Link-Time Optimization without Linker Support

Yunzhong Gao
Sony Computer Entertainment

LLVM Dev Meeting, 7 Nov 2013

Motivation

- PlayStation®4 toolchain is based on Clang
- Uses a proprietary linker which does not (yet) understand LLVM libLTO plugin
- Game developers looking for every last bit of performance
- Would LTO in linker be useful for game developers? Can we get LTO now, before doing that work?

Taking a Huge Detour…

- Compile each source to bitcode

Input bc files

Taking a Huge Detour…

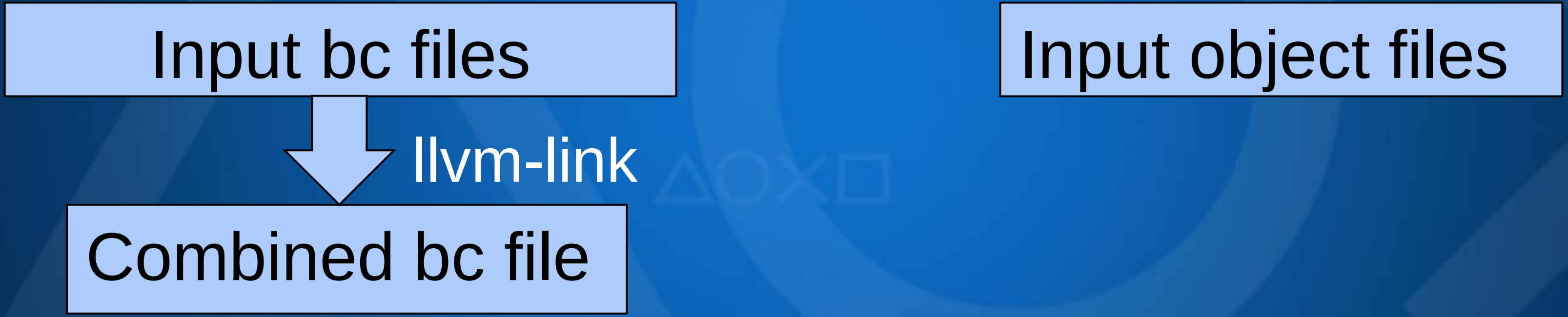
- Save the libraries and other files for later

Input bc files

Input object files

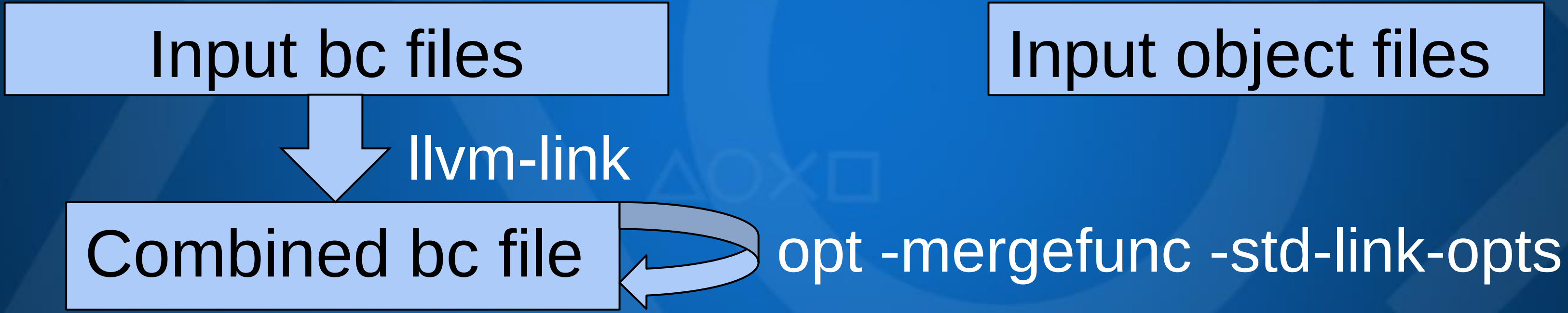
Taking a Huge Detour…

- Run llvm-link to combine several input .bc files into one



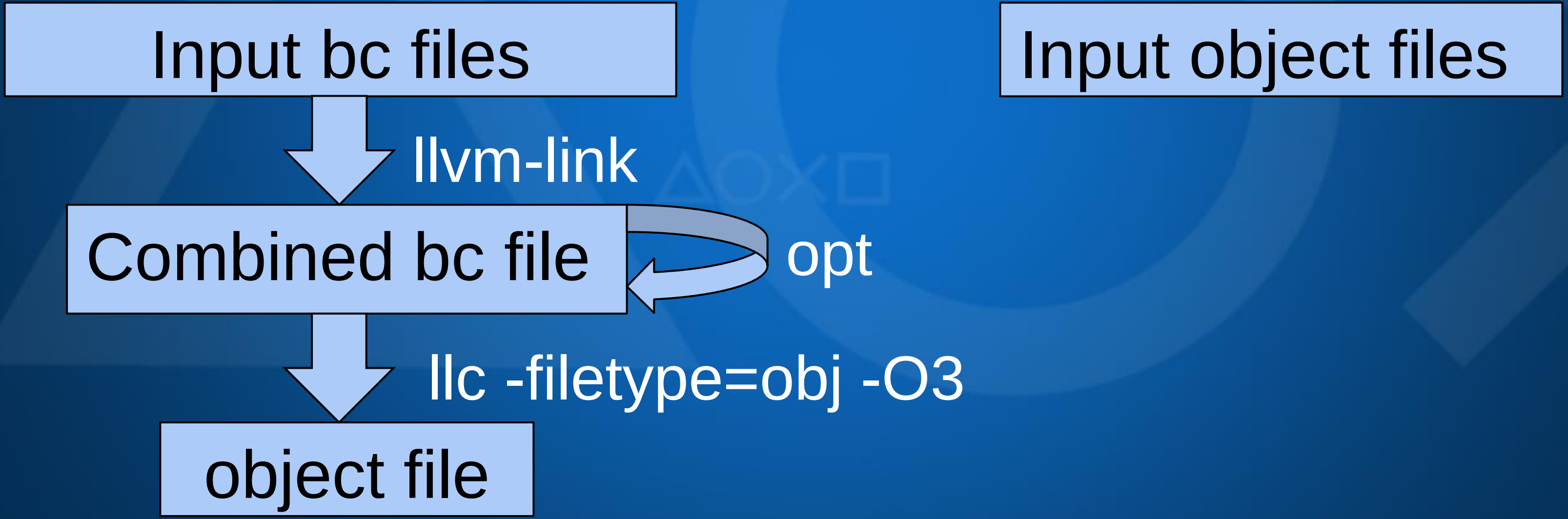
Taking a Huge Detour...

- Run opt on the combined .bc file



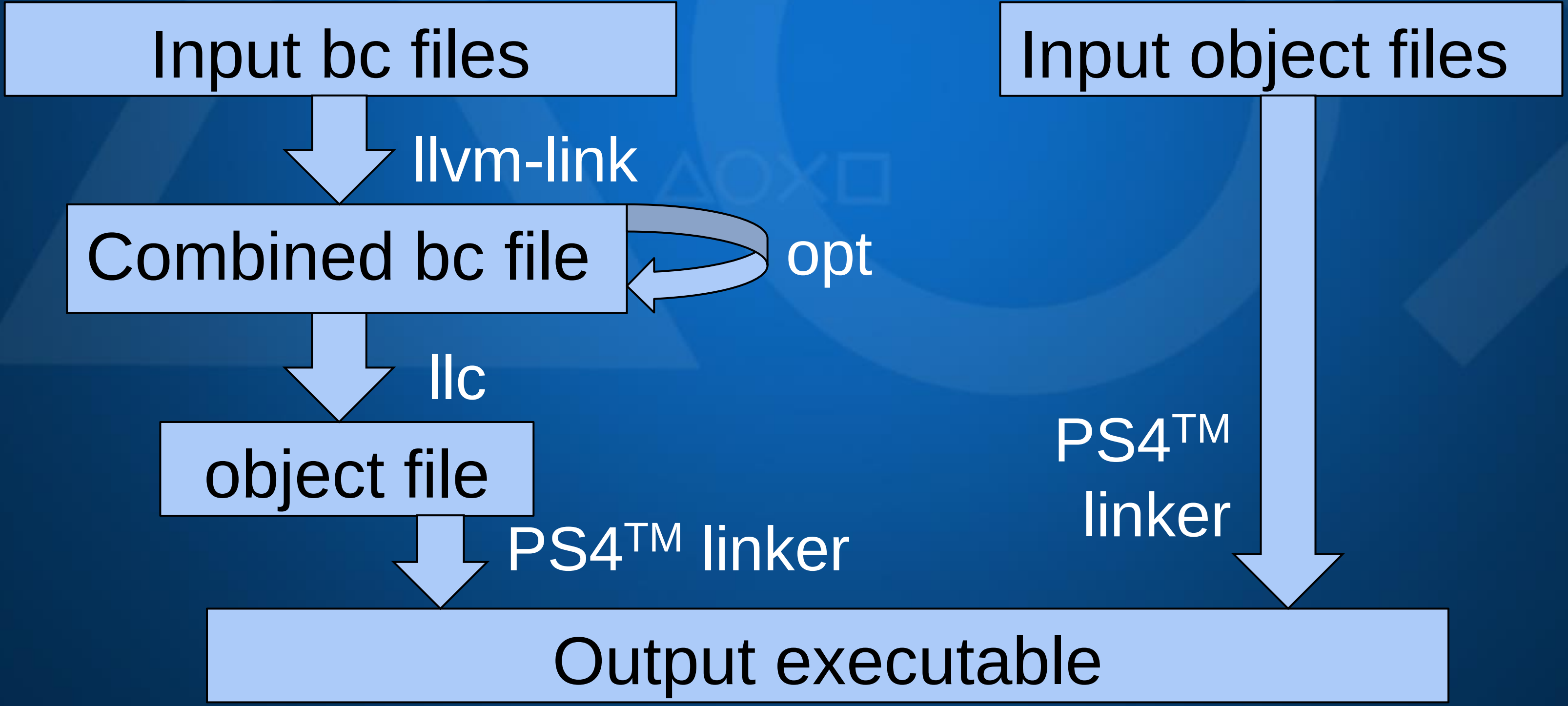
Taking a Huge Detour…

- Run llc to generate combined .o file



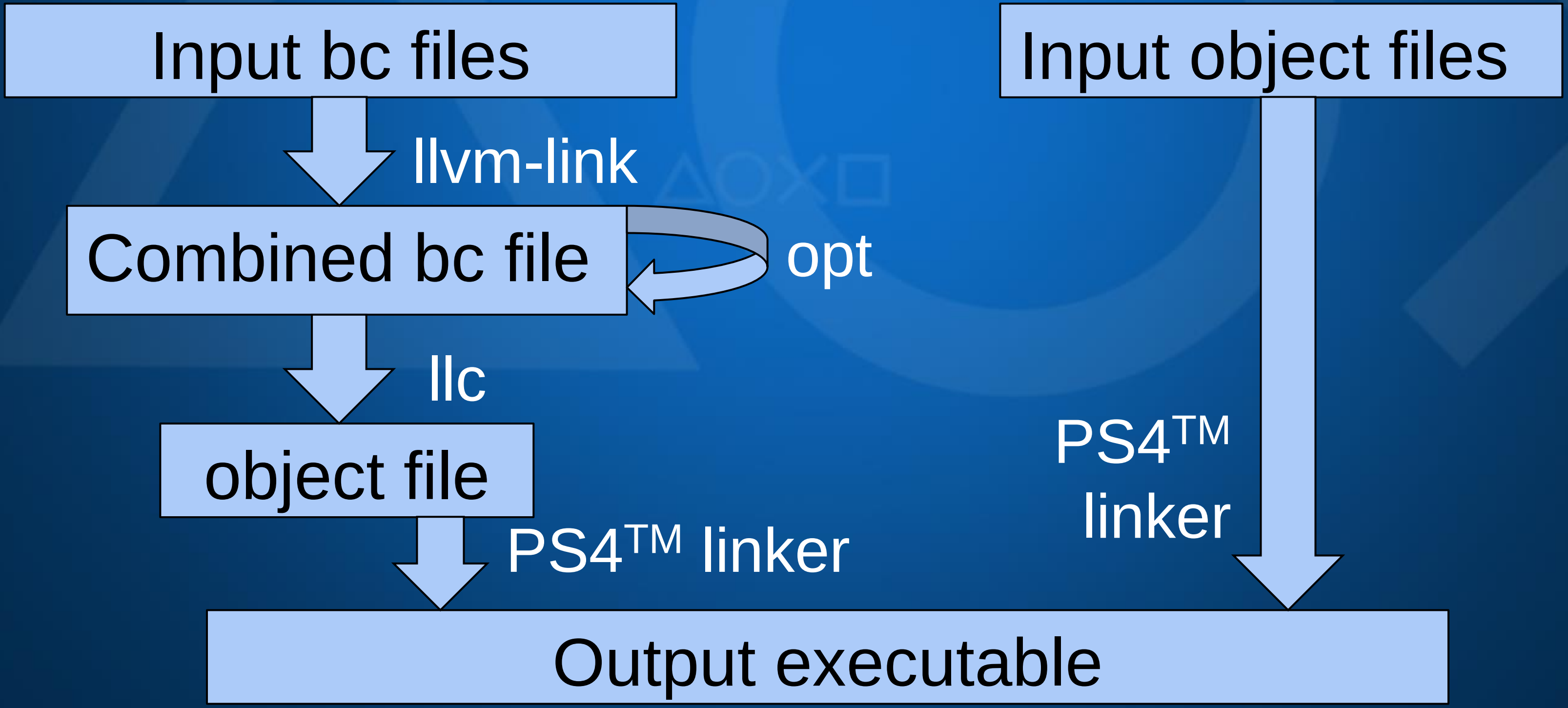
Taking a Huge Detour…

- Normal link with the combined .o file



Taking a Huge Detour…

- Not fun to patch this into an existing build process…



Trick the build process

- Write a Python script to do all the hard parts
- Rename original linker, replace it with our script
- Add –flto to the compilation steps to generate bitcode
- Link step runs our script
- Hey, it kind of works!
  - Enough for experimentation/evaluation, anyway
  - Limitation: opt does not know what symbols are referenced externally; need to mark some stuff with __attribute__((used))

LTO is worth the trouble

Bullet benchmark:

- Memory footprint reduced > 50% at -O2
  - Text size reduced ~ 15%
  - Data size reduced ~ 45%
  - BSS size reduced > 90%
- Size improvements similar to linker's dead-stripping
  - Dead-stripping is cheaper in build-time
- Execution time reduced ~ 5% versus non-LTO at -O2

LTO is worth the trouble

One major PS4™ launch title tried this LTO implementation, and has seen improvement:

- ~10% code size
- ~6% run time