cymbl: To -jInfinity & Beyond

William S. Moses

Kevin Kwok

wmoses@mit.edu
June 3, 2021
Compilation Bottlenecks

- As software proliferates in all parts of life, the amount of code in the world has grown exponentially
  
  - As of the 2015, Google alone had more than 9 million source code files (>2 billion LOC)[1]

- Compiling code is a bottleneck for development, testing, and publication of software

- Most compilation tasks are highly parallel (many individual files) but practically limited by the number of cores on your machine

- Most builds unnecessarily repeat existing work
  
  - Everyone building the same existing package

  - Development is incremental — typically few files are modified in a given patch

Ideal Remote Compilation

- Drop in replacement without rewriting the codebase (e.g. “it just works”)
- Infinite parallelism by offloading compilation to remote machines
- Cache equivalent compilation tasks rather than recomputing
## Existing Remote Compilation Tools

<table>
<thead>
<tr>
<th></th>
<th>Compatibility</th>
<th>Parallelism</th>
<th>Caching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bazel</td>
<td>![x] Must use build system</td>
<td>![x] Requires user cluster*</td>
<td>![?] Per-codebase caching</td>
</tr>
<tr>
<td>DistCC</td>
<td>![?] Models compile command</td>
<td>![x] Requires user cluster</td>
<td>![x] Limited or no caching</td>
</tr>
<tr>
<td>Goma</td>
<td>![?] Models compile command</td>
<td>![x] Requires user cluster</td>
<td>![?] Per-codebase caching</td>
</tr>
<tr>
<td>gg</td>
<td>![x] Models all build commands</td>
<td>![✓] On-demand compute</td>
<td>![?] Per-invocation caching</td>
</tr>
</tbody>
</table>
• Idea: Integrate remote execution into the compiler itself
  
  • Usable in any existing build system & “model” will always be perfect
  
  • Much more effective cache as the compiler has all the relevant information to normalize builds
  
  • Merging remote execution and the compiler results in much more efficient execution, reducing both latency and total build time
  
  • Leverages cloud functions to provide infinite parallelism without requiring the user to maintain infrastructure and without gg’s requirement to model all commands
  
  • Reduces 21-hour Chrome build down to a few minutes
Drop-in Replacement

• After downloading Cymbl, change the default compiler to use Cymbl instead of default

• When building, set desired parallelism and let it run!
Cymbl Design

- `a.out`: output binary
- `make -j∞`: original arguments
- `clang lld clang lld`: compile & link jobs
- `file paths`: normalized arguments
- `source code`: compilation cache
- `cymbld`: content-addressed storage
- `gatekeeper`: unique

Diagram shows the process flow from source code to output binary, involving preprocessing, compilation, and linking, with integration of cloud services and a gatekeeper mechanism.
Cymbl Daemon (cymbld)

- Many compilation tasks share the same dependencies, so to avoid duplicate uploads, file uploading is handled by a shared daemon process (cymbld).
- Clang and lld processes send dependency file paths to cymbld through IPC.
- Cymbld hashes, dedups, and batches before querying the server for cache misses.
- Cymbld uploads files and notifies clang/lld when dependencies have been uploaded and provides credentials for invoking lambdas.
Caching

- Ensure Deterministic Builds
  - Rewrite all “time of build” macros to be a fixed constant for determinism
  - All files used are explicitly passed by hash
  - Normalize tasks for better cache hits
  - When executing a task, first check it exists inside the cache and if so immediately return the result
Task Normalization by Preprocessing Source

- Identify required arguments & inputs (purple)
- Remove unused defines (blue)
- Normalize include paths (green)
- Provide map of exactly what files are used with their corresponding hash in content-addressable storage (red)

```bash
```

```
args: ['-cc1', '-triple', 'arm64-apple-ios10.0.0', '-o', 'o0', '-x', 'objective-c', 'PropertyAnimatorViewController.m', '-internal-isystem', '/fakeroot-s'],
inputs: {
'/fakeroot-s/UIKit.framework/Headers/UIKit.h': 'wFr1pQYtbT2X041sYCrrKR3FfJUGhvvy9Xw8sYycGq4=',
'PropertyAnimatorViewController.h': 'fke8yluU1f/H55VrnLK3xOzubvr/3h24VjBSW8aZc+Q=',
'PropertyAnimatorViewController.m': 'uqncMT16aeuzl1Fr1wYh4vH0WtP1nB+Nz8Vc&2nuc=',
```
Cross-Platform & Cross-Architecture

- When client binaries are run it identifies the desired target platform and architecture which are later passed to the lambda compilation task.

- Every Raspberry Pi is secretly a thousand-core compiling supercomputer!

- Compile for ARM iOS/macOS on x86 Linux cluster (or other)
Performance Optimizations

Three primary components of Cymbl compilation time:

1. File Transfer (upload inputs / download results)

2. Communication Latency

3. Remote Task Execution (clang/lld jobs)
   
   * Time = Money and Shared among everyone
   
   * Full link time optimization (libc, libc++, DNS resolver, boringssl, curl, libclang, …)

   * Statically link everything

   * Bonus: Binaries are very portable (no dependencies)
File Transfer

- Biggest (initial) bottleneck for clients is transferring inputs/results

- Daemon serves as a single point to optimize transfers (rather than per process)
  - Staged existence caching (with invalidation)
    - Local concurrent map (fastest); Batched remote check (mid speed); (potential) re-upload (slowest)
  - Limit the number of concurrent uploads/connections (per network performance)
  - Assuming cluster network is much faster than one’s ISP, batch upload many files together for later split by remote upload processing lambda

- Storage with weaker properties (non-atomic) is vastly faster than that with stronger properties
  - Design invalidation-safe idempotent upload process
  - Retry compilation task if file has not been propagated to storage where needed
Latency

- Reducing the latency of file existence checks and already-cached tasks is key to performance of large workloads

\[
\begin{align*}
\text{~300ms} & : \text{“Standard” lambda function} \\
\text{~50ms} & : \text{FullLTO + statically linked everything} \\
& \quad \text{Latency reduced both by time optimization and reduction in file size} \\
\text{~5ms} & : \text{Handwritten AWS API, persistent connection, fine tuning flags}
\end{align*}
\]
## Evaluation

<table>
<thead>
<tr>
<th></th>
<th>1-Core</th>
<th>96-Core</th>
<th>Cymbl</th>
<th>Cached Cymbl</th>
<th>gg*</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FFmpeg</strong></td>
<td>9.43</td>
<td>0.48</td>
<td>0.53</td>
<td>0.04</td>
<td>0.73*</td>
</tr>
<tr>
<td><strong>InkScape</strong></td>
<td>39.96</td>
<td>1.06</td>
<td>1.12</td>
<td>0.25</td>
<td>1.45*</td>
</tr>
<tr>
<td><strong>Clang</strong></td>
<td>183.55</td>
<td>4.32</td>
<td>2.42</td>
<td>0.36</td>
<td></td>
</tr>
<tr>
<td><strong>Chrome</strong></td>
<td>1302.65</td>
<td>25.71</td>
<td>6.99</td>
<td>4.42</td>
<td>18.92*</td>
</tr>
</tbody>
</table>

*gg results taken from paper, due to inability to reproduce results*
Relative Speed-up (vs Single Core)

*gg results taken from paper, due to inability to reproduce results
Costs & Other Analysis

- Costs computed for initial ram budget (3GB)
- 50k file compilation task
- 96-core cost $4.08/hour (need the hour)
  - ~2x more expensive uncached [3.5x speed]
  - ~22x cheaper when cached [and 300x speed]
- 47 hours of compute for uncached; 1 hour of compute for cached

<table>
<thead>
<tr>
<th></th>
<th>Chrome Uncached</th>
<th>Chrome Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang</td>
<td>$8.478</td>
<td>$0.184</td>
</tr>
<tr>
<td>lld</td>
<td>$0.047</td>
<td>$0.002</td>
</tr>
<tr>
<td>exists</td>
<td>$0.014</td>
<td>$0.000</td>
</tr>
<tr>
<td>upload</td>
<td>$0.026</td>
<td>$0.000</td>
</tr>
<tr>
<td>Total</td>
<td>$8.565</td>
<td>$0.186</td>
</tr>
</tbody>
</table>
## Optimized Costs

- As >99.996% tasks use <1.5GB (can half the cost)

- 50k file compilation task

- 96-core cost $4.08/hour (need the hour)

- ~On par when uncached [3.5x speed]

- ~43x cheaper when cached [and 300x speed]

- 47 hours of compute for uncached; 1 hour of compute for cached

<table>
<thead>
<tr>
<th></th>
<th>Chrome Uncached</th>
<th>Chrome Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang</td>
<td>$4.240</td>
<td>$0.092</td>
</tr>
<tr>
<td>lld</td>
<td>$0.047</td>
<td>$0.002</td>
</tr>
<tr>
<td>exists</td>
<td>$0.014</td>
<td>$0.000</td>
</tr>
<tr>
<td>upload</td>
<td>$0.026</td>
<td>$0.000</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>$4.326</strong></td>
<td><strong>$0.094</strong></td>
</tr>
</tbody>
</table>
Security

- All accesses to any cloud data are mediated by a Gatekeeper

- Gatekeeper only grants downloads of results of tasks submitted by that user
  
  - Cannot download another’s source

  
  - Cannot download another’s artifacts without a compilation job that would result in that artifact anyways

- Remaining attack vector: brute force timing attack of existence queries for source code / compilation jobs to attempt to identify another user’s source:

  
  - Intractable space size (all programs) and only can work once (since all brute forced jobs will be subsequently cached)
## Potential Additional Security Extensions

### Increasingly Paranoid Threat Model

<table>
<thead>
<tr>
<th>No Artifact Timing Attacks</th>
<th>No Input Timing Attacks</th>
<th>Distrust service provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Solution: Per user / company cache, or disable compilation cache</td>
<td>• Solution: Per user / company content-addressable storage</td>
<td>• Solution: User / company hosted task executors</td>
</tr>
<tr>
<td>• Cost: Reduction or loss of caching speedups</td>
<td>• Cost: Reduction of file-upload speedups AND costs to the left</td>
<td>• Cost: Maximum parallelism is limited to the size of the cluster, cost of maintaining a cluster, AND costs to the left</td>
</tr>
</tbody>
</table>
Status & Limitations

• Built on top of LLVM version 11

• Tool can (and has been) rebased across LLVM versions

• LLD only supports ELF not MACH targets (cymbl mach target works but LLVM proper doesn’t handle frameworks)

• Does not yet support caching with modules (falling back to caching with headers)

• Use as compile-tool and CI for MIT projects

• Accepting beta users for SAAS
Future Work

• Global Scale Compilation

• Super-optimization

• Profile-guided optimization database

• Language Extension (Swift, Rust, Go)

• Fine-Granularity Caching
Conclusions

• Raspberry Pi + Cymbl Cloud = Compiling Supercomputer!

• Compiler-level integration enables significantly better caching and compatibility

• State-of-the-art performance without the cost of a cluster

• Sign up for our beta! https://cymbl.dev/

• William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019323.
Questions?
Usage

• Same compiler binaries can be used for either local or remote builds

• Environmental variable enables or disables (CYMBL=On by default)
Existing Techniques

• Compatibility
  
  • Build-System Based (Bazel)
    
    • Requires rewriting all code to use the given build system, which handles remote task execution
  
  • Substitution-Based (Goma, DistCC, IceCC, gg)
    
    • Create fake “cc” compiler scripts to intercept tasks and execute remotely
    
    • gg builds a static graph of all computations ahead of time (potentially faster) at the cost of requiring all commands in the build process to be perfectly modeled
    
    • Requires maintaining an accurate model of all potential flags / behaviors for all tools, quickly becoming out of date and unlikely to align with a given system
  
  • Excluding gg, all tools require a user-maintained cluster, limiting parallelism and increasing cost
  
  • Caches at best recognize files in the same codebase being compiled in the same way
Potential Additional Security Extensions

- Per user / company cache, or disable entirely (request no cache)
  - Pro: Eliminate any compilation-job cache timing attacks
  - Con: Reduction or loss of caching speedups

- Per user / company content-addressable storage
  - Pro: Eliminate any input file cache timing attacks
  - Con: Above and reduction of file-upload speedups

- User / company-hosted job executors
  - Pro: No need to trust service provider (e.g. AWS)
  - Con: Above and maximum parallelism is limited to size of cluster which must be always on