content in the CAS

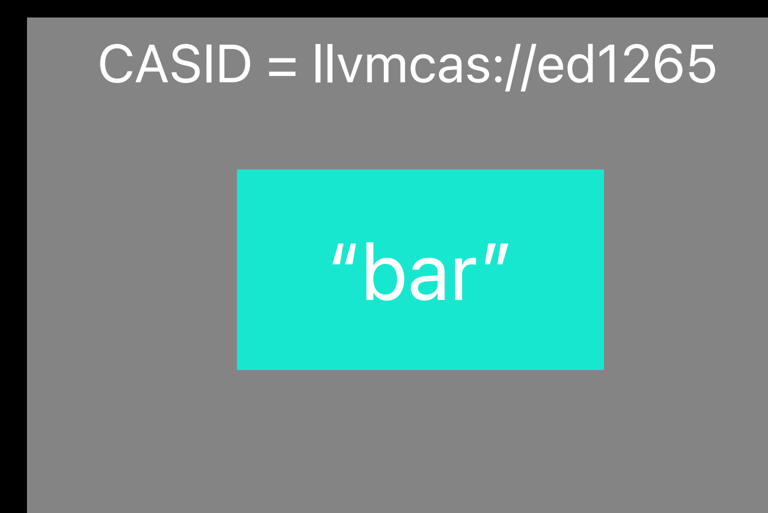
- Content is a DAG of CASObjects
- Each CASObject has data and a list of references to other CASObjects



Deduplication explained

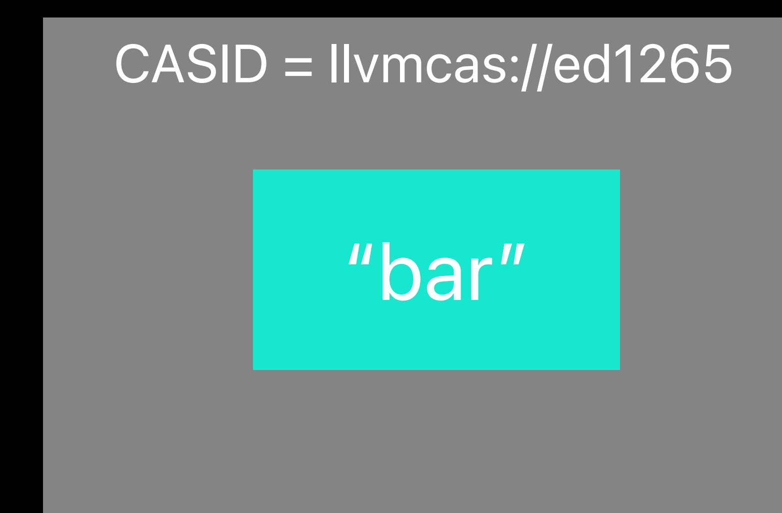
CASObject

CASObject



CASObject

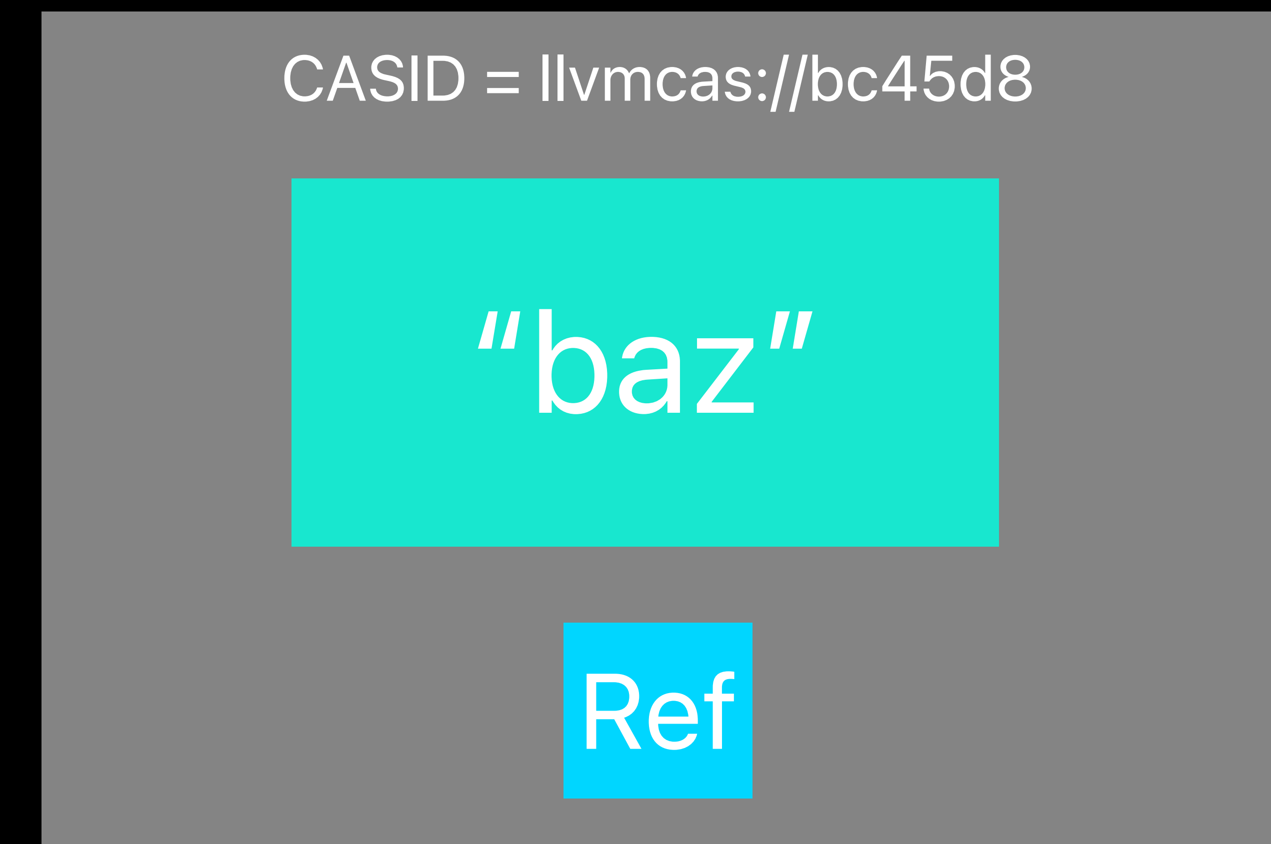
CASObject



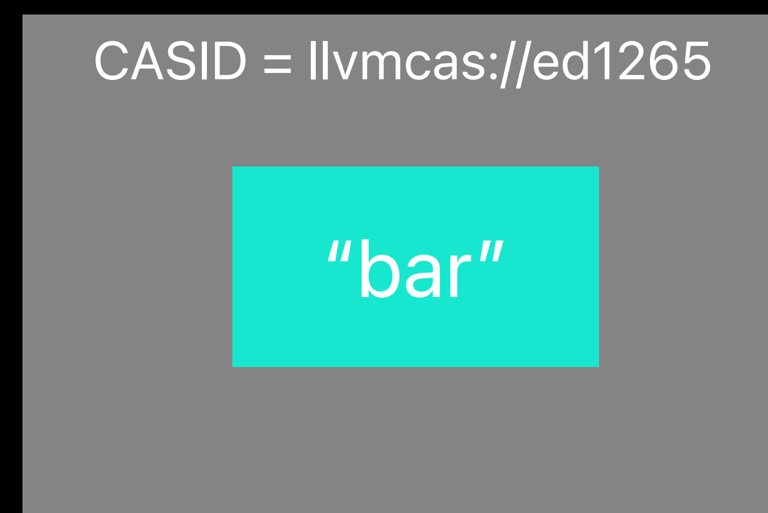
Deduplication explained

CASObject

CASObject



CASObject



Why use a CAS?

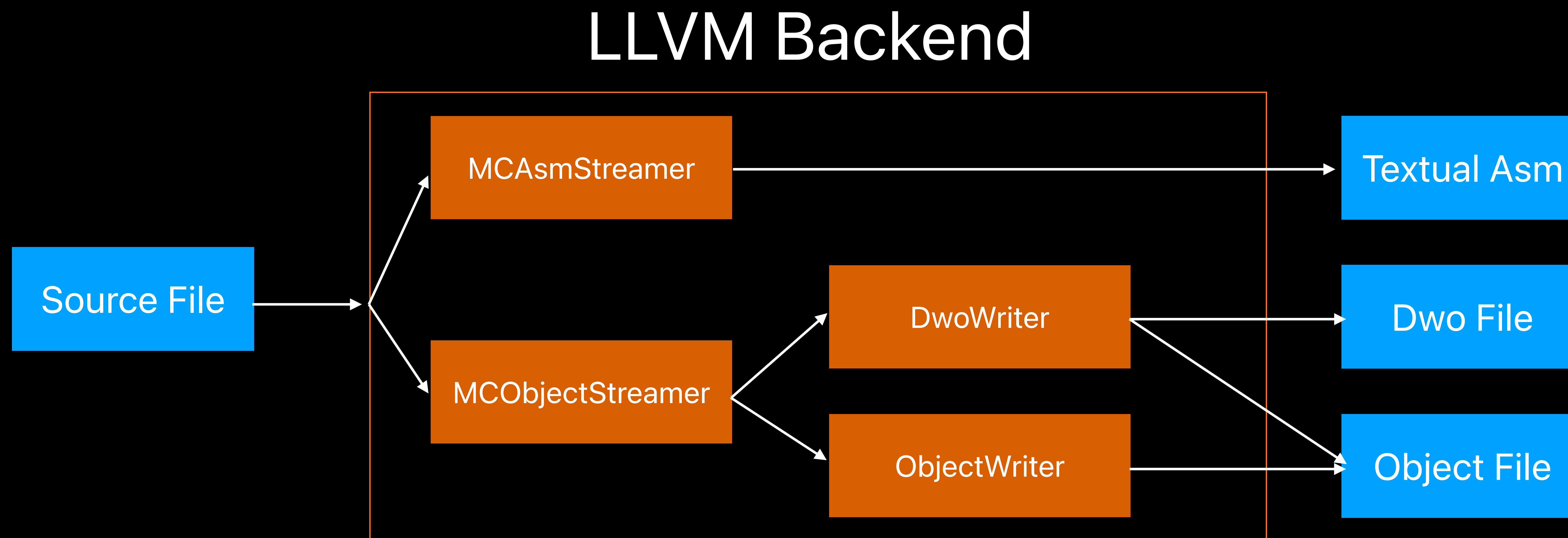
- We want to use a CAS to create a build cache, comparable to ccache
- ccache granularity: object files
- We want to split object files into multiple CASObjects for finer-grained caching

Representation of the CAS

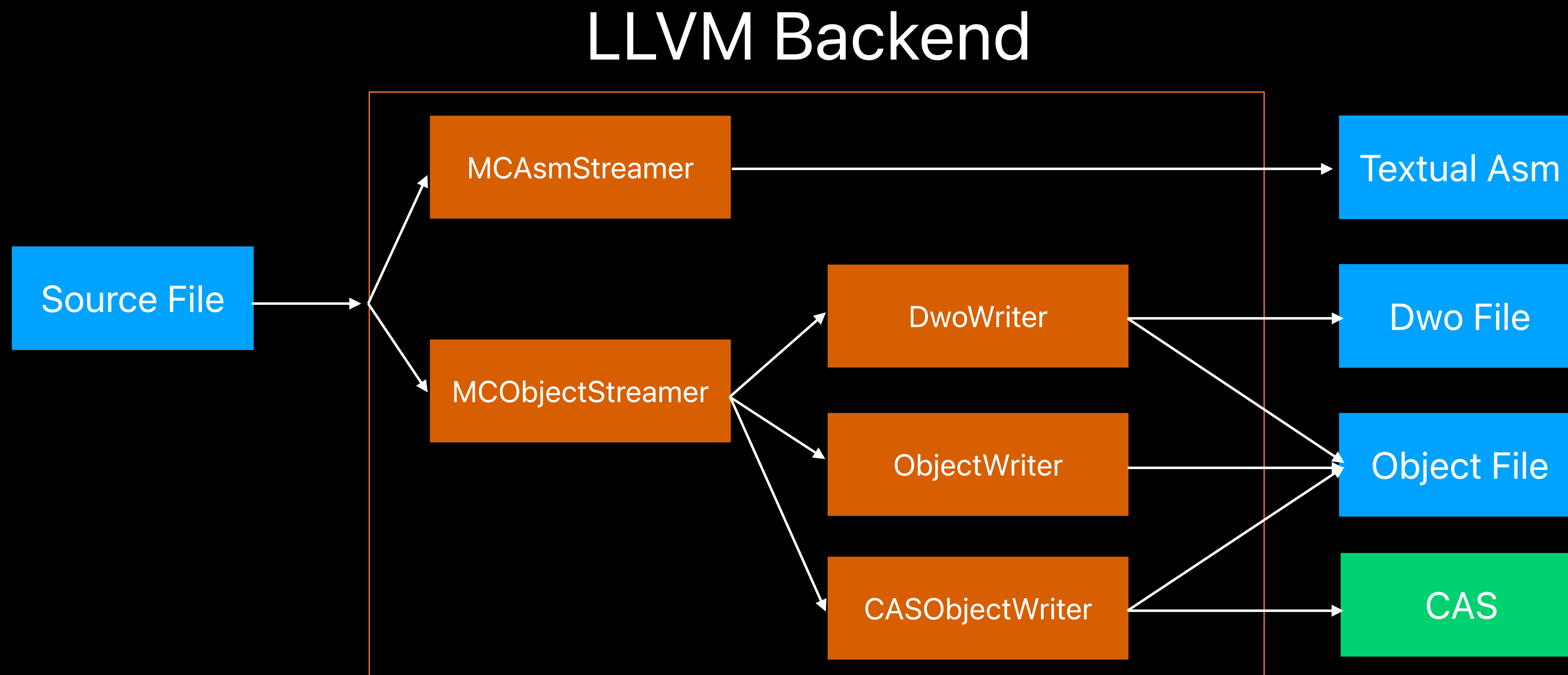
LLVM Dev 2022: Using Content-Addressable Storage in Clang for Caching Computations and Eliminating Redundancy
https://www.youtube.com/watch?v=E9GdNKjGZ7Y

RFC: Add an LLVM CAS library and experiment with fine-grained caching for builds
https://discourse.llvm.org/t/rfc-add-an-llvm-cas-library-and-experiment-with-fine-grained-caching-for-builds/59864

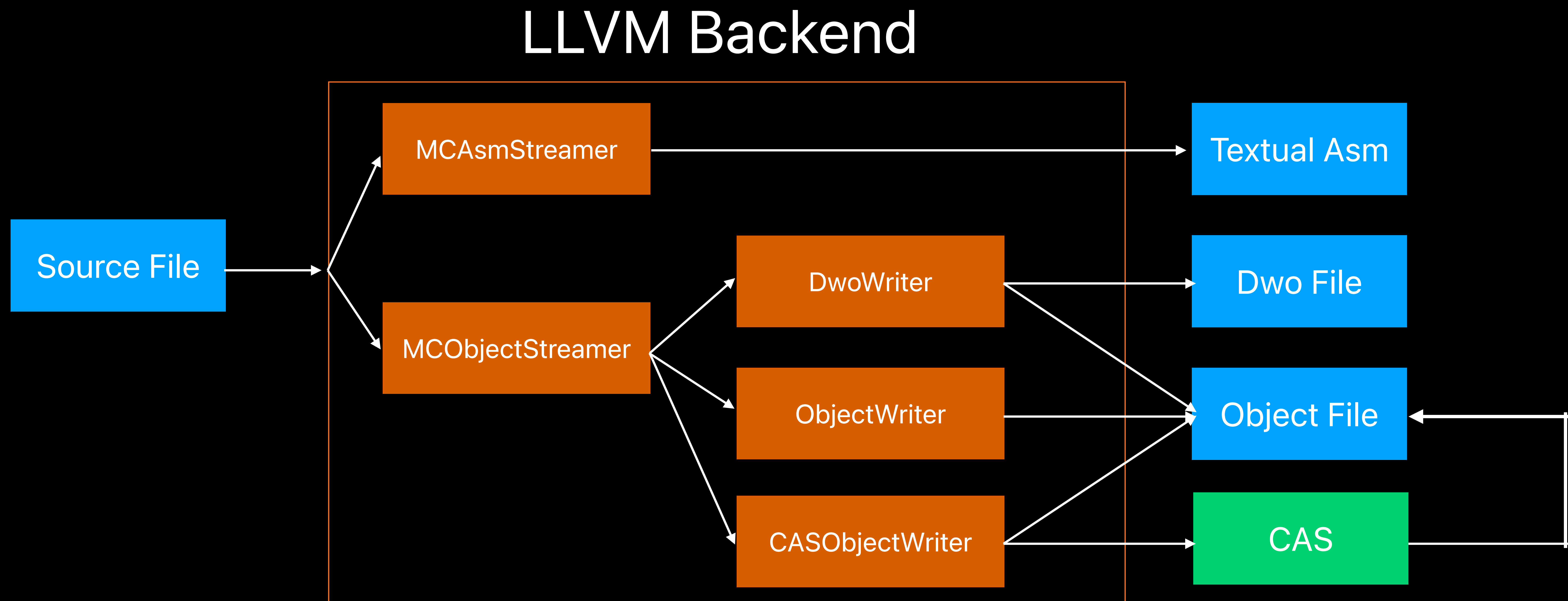
Current build workflow vs CAS Workflow



Current build workflow vs CAS Workflow



Current build workflow vs CAS Workflow



Contents of an object file

- An object file contains multiple sections

Contents of an object file

- An object file contains multiple sections
- For example, Mach-O groups sections into segments

Object file contents

Debug Information Representation

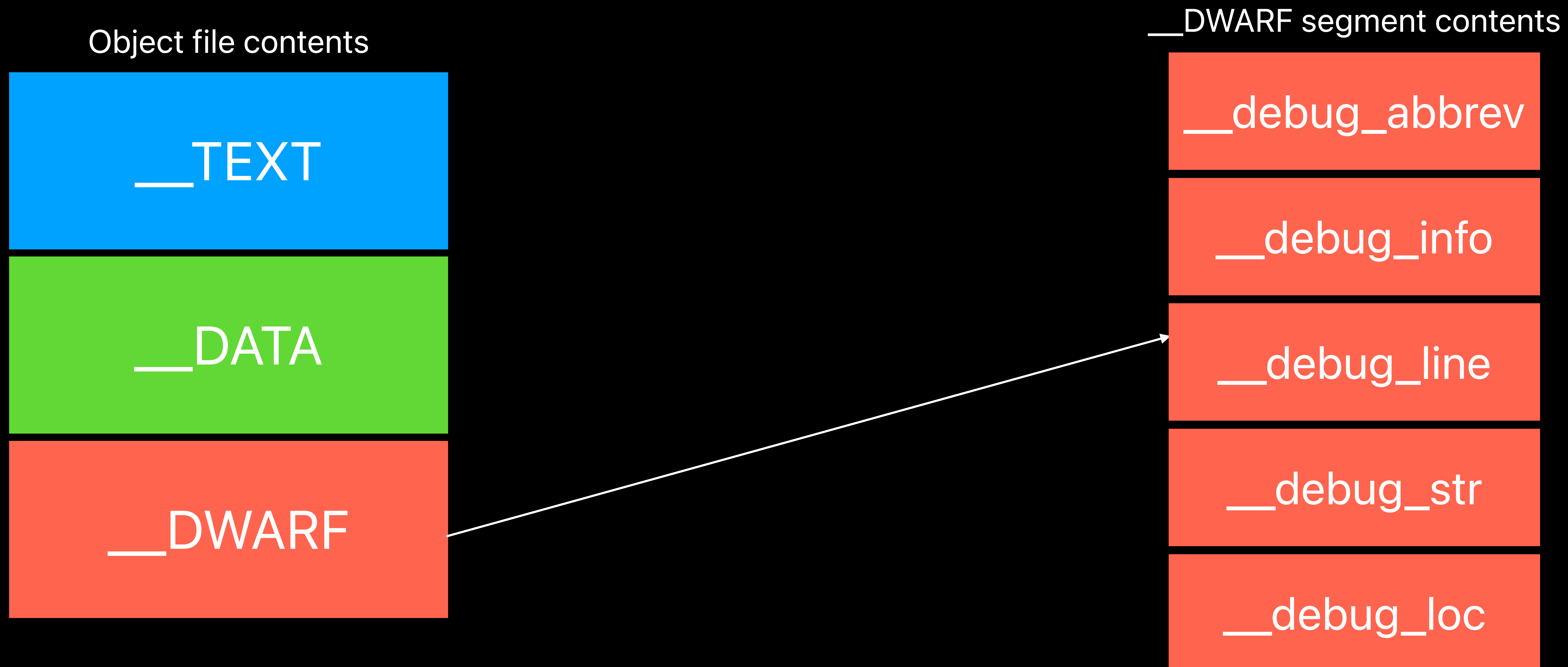
- The Debug Information is part of the __DWARF segment

Object file contents

__TEXT
__DATA
__DWARF

Debug Information Representation

- The Debug Information is part of the __DWARF segment
- The __DWARF segment can be further divided into multiple sections



Making __DWARF sections more CAS-friendly

__debug_str representation

```cpp
// a.cpp

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

```cpp
// b.cpp

int baz() {
    return 1;
}
```

Example, __debug_str representation contd...

```
dwarfdump a.o --debug-str
a.o:    file format Mach-O arm64

.debug_str contents:
0x00000000: "clang version 18.0.0 …"
0x00000065: "a.cpp"
0x0000006b: "/"
0x0000006d: "/Users/shubham/Development"
0x00000094: "func"
0x00000099: "_Z4funci"
0x000000a2: "int"
0x000000a6: "x"
```

```
dwarfdump b.o --debug-str
b.o:    file format Mach-O arm64

.debug_str contents:
0x00000000: "clang version 18.0.0 …"
0x00000065: "b.cpp"
0x0000006b: "/"
0x0000006d: "/Users/shubham/Development"
0x00000094: "baz"
0x00000098: "_Z3bazv"
0x000000a0: "int"
```

Strings are stored one after the other with the offset of the string referenced in other sections

Example, __debug_str representation contd…

```
dwarfdump a.o --debug-str
a.o:    file format Mach-O arm64

.debug_str contents:
0x00000000: "clang version 18.0.0 …"
0x00000065: "a.cpp"
0x0000006b: "/"
0x0000006d: "/Users/shubham/Development"
0x00000094: "func"
0x00000099: "_Z4funci"
0x000000a2: "int"
0x000000a6: "x"
```

```
dwarfdump b.o --debug-str
b.o:    file format Mach-O arm64

.debug_str contents:
0x00000000: "clang version 18.0.0 …"
0x00000065: "b.cpp"
0x0000006b: "/"
0x0000006d: "/Users/shubham/Development"
0x00000094: "baz"
0x00000098: "_Z3bazv"
0x000000a0: "int"
```

CAS Representation of __debug_str section

CAS representation for a.o

debug_str

```
"clang version …"
"a.cpp"
…
"int"
```

CAS representation for b.o

debug_str

```
"clang version …"
"b.cpp"
…
"int"
```

Storing __debug_str in the CAS efficiently

- Idea, why not create one CASObject per string?
- Redundant strings will share the same ID, and deduplicate
- Need to be careful, CAS References are not free

CAS Representation of __debug_str section

CAS representation for a.o

debug_str

```
"clang version …"
"a.cpp"
…
"int"
```

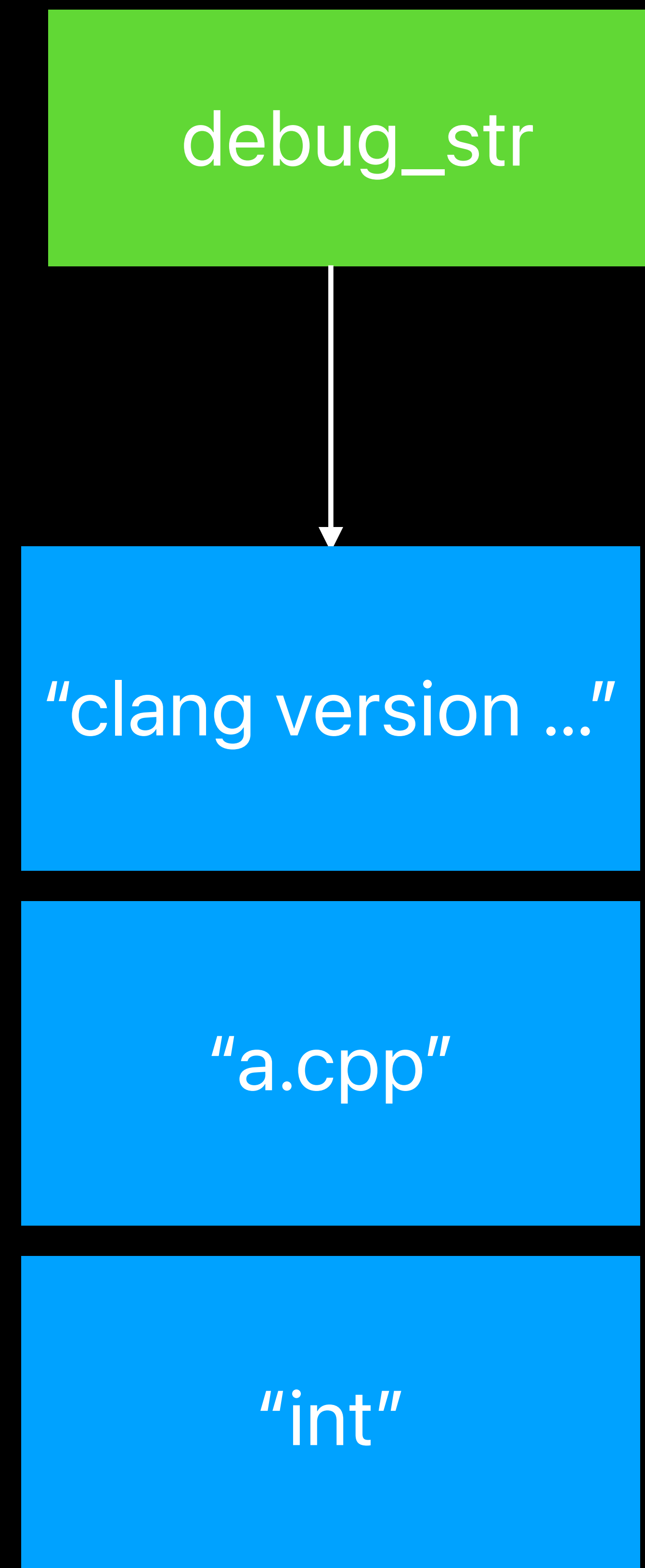
CAS representation for b.o

debug_str

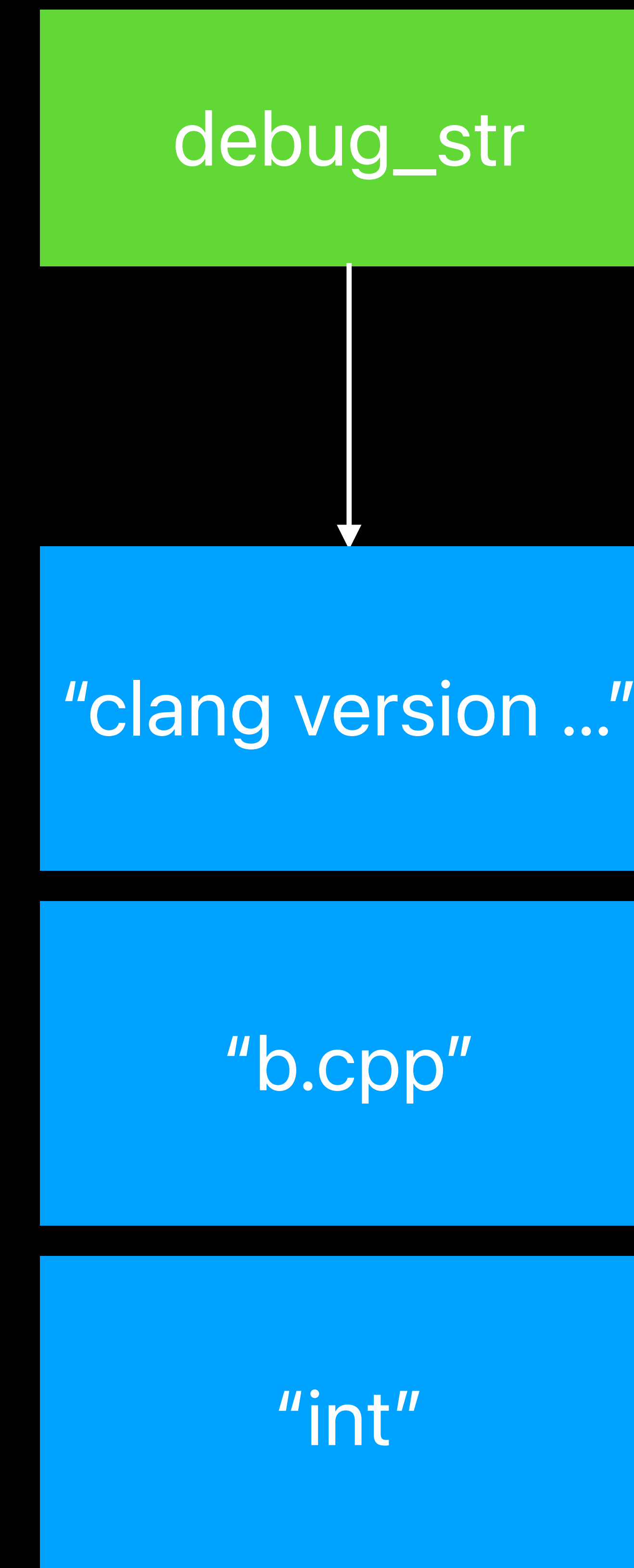
```
"clang version …"
"b.cpp"
…
"int"
```

CAS Representation of __debug_str section

CAS representation for a.o

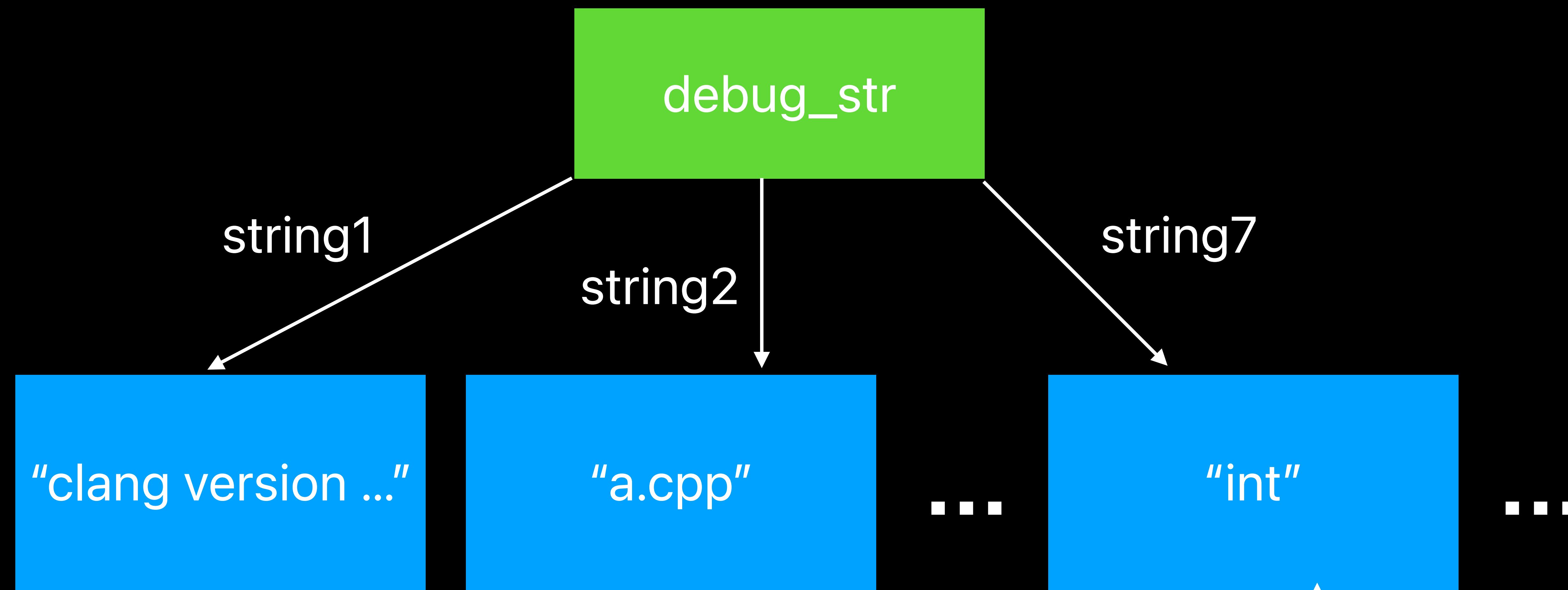


CAS representation for b.o



CAS Representation of __debug_str section

CAS representation for a.o



CAS representation for b.o

debug_str

string7

string1

string2

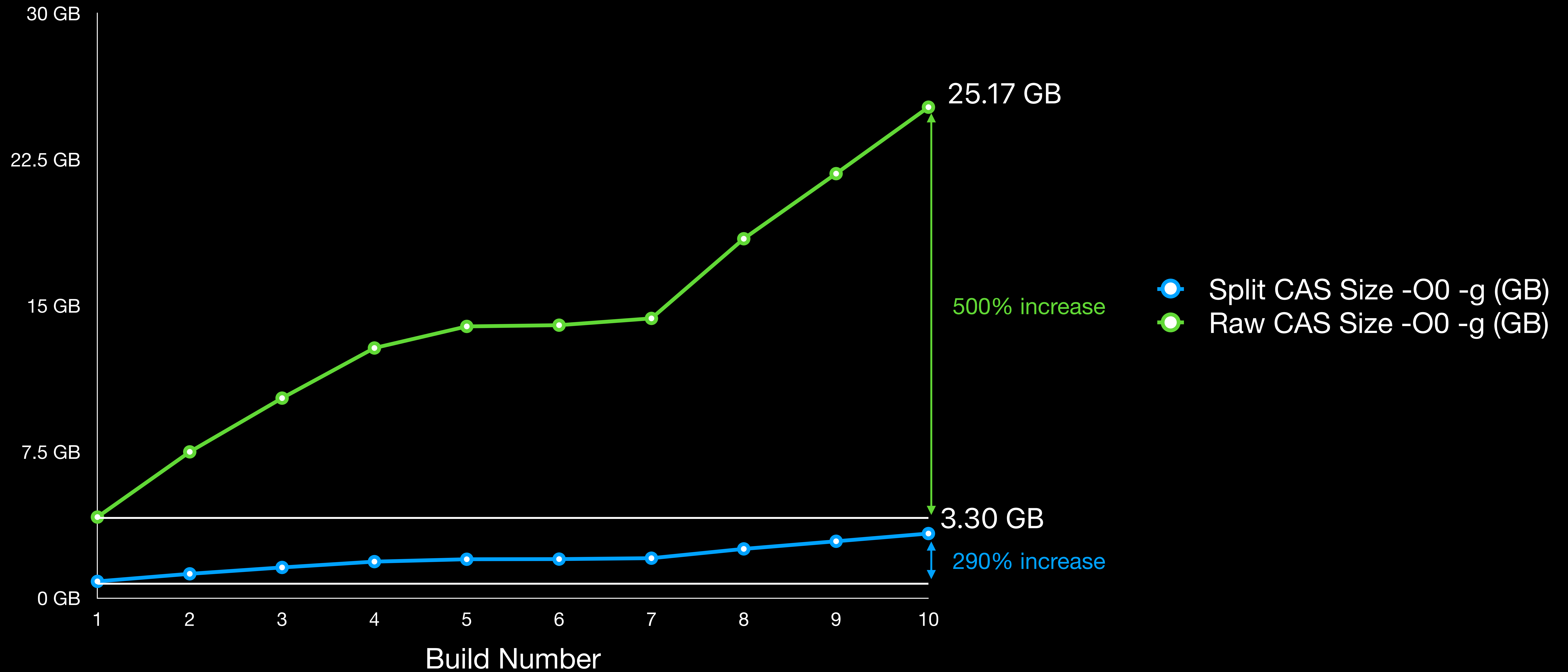
“b.cpp”

…

Evaluation Methodology

- To simulate incremental builds, 10 commits of llvm-project.git from 10 consecutive days were taken at the start of every day
- All of llvm was built with the resulting object files being ingested into the same CAS

Results from storing __debug_str in the CAS



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='llvm', ninja

__debug_str section representation

__debug_line representation

```
1  // a.cpp
2
3  #include "foo.h"
4  int bar(int x) {
5    return x+foo(x);
6  }
```

```
1  // b.cpp
2
3  int b() {
4    return 1;
5  }
6
7  #include “foo.h"
8
9  int main() {
10   return b()+foo(3);
11 }
```

```
1  // foo.h
2
3  int foo(int y) {
4    return 1;
5  }
```

What does the __debug_line section look like?

What does the __debug_line section look like?

- The line table maps source code addresses to source line numbers
- Mappings are compressed into opcodes to a state machine which is used to expand the line table by the debugger

Example, __debug_line representation contd...

```
dwarfdump --debug-line a.o
a.o:      file format Mach-O arm64

< line table header >

Address            Line   Column File   ISA Flags
------------------ ------ ------ ------ --- -------------
0x0000000000000000      3      0      1   0  is_stmt
0x0000000000000008      4      3      1   0  is_stmt
0x000000000000000c      4      3      1   0  epilogue_begin
0x0000000000000014      4      0      2   0  is_stmt
0x0000000000000024      5     10      2   0  is_stmt
0x000000000000002c      5     16      2   0
0x0000000000000030      5     12      2   0
0x0000000000000038      5     11      2   0
0x000000000000003c      5      3      2   0  epilogue_begin
0x0000000000000048      5      3      2   0  end_sequence
```

```
dwarfdump --debug-line b.o
b.o:      file format Mach-O arm64

< line table header >

Address            Line   Column File   ISA Flags
------------------ ------ ------ ------ --- -------------
0x0000000000000000      4      5      1   0  is_stmt
prologue_end
0x0000000000000008      3      0      2   0  is_stmt
0x0000000000000010      4      3      2   0  is_stmt
0x0000000000000014      4      3      2   0  epilogue_begin
0x000000000000001c      9      0      1   0  is_stmt
0x000000000000002c     10     12      1   0  is_stmt
0x0000000000000034     10     20      1   0
0x0000000000000044     10     18      1   0
0x0000000000000048     10      5      1   0  epilogue_begin
0x0000000000000054     10      5      1   0  end_sequence
```

This is the line table representation when expanded from the metadata stored in the __debug_line section

Note: Line table header not shown

Example, __debug_line representation contd...

```
dwarfdump --debug-line a.o
a.o:      file format Mach-O arm64

< line table header >

Address            Line   Column File   ISA Flags
------------------ ------ ------ ------ --- -------------
0x0000000000000000      3      0      1   0  is_stmt
0x0000000000000008      4      3      1   0  is_stmt
0x000000000000000c      4      3      1   0  epilogue_begin
0x0000000000000014      4      0      2   0  is_stmt
0x0000000000000024      5     10      2   0  is_stmt
0x000000000000002c      5     16      2   0
0x0000000000000030      5     12      2   0
0x0000000000000038      5     11      2   0
0x000000000000003c      5      3      2   0  epilogue_begin
0x0000000000000048      5      3      2   0  end_sequence
```

```
dwarfdump --debug-line b.o
b.o:      file format Mach-O arm64

< line table header >

Address            Line   Column File   ISA Flags
------------------ ------ ------ ------ --- -------------
0x0000000000000000      4      5      1   0  is_stmt
prologue_end
0x0000000000000008      3      0      2   0  is_stmt
0x0000000000000010      4      3      2   0  is_stmt
0x0000000000000014      4      3      2   0  epilogue_begin
0x000000000000001c      9      0      1   0  is_stmt
0x000000000000002c     10     12      1   0  is_stmt
0x0000000000000034     10     20      1   0
0x0000000000000044     10     18      1   0
0x0000000000000048     10      5      1   0  epilogue_begin
0x0000000000000054     10      5      1   0  end_sequence
```

Line table header is at the beginning and does not dedupe

Note: Line table header not shown

Example, __debug_line representation contd...

```
dwarfdump --debug-line a.o
a.o:      file format Mach-O arm64

< line table header >

Address            Line   Column File   ISA Flags
------------------ ------ ------ ------ --- -------------
0x0000000000000000      3      0      1   0  is_stmt
0x0000000000000008      4      3      1   0  is_stmt
0x000000000000000c      4      3      1   0  epilogue_begin
0x0000000000000014      4      0      2   0  is_stmt
0x0000000000000024      5     10      2   0  is_stmt
0x000000000000002c      5     16      2   0
0x0000000000000030      5     12      2   0
0x0000000000000038      5     11      2   0
0x000000000000003c      5      3      2   0  epilogue_begin
0x0000000000000048      5      3      2   0  end_sequence
```

foo (first three rows marked)

```
dwarfdump --debug-line b.o
b.o:      file format Mach-O arm64

< line table header >

Address            Line   Column File   ISA Flags
------------------ ------ ------ ------ --- -------------
0x0000000000000000      4      5      1   0  is_stmt
prologue_end
0x0000000000000008      3      0      2   0  is_stmt
0x0000000000000010      4      3      2   0  is_stmt
0x0000000000000014      4      3      2   0  epilogue_begin
0x000000000000001c      9      0      1   0  is_stmt
0x000000000000002c     10     12      1   0  is_stmt
0x0000000000000034     10     20      1   0
0x0000000000000044     10     18      1   0
0x0000000000000048     10      5      1   0  epilogue_begin
0x0000000000000054     10      5      1   0  end_sequence
```

foo (rows 0x08–0x14 marked)

Line table contribution for function “foo” is marked above

Note: Line table header not shown

Actual contents of 'foo' in the __debug_line section

Let's look at the line table contribution for foo in each *.o file

'foo' in a.o

```
0x00000032: 04 DW_LNS_set_file (1)
0x00000034: 00 DW_LNE_set_address (0x0000000000000000)
0x0000003f: 14 address += 0,  line += 2
            0x0000000000000000      3      0      1   0
0  is_stmt
0x00000040: 05 DW_LNS_set_column (3)
0x00000042: 0a DW_LNS_set_prologue_end
0x00000043: 83 address += 8,  line += 1
            0x0000000000000008      4      3      1   0
0  is_stmt prologue_end
0x00000044: 06 DW_LNS_negate_stmt
0x00000045: 0b DW_LNS_set_epilogue_begin
0x00000046: 4a address += 4,  line += 0
            0x000000000000000c      4      3      1   0
0  epilogue_begin
0x00000047: 02 DW_LNS_advance_pc (8)
0x00000049: 00 DW_LNE_end_sequence
            0x0000000000000014      4      3      1   0
0  end_sequence
```

'foo' in b.o

```
0x00000048: 04 DW_LNS_set_file (2)
0x0000004a: 00 DW_LNE_set_address (0x0000000000000008)
0x00000055: 14 address += 0,  line += 2
            0x0000000000000008      3      0      2   0
0  is_stmt
0x00000056: 05 DW_LNS_set_column (3)
0x00000058: 0a DW_LNS_set_prologue_end
0x00000059: 83 address += 8,  line += 1
            0x0000000000000010      4      3      2   0
0  is_stmt prologue_end
0x0000005a: 06 DW_LNS_negate_stmt
0x0000005b: 0b DW_LNS_set_epilogue_begin
0x0000005c: 4a address += 4,  line += 0
            0x0000000000000014      4      3      2   0
0  epilogue_begin
0x0000005d: 02 DW_LNS_advance_pc (8)
0x0000005f: 00 DW_LNE_end_sequence
            0x000000000000001c      4      3      2   0
0  end_sequence
```

This is the line table's raw encoding, that is actually stored in the __debug_line section

Actual contents of 'foo' in the __debug_line section

Let's look at the line table contribution for foo in each *.o file

'foo' in a.o

```
0x00000032: 04 DW_LNS_set_file (1)
0x00000034: 00 DW_LNE_set_address (0x0000000000000000)
0x0000003f: 14 address += 0,  line += 2
            0x0000000000000000      3      0      1   0
0  is_stmt
0x00000040: 05 DW_LNS_set_column (3)
0x00000042: 0a DW_LNS_set_prologue_end
0x00000043: 83 address += 8,  line += 1
            0x0000000000000008      4      3      1   0
0  is_stmt prologue_end
0x00000044: 06 DW_LNS_negate_stmt
0x00000045: 0b DW_LNS_set_epilogue_begin
0x00000046: 4a address += 4,  line += 0
            0x000000000000000c      4      3      1   0
0  epilogue_begin
0x00000047: 02 DW_LNS_advance_pc (8)
0x00000049: 00 DW_LNE_end_sequence
            0x0000000000000014      4      3      1   0
0  end_sequence
```

'foo' in b.o

```
0x00000048: 04 DW_LNS_set_file (2)
0x0000004a: 00 DW_LNE_set_address (0x0000000000000008)
0x00000055: 14 address += 0,  line += 2
            0x0000000000000008      3      0      2   0
0  is_stmt
0x00000056: 05 DW_LNS_set_column (3)
0x00000058: 0a DW_LNS_set_prologue_end
0x00000059: 83 address += 8,  line += 1
            0x0000000000000010      4      3      2   0
0  is_stmt prologue_end
0x0000005a: 06 DW_LNS_negate_stmt
0x0000005b: 0b DW_LNS_set_epilogue_begin
0x0000005c: 4a address += 4,  line += 0
            0x0000000000000014      4      3      2   0
0  epilogue_begin
0x0000005d: 02 DW_LNS_advance_pc (8)
0x0000005f: 00 DW_LNE_end_sequence
            0x000000000000001c      4      3      2   0
0  end_sequence
```

DW_LNS_set_file is an opcode that denotes the file number from a list of files in a translation unit

Actual contents of ‘foo’ in the __debug_line section

Let’s look at the line table contribution for foo in each *.o file

‘foo’ in a.o

```
0x00000032: 04 DW_LNS_set_file (1)
0x00000034: 00 DW_LNE_set_address (0x0000000000000000)
0x0000003f: 14 address += 0,  line += 2
            0x0000000000000000      3      0      1   0
0  is_stmt
0x00000040: 05 DW_LNS_set_column (3)
0x00000042: 0a DW_LNS_set_prologue_end
0x00000043: 83 address += 8,  line += 1
            0x0000000000000008      4      3      1   0
0  is_stmt prologue_end
0x00000044: 06 DW_LNS_negate_stmt
0x00000045: 0b DW_LNS_set_epilogue_begin
0x00000046: 4a address += 4,  line += 0
            0x000000000000000c      4      3      1   0
0  epilogue_begin
0x00000047: 02 DW_LNS_advance_pc (8)
```

‘foo’ in b.o

```
0x00000048: 04 DW_LNS_set_file (2)
0x0000004a: 00 DW_LNE_set_address (0x0000000000000008)
0x00000055: 14 address += 0,  line += 2
            0x0000000000000008      3      0      2   0
0  is_stmt
0x00000056: 05 DW_LNS_set_column (3)
0x00000058: 0a DW_LNS_set_prologue_end
0x00000059: 83 address += 8,  line += 1
            0x0000000000000010      4      3      2   0
0  is_stmt prologue_end
0x0000005a: 06 DW_LNS_negate_stmt
0x0000005b: 0b DW_LNS_set_epilogue_begin
0x0000005c: 4a address += 4,  line += 0
            0x0000000000000014      4      3      2   0
0  epilogue_begin
0x0000005d: 02 DW_LNS_advance_pc (8)
```

DW_LNE_set_address is an opcode that contains a relocatable address which is resolved by the linker

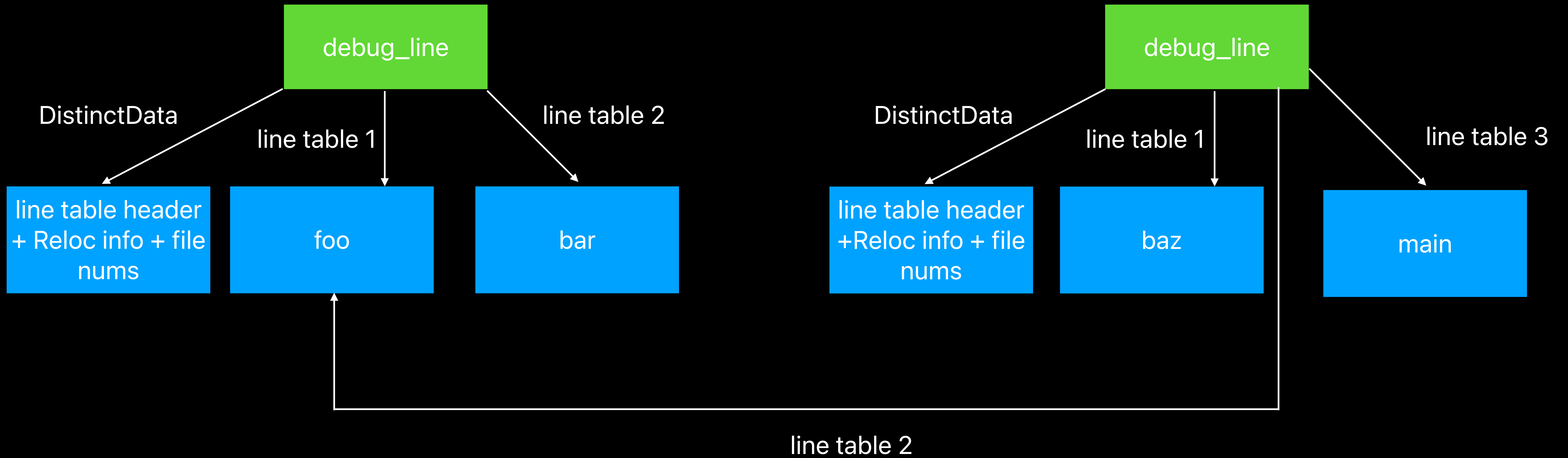
Storing __debug_line in the CAS efficiently

- Idea, split line table so that every function’s contribution is distinguishable
- Emit an ‘end_sequence’ opcode after every function, reset state machine
- Create one CASObject per function
- Line table header will not deduplicate, store in different *DistinctData* CASObject

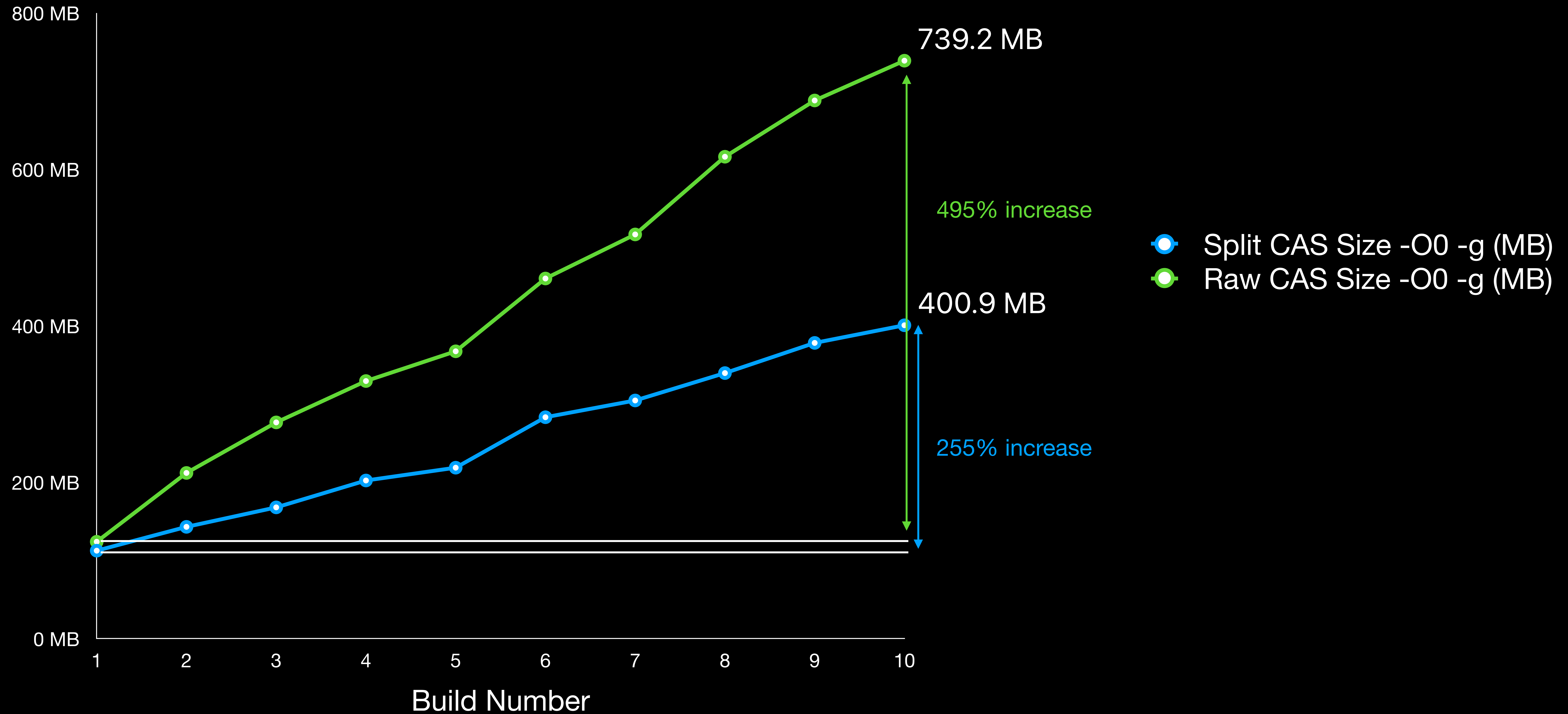
Storing __debug_line in the CAS efficiently contd...

- Store file numbers and relocation addends in *DistinctData*
- CAS references are ordered, we can guarantee that the line table will be rebuilt the same as it was in the object file

CAS representation of __debug_line section



Results of storing __debug_line in CAS



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='llvm', ninja

__debug_line section representation

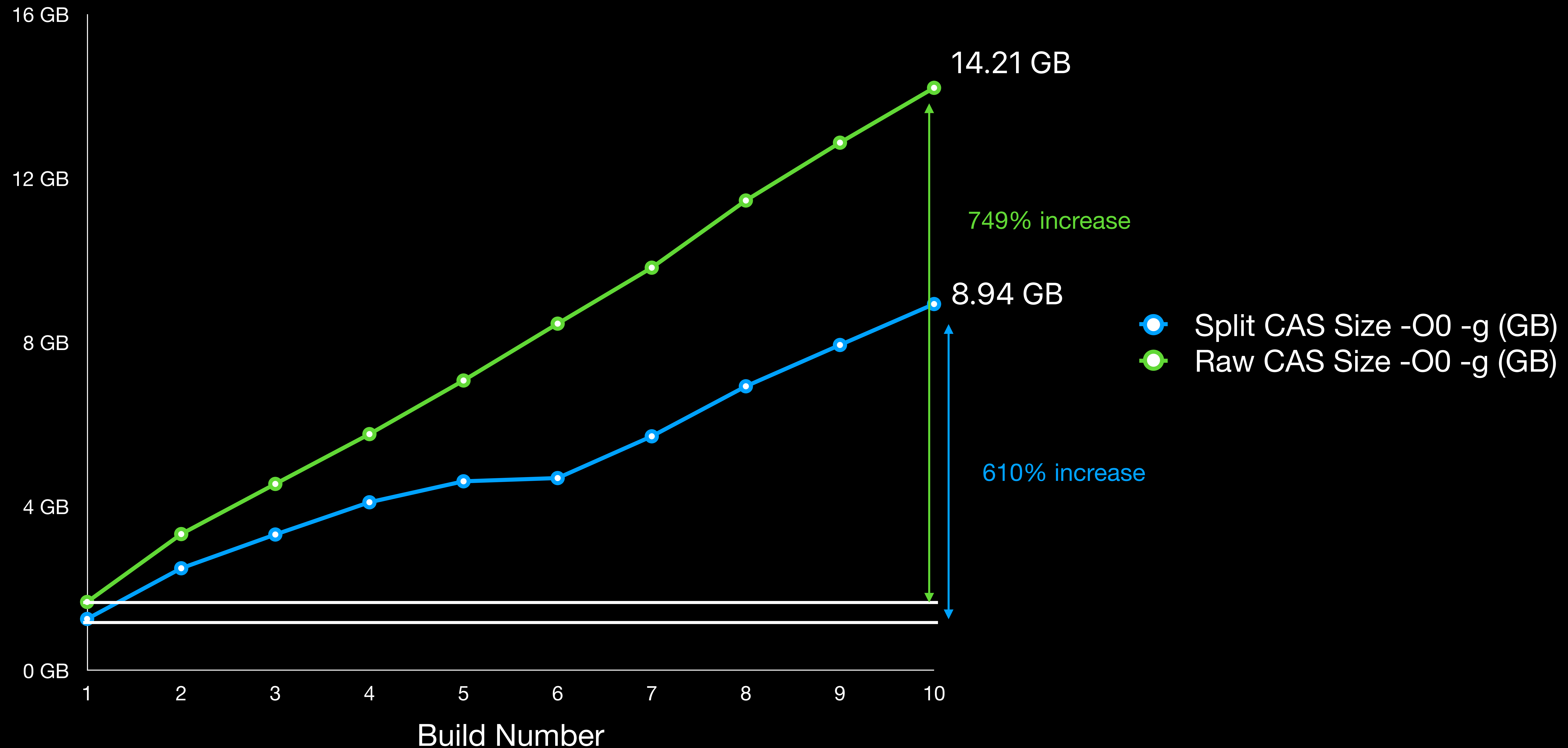
__debug_abbrev and __debug_info representation

- __debug_abbrev section contains the description of the metadata in the __debug_info section
- Both sections can be divided into DIEs (Debug Information Entries)
- Example: One type or function ≈ One DIE

__debug_abbrev and __debug_info representation

- Group DIEs into CASObjects via heuristic
- __debug_abbrev has abbrev codes that don’t deduplicate and aren’t stored
- __debug_info has specific metadata that doesn’t deduplicate and is stored in *DistinctData*

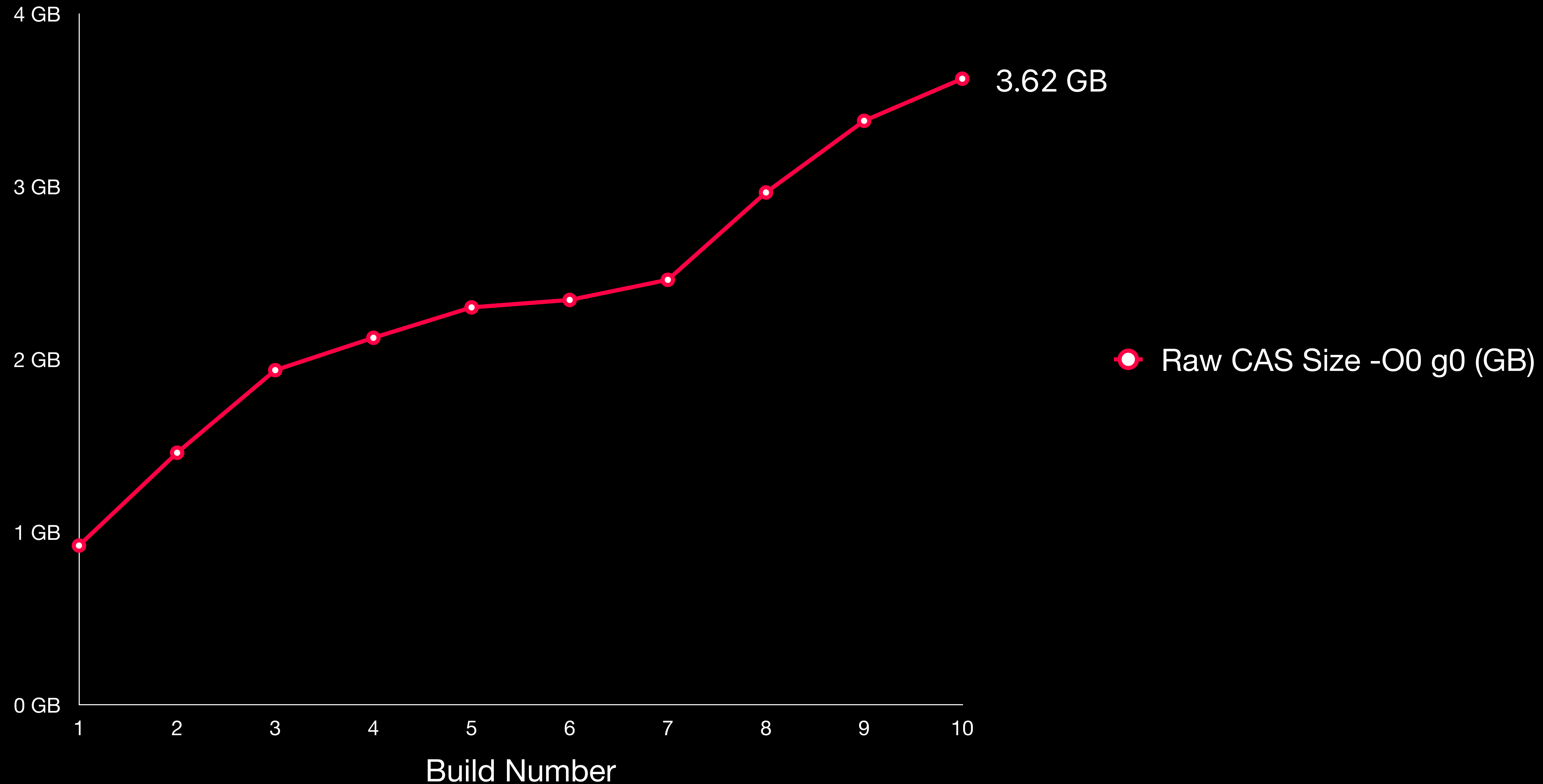
Results of storing __debug_abbrev and __debug_info in CAS



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='llvm', ninja

Putting it all together...

Results of total size of object file ingestion in CAS



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='llvm', ninja

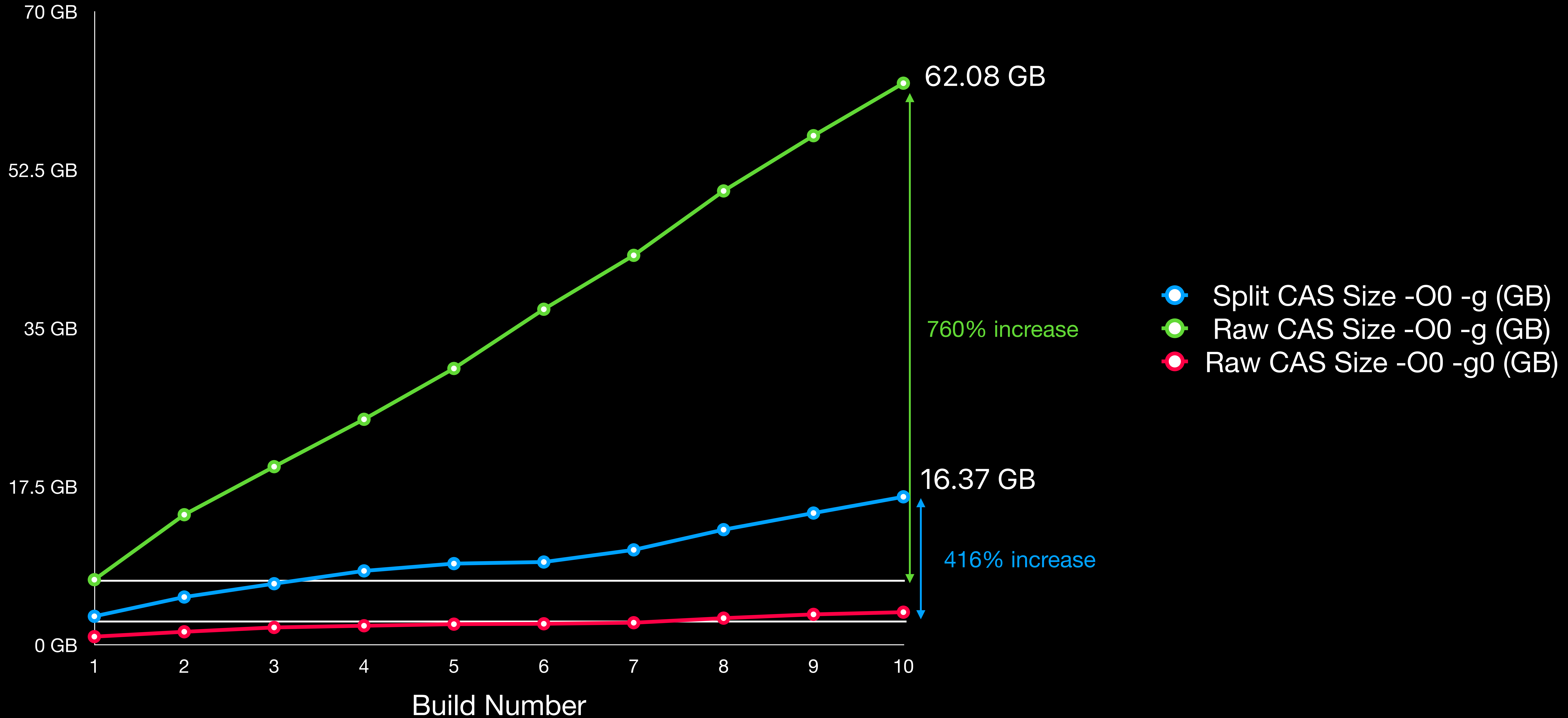
Results of total size of object file ingestion in CAS

llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='llvm', ninja

Results of total size of object file ingestion in CAS

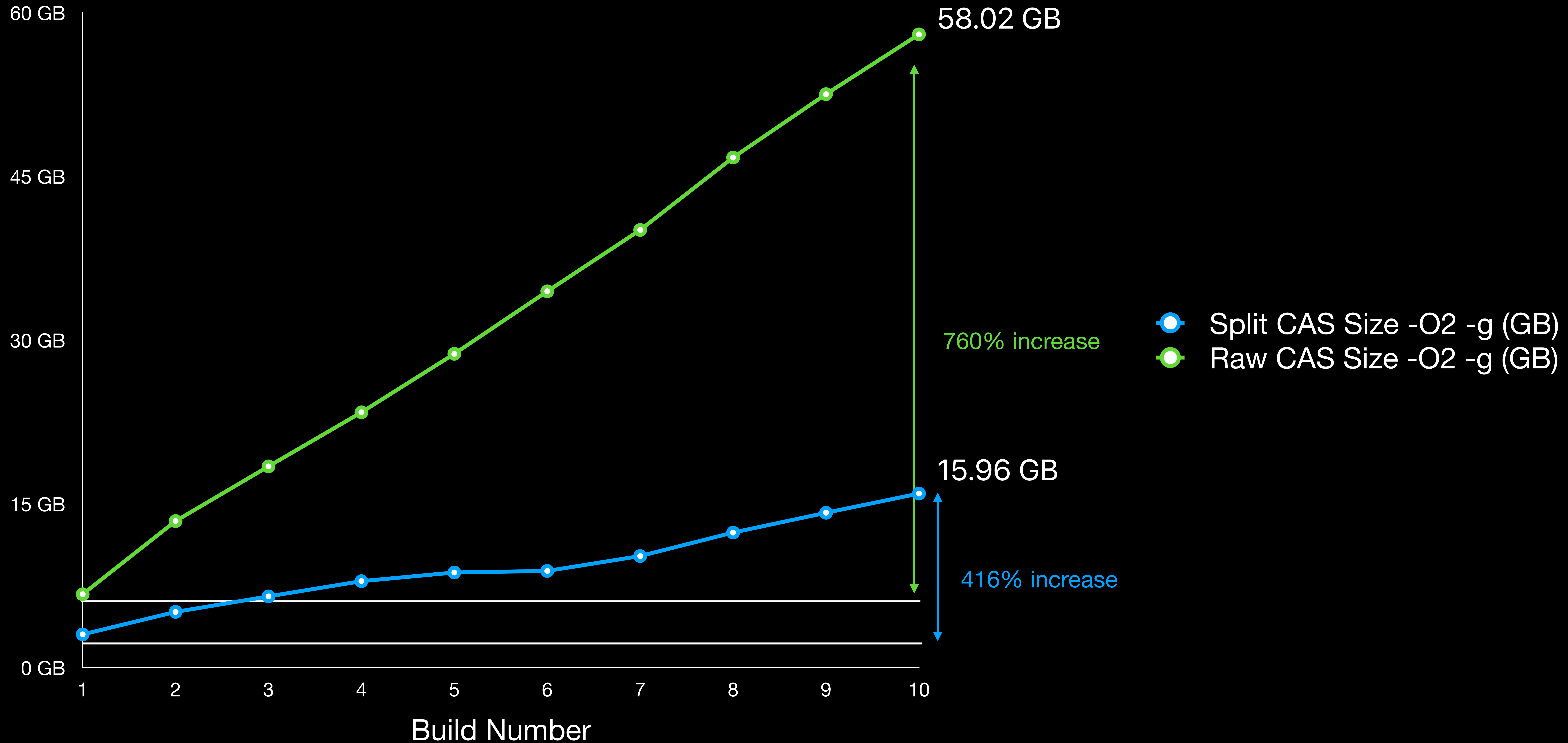
llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='llvm', ninja

Results of total size of object file ingestion in CAS



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='llvm', ninja

Results of total size of object file ingestion in CAS



llvm-project, -DCMAKE_ENABLE_LLVM_PROJECTS='llvm', ninja

Conclusions

- In this talk we showed how to expose redundancies in debug info to make it cach-able, efficiently
- Incremental builds benefit significantly from the CAS paradigm

Future work

- The __debug_info section can be further optimized
- We haven’t implemented any specific optimizations for other debug info sections such as __debug_loc
- We plan to contribute the CAS work into llvm.org
- RFC: https://discourse.llvm.org/t/rfc-add-an-llvm-cas-library-and-experiment-with-fine-grained-caching-for-builds/59864

Questions?