PlayStation

Distributed ThinLTO with Icecream

Konstantin Belochapka and Katya Romanova

LLVM Developers' Meeting, November 2021

Sony Interactive Entertainment

Why aren't many people using Distributed ThinLTO?

Distributed ThinLTO is supported upstream, but unfortunately, it's still not commonly used other than in Google.

Distributed ThinLTO is quite complicated to integrate with the majority of build systems, mainly because the build-rule dependencies that a distributed build system needs to construct the build graph are not known in advance. The dependencies become available midway through the Distributed ThinLTO build-process, after the ThinLink phase completes, and we know the list of import-files.

The only distributed build system we know of that is aware of the Distributed ThinLTO build-flow is Bazel.

- Bazel is an open-source project and is an excellent choice for projects where Distributed ThinLTO is desired to be used.
- Unfortunately, configuration and deployment of Bazel takes a lot of effort and time.

Bazel
(Fast, Correct) - Choose two

Enabling Distributed ThinLTO for existing build projects is hard

Not everyone is ready to rewrite their build projects for Bazel from scratch to take advantage of Distributed ThinLTO, but it might even be harder to enable distributed ThinLTO for an existing software build-project written for makefiles, ninja, autotool or cmake.

To use Distributed ThinLTO in an existing makefile project, it will be required to:

1. For every bitcode file generate a list of files that ThinLTO needs to import from (after the ThinLink stage of ThinLTO finishes).
2. Analyze these dependencies.
3. Use these dependencies to drive remote distributed execution of ThinLTO's CodeGen processes.

This is not a trivial task even for a very experienced build master.

Due to the dynamic nature of the dependencies, these steps have to be redone when a code-change is made in the project.

Integrated Distributed ThinLTO approach

We came up with the Integrated Distributed ThinLTO approach where all the tasks that I just described are done by the linker. If someone has a project that already uses ThinLTO and wants to enable distributed ThinLTO, only a couple of small things are needed:

- To deploy a distributed build system (for example, Icecream).
- To add “--thinlto-distribute” option to the linker command line.

Dynamic dependency problem

Let’s assume we have a Makefile rule for performing the ThinLTO link-step that looks like that:

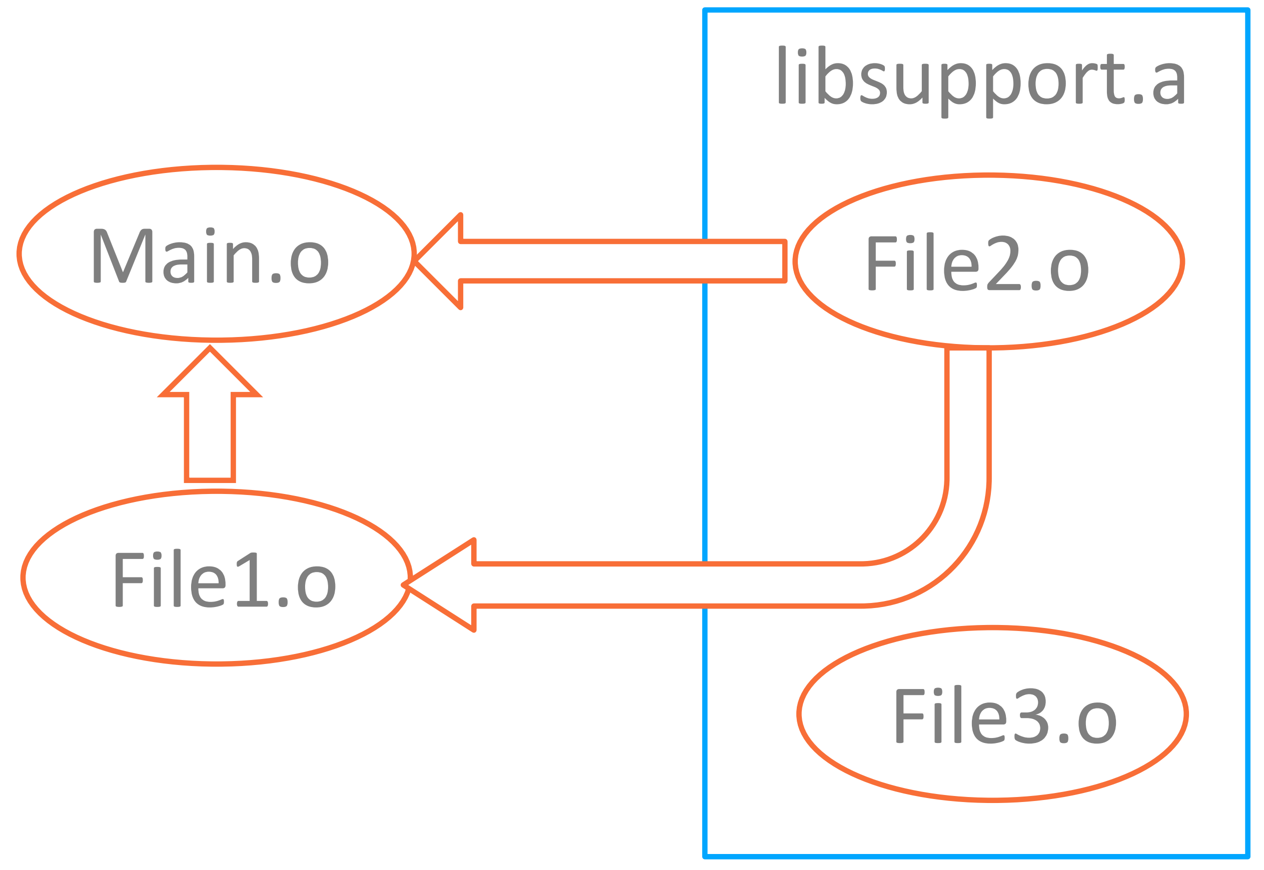
```
program.elf: main.o file1.o libsupport.a
        clang main.o file1.o -flto=thin -o program.elf -lsupport
```

- *main.o, file1.o are bitcode files*
- *libsupport.a is an archive containing 2 bitcode files: file2.o and file3.o*
- *main.o imports from file1.o, and file2.o (in libsupport.a)*
- *file1.o imports from file2.o*
- *file2.o has no dependencies*
- *file3.o is not referenced*

• It’s not obvious that because of cross-importing we need to send file2.o on the remote executor node for doing distributive code generation for main.o and file1.o.

• It’s not obvious that we need to do code generation for file2.o.

We will only find out these dependencies after the ThinLink phase of the ThinLTO process finishes.



Dynamic dependency problem solution

We implemented functionality within the LLD linker that generates an additional makefile containing the build rule with dynamically calculated dependencies when they become available after the completion of the ThinLink phase of ThinLTO.

Here is the example of an additional makefile **distr.makefile** that the linker generates for the rule on the previous slide:

```
DIST_CC := <path to a tool that can distribute ThinLTO codegen job>

main.native.o : main.thinlto.bc main.o file1.o file2.o
    $(DIST_CC) clang -thinlto-index=main.thinlto.bc main.o file1.o file2.o -o main.native.o
file1.native.o : file1.thinlto.bc file1.o file2.o
     $(DIST_CC) clang -thinlto-index=file1.thinlto.bc file1.o file2.o -o file1.native.o
file2.native.o : file2.thinlto.bc file2.o
    $(DIST_CC) clang -thinlto-index=file2.thinlto.bc file2.o -o file2.native.o
program.elf: main.native.o file1.native.o file2.native.o
    $(LD) main.native.o file1.native.o file2.native.o -o program.elf
```

Dynamic dependency problem solution

Now, from within the LLD linker, we simply invoke the make utility with the makefile that we just generated.

$(MAKE) -j<N> -f distr.makefile

distr.makefile

```
main.native.o : main.thinlto.bc main.o file1.o file2.o
    $(DIST_CC) clang -thinlto-index=main.thinlto.bc main.o file1.o file2.o -o main.native.o
file1.native.o : file1.thinlto.bc file1.o file2.o
     $(DIST_CC) clang -thinlto-index=file1.thinlto.bc file1.o file2.o -o file1.native.o
file2.native.o : file2.thinlto.bc file2.o
    $(DIST_CC) clang -thinlto-index=file2.thinlto.bc file2.o -o file2.native.o
program.elf: main.native.o file1.native.o file2.native.o
    $(LD) main.native.o file1.native.o file2.native.o -o program.elf
```

Dynamic dependency problem solution

With this approach the only modification that is required for the user to trigger Distributed ThinLTO is to add one additional option on the linker command line (**--thinlto-distribute**) and optionally specify which distribution system to use (**--distribute-cc=<path to a tool that can distribute ThinLTO codegen job>)**.

Of course, what's happening 'under the hood' is a little more complicated. The linker does the following tasks:

- analyzes dependencies.
- generates additional makefiles or build scripts.
- invokes the make utility that will launch the distributed build system for performing CodeGen.
- converts solid archives into thin archives and treats archive members as regular individual bitcode files.

Distributed ThinLTO and Icecream integration

Initially, we have implemented Integrated Distributed ThinLTO on Windows and hooked it up with Sony's proprietary build system SN-DBS. We wanted to submit our work for the linker and compiler upstream, but since SN-DBS can be used only by Sony's customers, we thought it might be difficult to justify this patch.

We decided to port the Integrated Distributed ThinLTO functionality on Linux and use the open-source distributed build system, Icecream (IceCC) to make our work useful for the LLVM community.

DISTRIBUTED THINLTO **+**

Distributed ThinLTO and Icecream integration

Some changes had to be done in Icecream side to allow us to hook it up with the distributed ThinLTO. Luckily, Icecream is an Open Source project.

Here is an abbreviated list of things that we had to do in Icecream to teach it to do code generation:

- Support –thinlto-index-only and -flto option for the IceCC compiler wrapper.
- Add LLVM IR bitcode files to the list of supported input files format.
- Support transferring a set of input files (instead of one file which was a limitation) both on Icecream client (IceCC) and Icecream server (iceccd deamon) sides.
- Implement an extension to the Icecream server, so that it could do CodeGen in multiple remote execution environments in parallel (to avoid file name collisions).

DISTRIBUTED
THINLTO

+

Performance

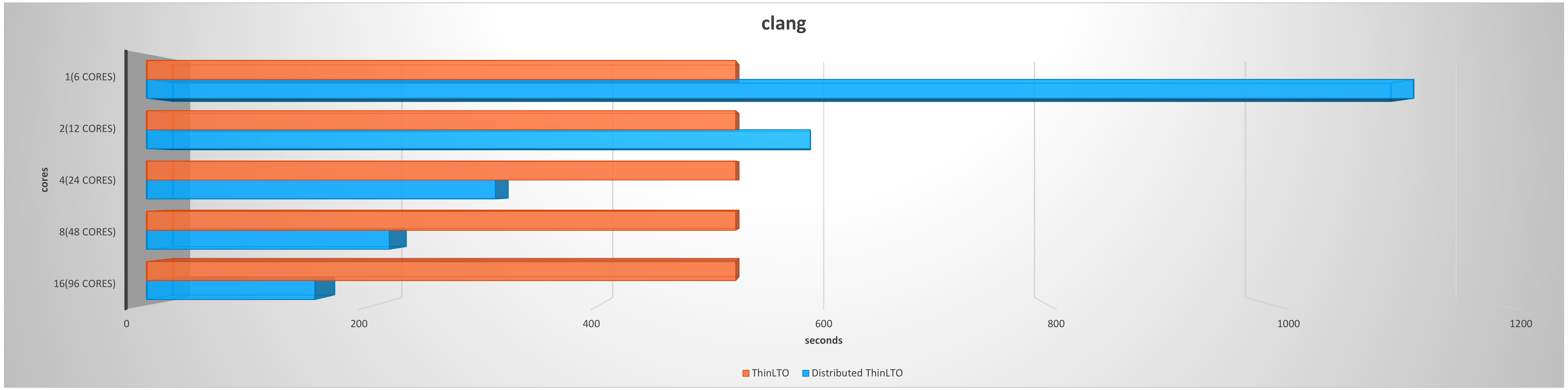
Distributed build performance obviously depends on a deployed hardware infrastructure. Bigger number of faster executor nodes alongside with fast network communications yields better distributed performance.

Distributed build performance is also very much input data dependent – typically best distributed performance can be achieved when all subjects of a build are large (more precisely require significant processing time) and, they have approximately equal size. Big number of small files significantly increase a distributed system cumulative service time against cumulative processing time. On the other hand, small number of significantly unequal files would prevent to archive good balanced load on a pool of distributed executor nodes.

Linking time for the Clang project. Distributed ThinLTO vs. regular ThinLTO.

We made linking time comparison for regular (multi-threaded) ThinLTO vs Distributed ThinLTO on pools of 1, 2, 4, 8, and 16 PCs.

The Distributed ThinLTO does have overhead and does significantly more file I/O and networking relative to ThinLTO, hence on a pool of one PC, the Distributed ThinLTO is slower, as expected. On two PCs the performance is almost even, and on four or more PCs the Distributed ThinLTO has good performance advantage.



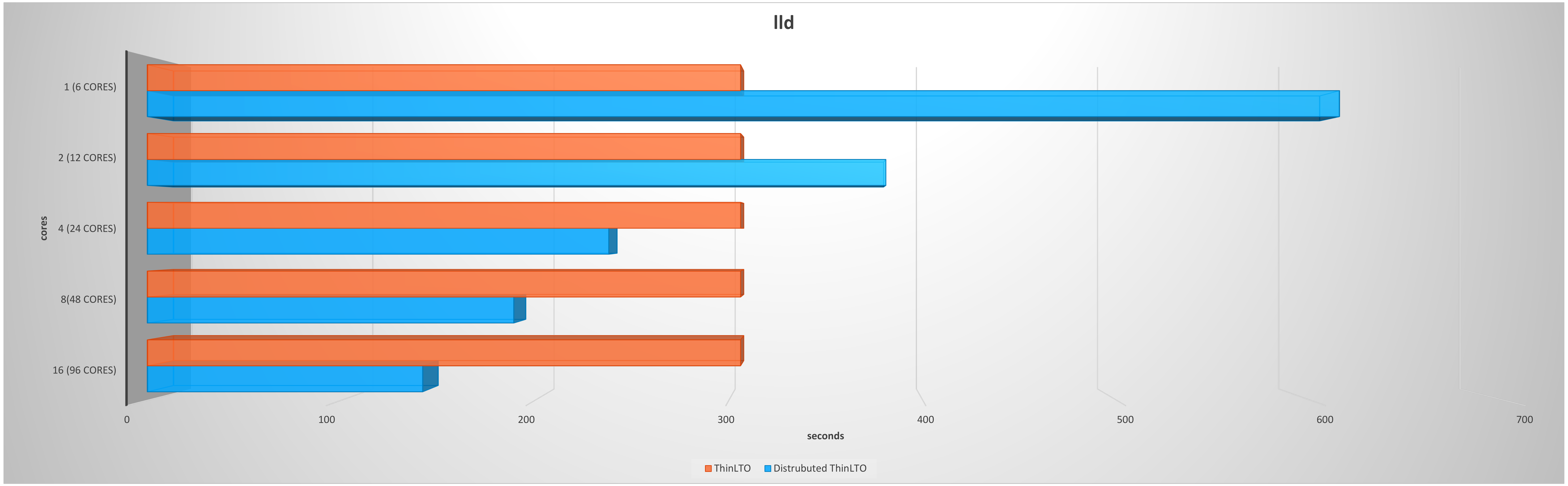
Typical PC has Xeon E5-1650 CPU – 6 physical cores, 32GB memory, 1Gb network.

Note that on 16 PCs Distributed ThinLTO is 3.5 times faster than regular ThinLTO.

Linking time for lld project. Distributed ThinLTO vs. regular ThinLTO.

Similar results as for the clang project. Starting from 4 PCs pool Distrusted ThinLTO has better performance.

Note that on 16 PCs Distributed ThinLTO is 2.2 times faster than regular ThinLTO.



Future Work

We are planning to send RFC for the Integrated Distributed ThinLTO to the LLVM mailing list and start upstreaming our work to LLVM and LLD. Our design is modular and the support for many other distributed build systems could be added with the minimal efforts.

If there is sufficient interest in LLVM community to integrate the Distributed ThinLTO with Icecream, we will upstream our modifications to Icecream as well.

Thanks for listening!

Sony
Interactive
Entertainment