cymbl: To -jInfinity & Beyond

William S. Moses[1] (wmoses@mit.edu), Kevin Kwok[1] (kkwok@mit.edu), Lizhou Sha[12] (lsha@wisc.edu)

[1]MIT SIPB [2]University of Wisconsin–Madison

Introduction

Recent years have seen the advent of serverless computing, where thousands of containers can be launched instantaneously, and paid in millisecond increments. For embarassingly parallel burst tasks, such as compiling and linking software, these platforms can be used for massive speed-ups, while costing much less than a dedicated server.

Existing tools often require maintaining prohibitively expensive clusters, while existing serverless approaches such as gg have limited build system compatibility. By integrating into the Clang compiler itself, Cymbl provides four key advantages over existing tools: infinite parallelism without user-supplied infrastructure, drop-in compatibility with existing build-systems, deterministic builds, and a fine-grained compilation cache across users and codebases. Cymbl also has support for Clang's tools and compiler plugins such as LLVM-based automatic differentiator (see the Enzyme poster).

	Cymbl	gg [1]	Goma	DistCC [2]	Bazel
Any Build System	✓	?	✓	✓	✗
No Modeling	✓	✗	✗	✗	?
Infinite Parallelism	✓	✓	✗	✗	?
Determinism	✓	✗	✗	✗	✓
Intra-Codebase Cache	✓	✓	✓	✗	✓
Inter-User Cache	✓	✗	✓	✗	✓
Cross-Codebase Cache	✓	✗	✗	✗	✗
Plugin Support	✓	✗	✗	✗	✗

Normalization

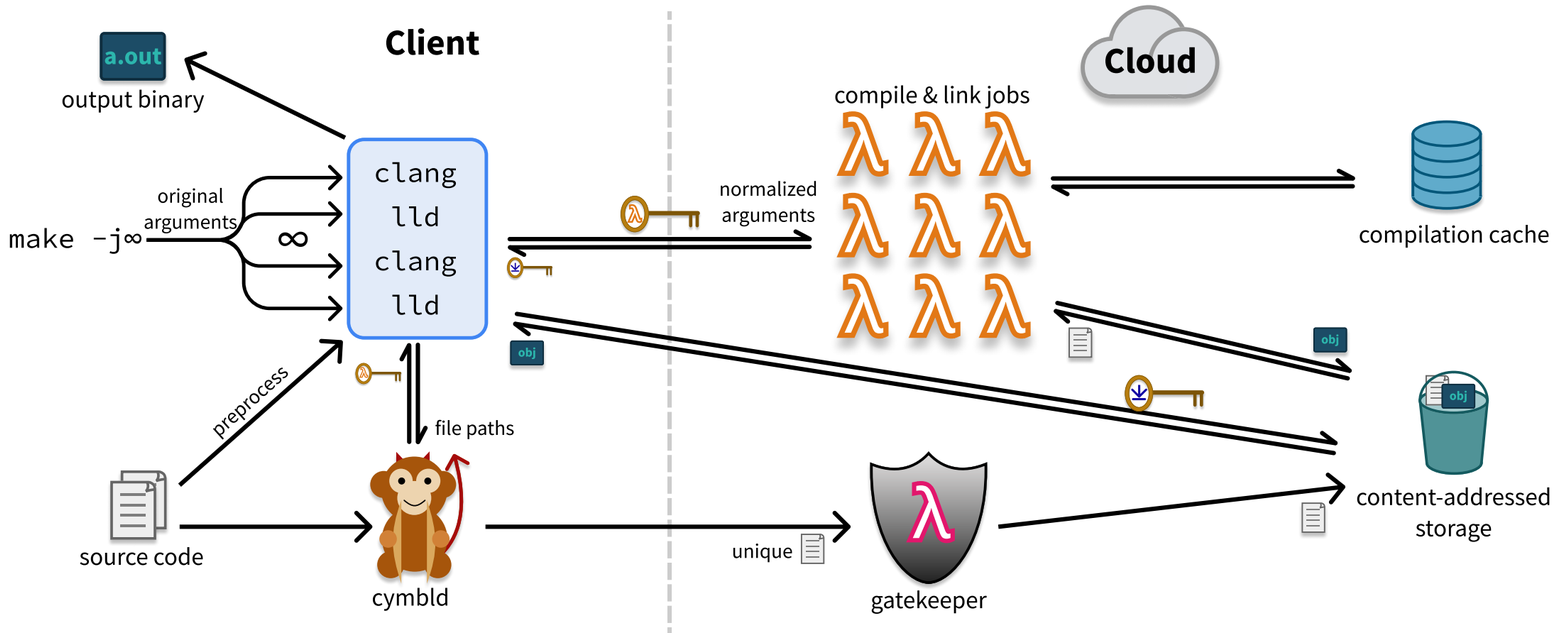
Deterministic builds are both desirable for debugging and allows Cymbl to legally cache compilations.

- **Content-Addressable Hash**: Files are represented to Cymbl by the hash of their contents.
- **Deterministic Macros**: We replace nondeterminstic macros (e.g. time/date) with fixed values.

To maximize cache hits, we normalize the input as follows:

- **Path Normalization**: We derive all files that are used by the compilation job and create an equivalent compilation task where all include paths are inside "/fakeroot". This allows our cache to work invariant of build location, include directory, or machine.
- **Macro Removal**: For ease, it is common for build systems to define macros which would otherwise break caching. As we run the preprocessor, we take a note of what macros are used, removing any extraneous ones. Note: Sometimes these are added explicitly to prevent caching if the caching mechanism isn't 100% correct.
- **Argument Normalization** Being the compiler itself, we have a true understanding of how compiler arguments are used and can normalize input arguments.

Workflow



Evaluation

	1-Core	96-Core	cymbl	cached	gg
FFmpeg	9.43	0.48	0.53	0.04	0.73*
Inkscape	39.96	1.06	1.12	0.25	1.45*
Clang	183.55	4.32	2.42	0.36	
Chrome	1302.65	25.71	6.99	4.42	18.92*

Table 1. Geomean build time (in minutes) of various codebases when compiled with different tools. We also include the gg build times of codebases where available as the current state of the art. We could not reproduce their results and include the results from the gg paper.

We evaluate the performance of Cymbl by benchmarking compile times of several codebases. We evaluate 1 local core, 96 local cores, and 8000 simultaneous Cymbl tasks. We also evaluate the performance of Cymbl when the compilation has already been cached. Times are the geomean of three runs taken from an Amazon c5.metal instance.

All codebases greatly benefit from increased parallelism both when adding more local cores and when using Cymbl in the cloud. Smaller codebases like FFmpeg and Inkscape maxed out the available build parallelism and didn't see a speedup between 96-cores and an uncached Cymbl. Notably, there wasn't much additional overhead from using Cymbl. Using Cymbl cached on these codebases was significantly faster. Larger codebases like Clang and Chrome saw additional and significant speedup when using Cymbl above even a 96-core build. Caching added additional speedboosts. Notably, the entire cached build time of Clang/LLVM was running tablegen and the Chrome build was often bottlenecked by network and disk.

We also include performance numbers of gg as the current state of the art. In spite of several attempts to patch gg, it was incompatible with the build flags needed to build the current version of these codebases. We thus include the numbers from the gg paper (also an Amazon machine). While we found roughly similar performance on 1 and 48-core builds, this isn't a true apples-to-apples comparison. That said, it appears gg had a much more significant overhead than Cymbl. On the Chrome build, Cymbl also appears to be significantly more performant.

To try Cymbl out, please email us and visit `https://cymbl.dev/`.

Acknowledgments

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

References

[1] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, 2019.

[2] Jes Hall. Distributed computing with distcc. *Linux Journal*, 2007(163):4, 2007.