

Optimizing real-world applications with GCC Link Time Optimization

Taras Glek
Mozilla Corporation

Honza Hubička
SuSE ČR s.r.o

GCC Summit, 2010

Outline

- 1 Basic overview of LTO
- 2 Compiling large applications
- 3 Problems specific for large applications

Link Time Optimization and Inter Procedural Analysis

- *Link time optimization* (LTO) extends the scope of interprocedural analysis from single source file to whole program visible at the link time

Link Time Optimization and Inter Procedural Analysis

- *Link time optimization* (LTO) extends the scope of interprocedural analysis from single source file to whole program visible at the link time
 - Implemented by calling back to the optimizer backend from the linker.
 - Development started in 2005, merged to mainline in 2009.
 - First released in GCC 4.5.

Link Time Optimization and Inter Procedural Analysis

- *Link time optimization* (LTO) extends the scope of interprocedural analysis from single source file to whole program visible at the link time
 - Implemented by calling back to the optimizer backend from the linker.
 - Development started in 2005, merged to mainline in 2009.
 - First released in GCC 4.5.
- *Interprocedural analysis* (IPA) and optimization is about optimizing across function boundaries.

Link Time Optimization and Inter Procedural Analysis

- *Link time optimization* (LTO) extends the scope of interprocedural analysis from single source file to whole program visible at the link time
 - Implemented by calling back to the optimizer backend from the linker.
 - Development started in 2005, merged to mainline in 2009.
 - First released in GCC 4.5.
- *Interprocedural analysis* (IPA) and optimization is about optimizing across function boundaries.
 - GCC callgraph module, in GCC mainline since 2003

Basic components

- 1 Infrastructure for streaming an intermediate language to disk

Basic components

- 1 Infrastructure for streaming an intermediate language to disk
- 2 A new compiler front-end (lto1)

Basic components

- 1 Infrastructure for streaming an intermediate language to disk
- 2 A new compiler front-end (lto1)
- 3 A linker plugin integrated into the Gold linker

Basic components

- 1 Infrastructure for streaming an intermediate language to disk
- 2 A new compiler front-end (`lto1`)
- 3 A linker plugin integrated into the Gold linker
- 4 Modifications to the GCC driver (`collect2`) to support linking of LTO object files using either the linker plugin or direct invocation of the LTO front-end,

Basic components

- 1 Infrastructure for streaming an intermediate language to disk
- 2 A new compiler front-end (`lto1`)
- 3 A linker plugin integrated into the Gold linker
- 4 Modifications to the GCC driver (`collect2`) to support linking of LTO object files using either the linker plugin or direct invocation of the LTO front-end,
- 5 Various middle-end infrastructure updates
(Symbol table representation, support for merging of declarations and types etc. . .)

Basic components

- 1 Infrastructure for streaming an intermediate language to disk
- 2 A new compiler front-end (`lto1`)
- 3 A linker plugin integrated into the Gold linker
- 4 Modifications to the GCC driver (`collect2`) to support linking of LTO object files using either the linker plugin or direct invocation of the LTO front-end,
- 5 Various middle-end infrastructure updates
(Symbol table representation, support for merging of declarations and types etc. . .)
- 6 Support for using the linker plugin in the tool-chain
(`ar` and `nm`)

Basic components

- 1 Infrastructure for streaming an intermediate language to disk
- 2 A new compiler front-end (`lto1`)
- 3 A linker plugin integrated into the Gold linker
- 4 Modifications to the GCC driver (`collect2`) to support linking of LTO object files using either the linker plugin or direct invocation of the LTO front-end,
- 5 Various middle-end infrastructure updates
(Symbol table representation, support for merging of declarations and types etc. . .)
- 6 Support for using the linker plugin in the tool-chain
(`ar` and `nm`)
- 7 Libtool update to handle LTO

On disk representation

- Program is represented in GIMPLE IL in the SSA form

On disk representation

- Program is represented in GIMPLE IL in the SSA form
- Intermediate language is streamed into target object files
 - Allows integration with the rest of toolchain (producing archives etc.)
 - Supports “fat” object files with both the IL and assembly

On disk representation

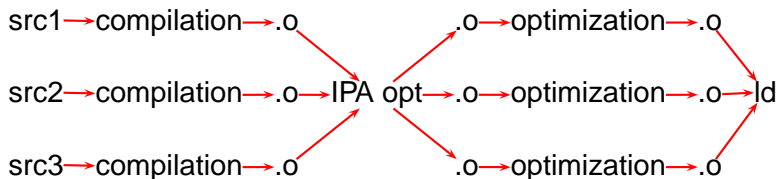
- Program is represented in GIMPLE IL in the SSA form
- Intermediate language is streamed into target object files
 - Allows integration with the rest of toolchain (producing archives etc.)
 - Supports “fat” object files with both the IL and assembly
- LTO information is structured into several sections of the object file.
 - **Command line options** (`.gnu.lto_.opts`)
 - **The symbol table** (`.gnu.lto_.symtab`)
 - **Global declarations and types** (`.gnu.lto_.decls`).
 - **The callgraph** (`.gnu.lto_.cgraph`).
 - **IPA references** (`.gnu.lto_.refs`).
 - **Function bodies**
 - **Static variable initializers** (`.gnu.lto_.vars`).
 - **Summaries and optimization summaries.**

LTO versus WHOPR

- LTO reads whole program into memory at link time and optimizes it as single compilation unit
- WHOPR mode allows parallelization of the local optimization stage.

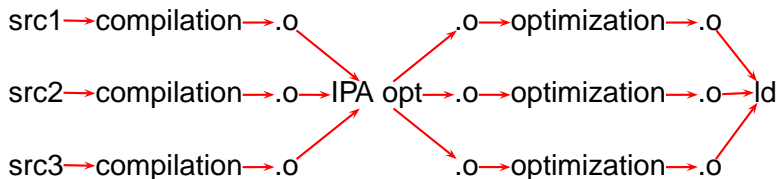
LTO versus WHOPR

- LTO reads whole program into memory at link time and optimizes it as single compilation unit
- WHOPR mode allows parallelization of the local optimization stage.



LTO versus WHOPR

- LTO reads whole program into memory at link time and optimizes it as single compilation unit
- WHOPR mode allows parallelization of the local optimization stage.



- 3 stage compilation process
- only IPA propagation stage sees whole program and is not executed in parallel
- WHOPR does not work in GCC 4.5. In GCC 4.6 it will replace LTO by default

3 stages of WHOPR

- LGEN (compile time — parallel via make)
 - Parsing
 - early optimization
 - function summaries production
 - streaming
 - late compilation for “fat objects”

3 stages of WHOPR

- LGEN (compile time — parallel via make)
 - Parsing
 - early optimization
 - function summaries production
 - streaming
 - late compilation for “fat objects”
- WPA (link time — serial)
 - Merge declarations and types
 - produce combined callgraph
 - interprocedural optimizations
 - streaming

3 stages of WHOPR

- LGEN (compile time — parallel via make)
 - Parsing
 - early optimization
 - function summaries production
 - streaming
 - late compilation for “fat objects”
- WPA (link time — serial)
 - Merge declarations and types
 - produce combined callgraph
 - interprocedural optimizations
 - streaming
- LTRANS (link time — parallel via temporary Makefile)
 - Apply results of interprocedural optimizations
 - late optimization; production of assembly

WHOPR Interprocedural optimization pass

To make WHOPR possible, inter-procedural optimization passes are split to the following stages:

- LGEN time:
 1. **Generate summary**
 2. **Write summary**

WHOPR Interprocedural optimization pass

To make WHOPR possible, inter-procedural optimization passes are split to the following stages:

- LGEN time:
 1. **Generate summary**
 2. **Write summary**
- WPA time:
 3. **Read summary**
 4. **Execute**
 5. **Write optimization summary**

WHOPR Interprocedural optimization pass

To make WHOPR possible, inter-procedural optimization passes are split to the following stages:

- LGEN time:
 1. **Generate summary**
 2. **Write summary**
- WPA time:
 3. **Read summary**
 4. **Execute**
 5. **Write optimization summary**
- LTRANS time:
 6. **Read optimization summary**
 7. **Transform**

Inter-procedural optimization infrastructure

- **Callgraph**: multi-graph where functions are nodes and call sites edges
- **Varpool**: list of static variables and initializers
- **IPA references**: Multi-graph across function and variables representing references (read, writes and addresses taken)
- **Jump functions** summarizing inter-procedural dataflow
- **Pass manager**
- **inter-procedural passes**

Pass ordering issues

- In classical LTO modes passes execute in sequence.
- In WHOPR passes execute “in parallel”
 - All passes perform analysis
 - All passes perform IP propagation
 - All passes apply changes

Pass ordering issues

- In classical LTO modes passes execute in sequence.
 - In WHOPR passes execute “in parallel”
 - All passes perform analysis
 - All passes perform IP propagation
 - All passes apply changes
- “parallel” execution if IP passes leads to ordering issues
- Virtual clones was introduced to avoid need for pass specific transform.

Pass ordering issues

- In classical LTO modes passes execute in sequence.
- In WHOPR passes execute “in parallel”

- All passes perform analysis
- All passes perform IP propagation
- All passes apply changes

“parallel” execution if IP passes leads to ordering issues

- Virtual clones was introduced to avoid need for pass specific transform.

Virtual clone is:

- A node in callgraph like normal function
- Unlike normal function has no body
- Has pointer to its master and description how to create clone from it.

Pass ordering issues

- In classical LTO modes passes execute in sequence.
- In WHOPR passes execute “in parallel”

- All passes perform analysis
- All passes perform IP propagation
- All passes apply changes

“parallel” execution if IP passes leads to ordering issues

- Virtual clones was introduced to avoid need for pass specific transform.
Virtual clone is:
 - A node in callgraph like normal function
 - Unlike normal function has no body
 - Has pointer to its master and description how to create clone from it.
- Callgraph hooks are available to maintain pass specific info consistent.

The Linker plugin

Linker plugin integrate into Gold linker and:

The Linker plugin

Linker plugin integrate into Gold linker and:

- 1 Recognize objects containing GCC LTO sections and claims them

The Linker plugin

Linker plugin integrate into Gold linker and:

- 1 Recognize objects containing GCC LTO sections and claims them
- 2 Reads LTO symbol table and pass it to linker for linking

The Linker plugin

Linker plugin integrate into Gold linker and:

- 1 Recognize objects containing GCC LTO sections and claims them
- 2 Reads LTO symbol table and pass it to linker for linking
- 3 After linking is decides obtain resolution info
 - Information about resolved, prevailed and prevailing symbols
 - Information whether given symbol is used outside LTO code

The Linker plugin

Linker plugin integrate into Gold linker and:

- 1 Recognize objects containing GCC LTO sections and claims them
- 2 Reads LTO symbol table and pass it to linker for linking
- 3 After linking is decides obtain resolution info
 - Information about resolved, prevailed and prevailing symbols
 - Information whether given symbol is used outside LTO code
- 4 Save resolution info into file and execute GCC

The Linker plugin

Linker plugin integrate into Gold linker and:

- 1 Recognize objects containing GCC LTO sections and claims them
- 2 Reads LTO symbol table and pass it to linker for linking
- 3 After linking is decides obtain resolution info
 - Information about resolved, prevailed and prevailing symbols
 - Information whether given symbol is used outside LTO code
- 4 Save resolution info into file and execute GCC
- 5 Adds GCC produced object files into the linker.

The Linker plugin

Linker plugin integrate into Gold linker and:

- 1 Recognize objects containing GCC LTO sections and claims them
- 2 Reads LTO symbol table and pass it to linker for linking
- 3 After linking is decides obtain resolution info
 - Information about resolved, prevailed and prevailing symbols
 - Information whether given symbol is used outside LTO code
- 4 Save resolution info into file and execute GCC
- 5 Adds GCC produced object files into the linker.

Linker is independent of GCC LTO infrastructure and Gold.
LLVM use the plugin, GNU LD is being updated, too.

Linker plugin and whole program assumptions

- Functions & vars not declared `static` prevent inter-procedural optimization.

Linker plugin and whole program assumptions

- Functions & vars not declared `static` prevent inter-procedural optimization.
- Optimization works a lot better when compiler assume they are now *whole program assumptions*.

Linker plugin and whole program assumptions

- Functions & vars not declared `static` prevent inter-procedural optimization.
- Optimization works a lot better when compiler assume they are now *whole program assumptions*.
- `-fwhole-program` makes GCC to declare every function and var static to the linktime unit.
- `externally_visible` attribute overwrite the effect (`main()` is implicitly externally visible).

Linker plugin and whole program assumptions

- Functions & vars not declared `static` prevent inter-procedural optimization.
- Optimization works a lot better when compiler assume they are now *whole program assumptions*.
- `-fwhole-program` makes GCC to declare every function and var static to the linktime unit.
- `externally_visible` attribute overwrite the effect (`main()` is implicitly externally visible).
- `-fwhole-program` does not fit shared libraries (users would need to annotate all of the interface).
 - To speedup dynamic linking a lot of libraries use `visibility ("hidden")` or `-fdefault-visibility=hidden`.
 - GCC use linker plugin to see what non-LTO objects use.
 - Without linker plugin hidden symbols are implicitly brought local.

Optimizations performed

- Early optimization (at compile time)
 - Into-SSA conversion

Optimizations performed

- Early optimization (at compile time)
 - Into-SSA conversion
 - Early inlining

Optimizations performed

- Early optimization (at compile time)
 - Into-SSA conversion
 - Early inlining
 - constant propagation, copy propagation, dead code elimination, and scalar replacement.

Optimizations performed

- Early optimization (at compile time)
 - Into-SSA conversion
 - Early inlining
 - constant propagation, copy propagation, dead code elimination, and scalar replacement.
 - Inter-procedural scalar replacement

Optimizations performed

- Early optimization (at compile time)
 - Into-SSA conversion
 - Early inlining
 - constant propagation, copy propagation, dead code elimination, and scalar replacement.
 - Inter-procedural scalar replacement
 - Tail recursion elimination

Optimizations performed

- Early optimization (at compile time)
 - Into-SSA conversion
 - Early inlining
 - constant propagation, copy propagation, dead code elimination, and scalar replacement.
 - Inter-procedural scalar replacement
 - Tail recursion elimination
 - Exception handling optimizations

Optimizations performed

- Early optimization (at compile time)
 - Into-SSA conversion
 - Early inlining
 - constant propagation, copy propagation, dead code elimination, and scalar replacement.
 - Inter-procedural scalar replacement
 - Tail recursion elimination
 - Exception handling optimizations
 - Static profile estimation

Optimizations performed

- Early optimization (at compile time)
 - Into-SSA conversion
 - Early inlining
 - constant propagation, copy propagation, dead code elimination, and scalar replacement.
 - Inter-procedural scalar replacement
 - Tail recursion elimination
 - Exception handling optimizations
 - Static profile estimation
 - Attributes discovery

Optimizations performed

- Early optimization (at compile time)
 - Into-SSA conversion
 - Early inlining
 - constant propagation, copy propagation, dead code elimination, and scalar replacement.
 - Inter-procedural scalar replacement
 - Tail recursion elimination
 - Exception handling optimizations
 - Static profile estimation
 - Attributes discovery
 - Function splitting

Optimizations performed

- Early optimization (at compile time)

Optimizations performed

- Early optimization (at compile time)
- Whole program visibility

Optimizations performed

- Early optimization (at compile time)
- Whole program visibility
- IPA profile propagation

Optimizations performed

- Early optimization (at compile time)
- Whole program visibility
- IPA profile propagation
- Constant propagation

Optimizations performed

- Early optimization (at compile time)
- Whole program visibility
- IPA profile propagation
- Constant propagation
- Constructor and destructor merging

Optimizations performed

- Early optimization (at compile time)
- Whole program visibility
- IPA profile propagation
- Constant propagation
- Constructor and destructor merging
- Inlining

Optimizations performed

- Early optimization (at compile time)
- Whole program visibility
- IPA profile propagation
- Constant propagation
- Constructor and destructor merging
- Inlining
- Function attributes

Optimizations performed

- Early optimization (at compile time)
- Whole program visibility
- IPA profile propagation
- Constant propagation
- Constructor and destructor merging
- Inlining
- Function attributes
- MOD/REF analysis (ipa-reference)
New `leaf` function attribute.

Optimizations performed

- Early optimization (at compile time)
- Whole program visibility
- IPA profile propagation
- Constant propagation
- Constructor and destructor merging
- Inlining
- Function attributes
- MOD/REF analysis (ipa-reference)
New `leaf` function attribute.
- Experimental passes

Outline

- 1 Basic overview of LTO
- 2 Compiling large applications**
- 3 Problems specific for large applications

Firefox and GCC

● Firefox

- Main shared lib `libxul.so` is about 6 000 000 lines of code.
- Links statically many libraries built in tree (libffi, cairo, gtk etc. . .).
- Developed and tested with LTO on other compilers (i.e. MSVC)
- Mostly portable C++ code but use many performance related features (visibility aliases, asm hunks etc.)

Firefox and GCC

● Firefox

- Main shared lib `libxul.so` is about 6 000 000 lines of code.
- Links statically many libraries built in tree (`libffi`, `cairo`, `gtk` etc. . .).
- Developed and tested with LTO on other compilers (i.e. MSVC)
- Mostly portable C++ code but use many performance related features (visibility aliases, `asm` hunks etc.)

● GCC

- 800 000 lines of hand written C code, 500 000 lines of auto generated.

Serial GCC Build times

- GCC non-LTO build time: 8m12s
- GCC LTO link time: 6m31s

Serial GCC Build times

- GCC non-LTO build time: 8m12s
- GCC LTO link time: 6m31s
 - Reading the IL: 3%
 - Merging of declarations: 1%.
 - Outputting of the assembly file: 2%.
 - Debug information generation (`var-tracking` and `symout`): 8%.
 - Garbage collection: 2%.
 - Local optimizations: rest
 - partial redundancy elimination (5%), GIMPLE to RTL expansion (8%), RTL level dataflow analysis (11%), instruction combining (3%), register allocation (6%), scheduling (5%).

Parallel GCC Build times (24 cores)

- GCC non-LTO build time: 56s
- GCC LTO link time: 48s

Parallel GCC Build times (24 cores)

- GCC non-LTO build time: 56s
- GCC LTO link time: 48s
Serial WPA stage: 19s
 - Reading global declarations and types: 28% of the overall time taken by the WPA stage.
 - Merging declarations: 6%.
 - Inter-procedural optimization: 9%.
 - Streaming of object files to be passed to LTRANS: 42%.

Serial Firefox Build times

- Firefox non-LTO build time: 39m
- Firefox LTO link time: 19m29s

Serial Firefox Build times

- Firefox non-LTO build time: 39m
- Firefox LTO link time: 19m29s
 - Reading of the IL: 7%.
 - Merging of declarations: 4%.
 - Output of the assembly file: 3%.
 - Debug information generation is disabled in our builds.
 - Garbage collection: 2%.
 - Local optimizations: the rest
operand scan (5%), partial redundancy elimination (5%),
GIMPLE to RTL expansion (13%), RTL level dataflow
analysis (5%), instruction combining (3%), register
allocation (9%), scheduling (3%).

Parallel Firefox Build times (24 cores)

- Firefox non-LTO build time: 9m38s
- Firefox LTO link time: 5m30s

Parallel Firefox Build times (24 cores)

- Firefox non-LTO build time: 9m38s
- Firefox LTO link time: 5m30s
Serial WPA stage: 4m24s
 - Reading global declarations and types: 24%.
 - Merging declarations: 20%.
 - Inter-procedural optimization: 8%.
 - Streaming of object files to be passed to LTRANS: 28%.
 - Callgraph and WPA overhead (callgraph merging and partitioning): 12%.

Memory usage

- LTO GCC: 2GB
- LTO Firefox: 8.5GB

Memory usage

- LTO GCC: 2GB
- LTO Firefox: 8.5GB
- WHOPR GCC: 415MB for WPA, LTRANS compilations all less than 400MB
 - Memory mapped object files: 170MB (not all of it is paged in)
 - Types and declarations: 260MB
 - Callgraph, varpool and other IPA stuff: 52MB

Memory usage

- LTO GCC: 2GB
- LTO Firefox: 8.5GB
- WHOPR GCC: 415MB for WPA, LTRANS compilations all less than 400MB
 - Memory mapped object files: 170MB (not all of it is paged in)
 - Types and declarations: 260MB
 - Callgraph, varpool and other IPA stuff: 52MB
- WHOPR Firefox: 4GB for WPA
3.7GB declarations and types

Code quality

- GCC gets faster only with -O3 (by about 4% on non-optimizing compilation of combine.c)
- Firefox gets a bit faster overall

benchmark name	speedup
dromeao css	1.83%
tdhtml	-0.54%
tp_dist	0.50%
tsvg	0.07%

Both projects was tuned for file-by-file compilation. Cross module optimizations are limited then.

Code quality II

- GCC gets 6% smaller (-O2)
- Firefox gets 6% smaller (-O3)
 - Reducing inlining unit growth save additional 12%.
Resulting -O3 -param inline-unit-growth=5 -flto binary is of same size as -Os binary!
 - At -Os GCC produce 11% smaller Firefox, LLVM is reported to save 13%.

Code quality III

	speedup	size
perlbench	+1.4%	+4%
bzip2	+2.6%	-45%
gcc	-0.3%	+1.2%
mcf	+1.9%	-33%
gobmk	+3.4%	+1.8%
hmmer	+0.8%	-55%
sjeng	+1.2%	-11%
libquantum	-0.5%	-61%
h264ref	+7.0%	-9%
omnetpp	-0.8%	-11%
astar	-1.3%	-20%

Code quality IV

	speedup	size
bwaves	0% (+15%)	-27%
gamess	-0.7%	-50%
milc	+2.2%	-26%
zeusmp	+0.4%	-27%
gromacs	0%	-18%
cactusADM	-0.8%	-42%
leslie3d	-2.1% (0%)	+0.6%
namd	0%	-40%
soplex	+1.5%	-50%
povray	+5%	-2.3%
calculix	1.1%	-38%
GemsFDTD	0%	-70%
tonto	-0.2%	-25%
lbm	+3.2%	0%
wrf	0%	-36%
sphinx3	+2.9%	-32%

SPEC2000 are easier

- 11% on EON
- 5% on Perl.
- 2.5% on GCC.
- 17% on Vortex.
- 7% on bzip.
- 33% on wupwise, but it is gone probably because of profile issues
- 4% on Applu
- 2% on ART.

Outline

- 1 Basic overview of LTO
- 2 Compiling large applications
- 3 Problems specific for large applications**

Code size and speed tradeoffs

- For common benchmarks code size is rarely issue
- Tuning GCC on benchmarks leads to code size growth (`-O2` and `-O3`)
- Firefox builds by default with `-Os`, they will switch to `-O3` because it is faster
- Linux kernel is often `-Os` too

Code size and speed tradeoffs

- For common benchmarks code size is rarely issue
- Tuning GCC on benchmarks leads to code size growth (`-O2` and `-O3`)
- Firefox builds by default with `-Os`, they will switch to `-O3` because it is faster
- Linux kernel is often `-Os` too

We need to take more care to `-O2` file size tradeoffs.

Code size and speed tradeoffs

- For common benchmarks code size is rarely issue
- Tuning GCC on benchmarks leads to code size growth (`-O2` and `-O3`)
- Firefox builds by default with `-Os`, they will switch to `-O3` because it is faster
- Linux kernel is often `-Os` too

We need to take more care to `-O2` file size tradeoffs.

LTO can help here:

- ipa-profile pass (not terribly effective)
- Global inliner and cloning decisions
- Whole program assumptions leads to large code size improvements

Code size and speed tradeoffs

- For common benchmarks code size is rarely issue
- Tuning GCC on benchmarks leads to code size growth (`-O2` and `-O3`)
- Firefox builds by default with `-Os`, they will switch to `-O3` because it is faster
- Linux kernel is often `-Os` too

We need to take more care to `-O2` file size tradeoffs.

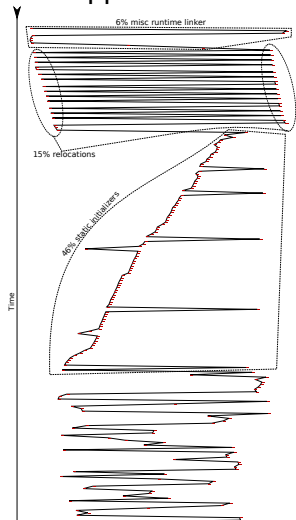
LTO can help here:

- ipa-profile pass (not terribly effective)
- Global inliner and cloning decisions
- Whole program assumptions leads to large code size improvements

Profile feedback helps even more. Firefox is going to use it.

Startup times

C++ applications tends to load slowly.



Code locality improvements

- Constructor merging pass (LTO only)
- Placing static ctors/dtors in special subsections
- Function reordering
 - Graph clustering techniques seems to work just slightly better than simple DFS order
 - Callgraph lacks virtual function calls so the pass is confused at Firefox
- Profile feedback driven function reordering

Other improvements

- Reducing amount of relocations

Other improvements

- Reducing amount of relocations
 - Privatizing comdat functions
 - Optimizing out some of vtables (30% reduction of `.data.rel.ro.local`)

Other improvements

- Reducing amount of relocations
 - Privatizing comdat functions
 - Optimizing out some of vtables (30% reduction of `.data.rel.ro.local`)

About 5% smaller dynamic linker table, 0.6% fewer relocations.

Other improvements

- Reducing amount of relocations
 - Privatizing comdat functions
 - Optimizing out some of vtables (30% reduction of `.data.rel.ro.local`)

About 5% smaller dynamic linker table, 0.6% fewer relocations.

- C++ API is not too PIC friendly, perhaps we can do local conventions

Other improvements

- Reducing amount of relocations

- Privatizing comdat functions
- Optimizing out some of vtables (30% reduction of `.data.rel.ro.local`)

About 5% smaller dynamic linker table, 0.6% fewer relocations.

- C++ API is not too PIC friendly, perhaps we can do local conventions
- Data section locality improvements
 - Order data sections to match references from code
 - Group structures used by static constructions

Thank you!

Questions?