Using Content-Addressable Storage in Clang for Caching Computations and Eliminating Redundancy

Steven Wu, Ben Langmuir

LLVM Developers' Meeting 2022 | Apple, Inc. | November 9, 2022
Vision

Compilers are getting more and more complex
  • Using lots of computation, memory and storage

... But much of the work is redundant
  • Parsing the same sources, optimizing the same functions, etc.
Vision

Compilers are getting more and more complex

• Using lots of computation, memory and storage

... But much of the work is redundant

• Parsing the same sources, optimizing the same functions, etc.

Our solution to the problem is: **Content Addressable Storage**!

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

What is a CAS?
Clang Caching
CAS ObjectFile Storage
Potential and Future Work
CAS: Content Addressable Storage

Content stored is assigned a unique address based on its content

Widely used, including many build systems

A new concept to introduce to compilers
CAS Characteristics

Uniqueness
• Identical data stored into CAS will be assigned the same address

Immutable
• The content assigned to the address cannot be changed
CAS: Data Uniqueness

Uniqueness

- Identical data stored into CAS will be assigned the same address

Hello World!

file://tmp/hello.txt

file://home/hello.txt
CAS: Data Uniqueness

Uniqueness

- Identical data stored into CAS will be assigned the same address

Hello World! → llvmcas://3a079dbc
CAS: Immutable Data

Immutable

• The content assigned to an address cannot be changed

```
llvmcas://a51718ca
Hello World!
Hello Again!
```

```
llvmcas://3a079dbc
Hello World!
```

```
file://tmp/hello.txt
Hello World!
Hello Again!
```

```
file://tmp/hello.txt
Hello World!
```

```
file://tmp/hello.txt
Hello Again!
```
Proposed LLVM CAS Library

CAS APIs

• Thread Safe, easy to use for compiler integration
• Extensible for different CAS implementation (e.g. RemoteCAS)
• ObjectStore: Content Addressable Storage
• ActionCache: KeyValue storage to associate inputs and outputs

Built-inCAS

• A default CAS implementation within LLVM
• Good performance for production usage
CASObject Model

CASObject contains:
- Data blob
- [ Ref ]

Ref points to another CASObject

The address of CASObject is computed based on both Data and Refs
Build CASObject graphs from bottom up from leaf nodes

Properties of CASObject graphs

- Direct Acyclic Graph: no cycles
- Easily composable
- Infer equality from the address of the root node
CASSchema

CASSchema: a model used to serialize and deserialize high-level objects from CAS
CASSchema: a model used to serialize and deserialize high-level objects from CAS

Example FileSystemTree where file1 and file3 are identical
Clang Caching
Motivation

Compilers perform redundant work
  • Within individual builds
  • Across builds
Motivation

Compilers perform redundant work

- Within individual builds
- Across builds

Want to cache computations, but...

- Inputs and outputs not well-specified
- Mutable filesystem
Motivation

Build against persistent CAS

• Isolate pure computations
• Sound and efficient caching by design
Motivation

Build against persistent CAS

- Isolate pure computations
- Sound and efficient caching by design

Example: whole compilation caching
Clang Compilation Caching

Cache compilation of each translation unit

Like "ccache", but with compiler integration
  • Sound and efficient caching by design
  • Designed to work with modules

Implemented at
  • https://github.com/apple/llvm-project/tree/experimental/cas/main
Clang Compilation Caching: Quick Tour

```cpp
#include "test.h"

int main() {
    printHello();
    return 0;
}
```

```cpp
void printHello();
```

```bash
clang-cache clang++ -c test.cpp -Rcompile-job-cache
```

Enable Caching
Enable cache-related remarks
Clang Compilation Caching: Quick Tour

```cpp
#include "test.h"

int main() {
    printHello();
    return 0;
}
```

clang-cache clang++ -c test.cpp -Rcompile-job-cache
```
remark: compile job cache miss for 'llvmcas://983e09ad7717df57dbcc6906c977e11c769936f5a740aeb46f40a57743abb2ad' [-Rcompile-job-cache-miss]
```

clang-cache clang++ -c test.cpp -Rcompile-job-cache
```
remark: compile job cache hit for 'llvmcas://983e09ad7717df57dbcc6906c977e11c769936f5a740aeb46f40a57743abb2ad' => 'llvmcas://a6b70c60e0165e98295f6f2ee40a70e0233748c2bf09ca2d1de0a869d99cca7b' [-Rcompile-job-cache-hit]
```
How does it work?

1. clang-scan-deps discovers inputs; ingest into CAS
2. Produce a `-cc1` command that only accesses the CAS
3. Capture outputs in `VirtualOutputBackend`
4. Fearlessly cache results
clang-scan-deps discovers inputs

clang-scan-deps is a library to discover dependencies

• Finds file and module dependencies
• Much faster than preprocessing

For more information

• clang-scan-deps: Fast Dependency Scanning for Explicit Modules EuroLLVM 2019
• Implicitly discovered, explicitly built Clang modules EuroLLVM 2022
clang-scan-deps discovers inputs: Ingest into CAS

Ingest input files using CachingOnDiskFileSystem (VFS)
Simple input discovery with CachingOnDiskFileSystem

Track filesystem accesses

open main.cpp → stat /usr/include → open stdio.h → open_stdio.h → ...

Ingest contents into CAS

#include <stdio.h>
#include <_stdio.h>
// _stdio.h
Simple input discovery with CachingOnDiskFileSystem

Build CASFileSystem tree

```
#include <stdio.h>
#include <_stdio.h>
```

Simple input discovery with CachingOnDiskFileSystem

CASFileSystem identified by root CAS ID / llvmcas://303fb490

-fcas-fs llvmcas://303fb490
Fearlessly cache results

Cache key:
• –cc1 command
• CAS inputs
• Compiler version
Fearlessly cache results

```
clang-cache clang++ -c test.cpp -Rcompile-job-cache
```

**remark:** compile job cache hit for 'llvmcas://983e09ad7717df57dbcc6906c977e11c76f936f5a740aeb46f40a57743abb2ad' => 'llvmcas://a6b70c60e0165e98295f8f2ee40a0e0e23714c832809ca2f9e089099ca7b' [-Rcompile-job-cache-hit]

```
command-line:
  -cc1
  -fcas-path llvm.cas.builtin.v2[BLAKE3]
  -fcas-fs llvmcas://89bafd50702f574967b46b94ed4da43c7bdefb9af3d69b8087d34029e6113af
  -fcas-fs-working-directory /Users/blangmuir/src/cas/build
  -o \
  -emit-obj \n  -x c++ test.cpp \
  ...
```

**CAS configuration**

```
filesystem: llvmcas://89bafd50702f574967b246b94ed4da43c7bdefb9af3d69b8087d34029e6113af
  tree llvmcas://f0945... /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/System/Library/Frameworks/
  tree llvmcas://f0945... /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/c++/v1/
  tree llvmcas://f0945... /Users/blangmuir/src/cas/build/lib/clang/16.0.0/include/
file llvmcas://31a68... /Users/blangmuir/src/cas/build/test.cpp
file llvmcas://f1748... /Users/blangmuir/src/cas/build/test.h
```

**CAS filesystem contains all used files**

```
version:
  clang version 16.0.0 (git@github.com:apple/llvm-project.git 8f121b6910e7418a28ca9d926512950ca0b75b47e)
```
CachingOnDiskFileSystem has room for improvement

Conservatively models all filesystem accesses

Some dependencies too coarse-grained, causing high cache misses
  • Consider headermaps: contain a list of every header in target/project
  • Add a new header => 100% cache misses since the headermap is modified

Run header search twice - already done in clang-scan-deps
Include Tree

Introducing "Include Tree"

Save which file is included at each #include directive

Header search is only run in clang-scan-deps

Only files actually needed by compilation are included
  • No more search paths, headermaps, ivfsoverlay in cached -cc1 command
Include Tree: Example

```
include_tree: llvmcas://2cfc99...
test.cpp llvmcas://c53da...
1:1 <built-in> llvmcas://deaec...
2:1 ./H1.h llvmcas://ccb75...
   2:1 ./H2.h llvmcas://3e709...
   2:16 ./H3.h llvmcas://7ff63...
```
Include Tree: Example

include-tree: llvmcas://2cfc99...

- test.cpp  llvmcas://c53da...
- 1:1  <built-in>  llvmcas://deaec...
- 2:1  ./H1.h  llvmcas://ccb75...
  - 2:1  ./H2.h  llvmcas://3e709...
  - 2:1  ./H3.h  llvmcas://7ff63...
  - 2:16  ./H3.h  llvmcas://7ff63...
Building clang with caching

cmake -G Ninja \
  -DCMAKE_C_COMPILER_LAUNCHER:PATH=$(which clang-cache) \ 
  -DCMAKE_CXX_COMPILER_LAUNCHER:PATH=$(which clang-cache) \ 
  -DLLVM_ENABLE_PROJECTS="clang" \ 
  -DLLVM_TARGETS_TO_BUILD="X86;ARM;AArch64" \ 
  -DCMAKE_BUILD_TYPE:STRING=Release \ 
  -DLLVM_ENABLE_ASSERTIONS:BOOL=ON \ 
  -DCMAKE_CXX_FLAGS="-Rcompile-job-cache" \ 
  ../llvm-project/llvm
Building clang with caching

time ninja clang
5004.98s user 238.51s system 922% cpu 9:28.23 total

ninja clean

time ninja clang
66.26s user 26.42s system 526% cpu 17.615 total
Building clang with caching

```
Time (s)

Baseline

CAS

ninja clang (Release+Asserts)

556

568

First Build
```
Building clang with caching

### ninja clang (Release+Asserts)

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>CAS</th>
<th>scache</th>
<th>ccache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>556</td>
<td>568</td>
<td>592</td>
<td>615</td>
</tr>
</tbody>
</table>

**First Build**
Building clang with caching

ninja clang (Release+Asserts)

Baseline: 556
CAS: 568 (First Build 17.6, Fully Cached Rebuild 47.2)
sccache: 592
ccache: 615 (First Build 29.4)

First Build
Fully Cached Rebuild
Clang Caching: Summary

Cache compilation of each translation unit

Compiler integration enables:

• Efficient and sound caching
  - Fast dependency scanning with clang-scan-deps
  - Isolation from the mutable filesystem using CAS
  - Domain-specific cache data structures (e.g. Include Tree)
• Designed to support modules (work in progress)
CAS ObjectFile Storage
Motivation

Build artifacts are large!

- Expensive to store
- Can be a bottleneck for object cache

Lots of information is duplicated

- Across incremental builds
- Within a single build
Observation

Within a build, duplicated information can be found in:

- Data: C-Strings
- ODR functions: C++ template functions
- Debug Info: Type info
Observation

LotsOfFunctions.o

__TEXT
0x100: ab c5 e4 ec
       be b3 45 74
0x200: 96 c2 33 ec
       e3 34 73 b8
0x300: f5 8c 46 b6
       b3 9b 46 0e

__DATA
0x400: 00 00 00 00
       00 00 00 00

__LINKEDIT

funcA = 0x100
funcB = 0x200
funcC = 0x300
var   = 0x400

UUID: 95EA-E8EF-D3DB-4CDA
Observation

<table>
<thead>
<tr>
<th>TEXT</th>
<th>DATA</th>
<th>LINKEDIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100:</td>
<td>ab c5 e4 ec</td>
<td></td>
</tr>
<tr>
<td></td>
<td>be b3 45 74</td>
<td></td>
</tr>
<tr>
<td>0x200:</td>
<td>96 c2 33 ec</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e3 34 73 b8</td>
<td></td>
</tr>
<tr>
<td>0x300:</td>
<td>f5 8c 46 b6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b3 9b 46 0e</td>
<td></td>
</tr>
<tr>
<td>0x400:</td>
<td>00 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEXT</th>
<th>DATA</th>
<th>LINKEDIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100:</td>
<td>8d dd ce 23</td>
<td></td>
</tr>
<tr>
<td></td>
<td>57 2b 8f 3a</td>
<td></td>
</tr>
<tr>
<td>0x220:</td>
<td>96 c2 33 ec</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e3 34 73 b8</td>
<td></td>
</tr>
<tr>
<td>0x320:</td>
<td>f5 8c 46 b6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b3 9b 46 0e</td>
<td></td>
</tr>
<tr>
<td>0x420:</td>
<td>00 00 00 00</td>
<td></td>
</tr>
</tbody>
</table>

Change `funcA` and rebuild. The size of `funcA` changes.

LotsOfFunctions.o

UUID: 95EA-E8EF-D3DB-4CDA

funcA = 0x100
funcB = 0x200
funcC = 0x300
var = 0x400

LotsOfFunctions.o

UUID: 16F6-47E1-83B5-268F

funcA = 0x100
funcB = 0x220
funcC = 0x320
var = 0x420
Observation

Change funcA and rebuild

The size of funcA changes

Duplicated content
MCCASObjectFormat

New CASObject representation for MachO object files

- MachOCASObjectWriter: new object writer directly writes to CAS
- Utilize all Machine Code information (e.g. MCFragments)
- Break up object file into blocks that can be reused
- Can be serialized back into MachO Object that is identical to the original
Object Storage Benchmark

Object storage cost benchmark

- building llvm + clang (-O3 -g0)
- 1 commit per day across 10 days
- only store unchanged native macho object files once

Ilvm-project from ilvmorg-15-init, 1200 commits of changes in total for 10 days
Object Storage Benchmark

CASObjectFormat

- space saving across the board
Object Storage Benchmark

CASObjectFormat
- space saving across the board
- much slower growth in storage cost
- significant long term benefit!
Future Opportunities
Fine-grained Caching

Finer-grained caching will expose more redundancies

- clang token cache (prototype available)
- Long term: refactor compilers for fine-grained request-driven computations
Other Opportunities

CAS-friendly Debug information
  • Make line tables *invariant to code motion*
  • Design *modular type information*

Add caching to more tools
  • TableGen
  • Linker

CAS-friendly build artifacts for less storage
Conclusion


Round Table for Content Addressable Storage (Today Nov. 9 4:30pm)

All examples and prototypes can be found at: [https://github.com/apple/llvm-project/tree/experimental/cas/main](https://github.com/apple/llvm-project/tree/experimental/cas/main)

Pull Requests:

- VirtualOutputBackend: [https://reviews.llvm.org/D133504](https://reviews.llvm.org/D133504)
- CAS: [https://reviews.llvm.org/D133716](https://reviews.llvm.org/D133716)
Q&A