When -O3 is not enough

Dominik Adamski
CEHUG
Łódź, 13.11.2019

About me

- Senior Software Engineer in Mobica
- Experienced with embedded systems
- Interested in high performance computing and advanced code optimization techniques

Scope of the presentation

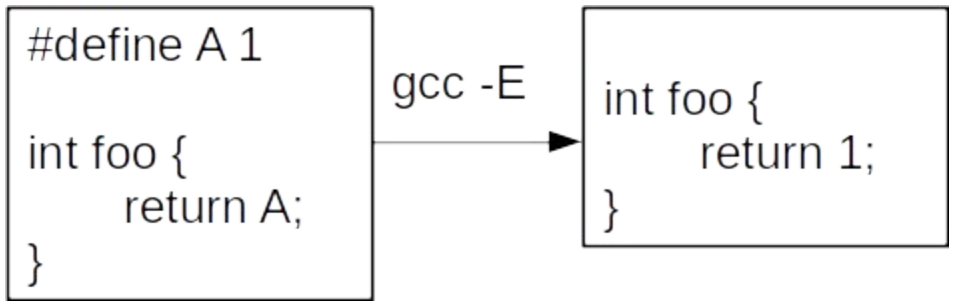
- Compiler (gcc and clang) support for optimization of the C/C++ code
- Review of existing code optimization techniques
- Efficiency of LTO and PGO optimization
- Recommendations
- Further reading

Motivation

- Software is getting more and more complex
- Users expect better and better performance
- No more performance via speedup of processor frequency
- Constantly increasing costs of software development
- Large amount of legacy code

Introduction: From source code to application - preprocessing

- Executed for every source file which needs to be compiled
- Insert content of the header files into source files.
- Resolve preprocessor directives (#include, #ifdef, #define etc.)



Introduction: From source code to application – object file generation

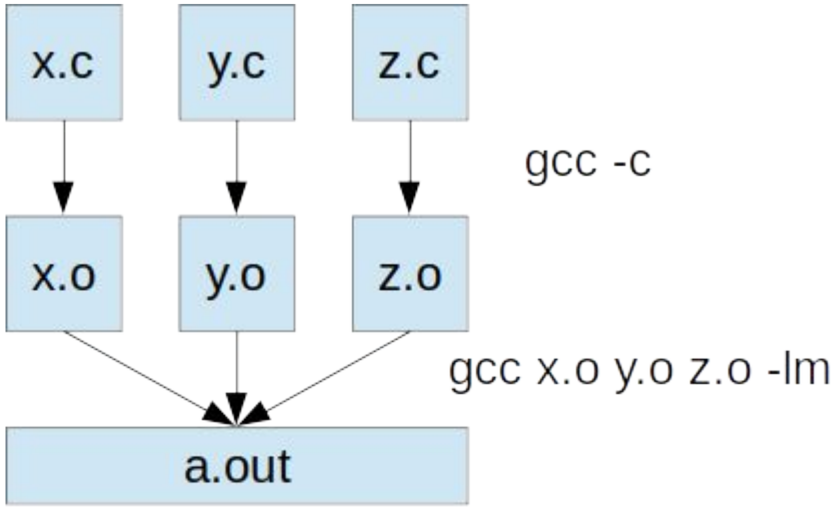
- Executed for every source file which needs to be compiled
- Every source file can be handled independently
- It can include optimization
- Scope of optimization is defined by the user
- Optimizer does not know how given code will be used by other modules

Introduction: From source code to application – linking

- Combines several object files into an executable application
- Attaches system libraries (for example: thread, math library)
- Can be dynamic or static
- Modification of any source file causes re-linking of the whole binary
- Checks if all required symbols are defined (static linking)

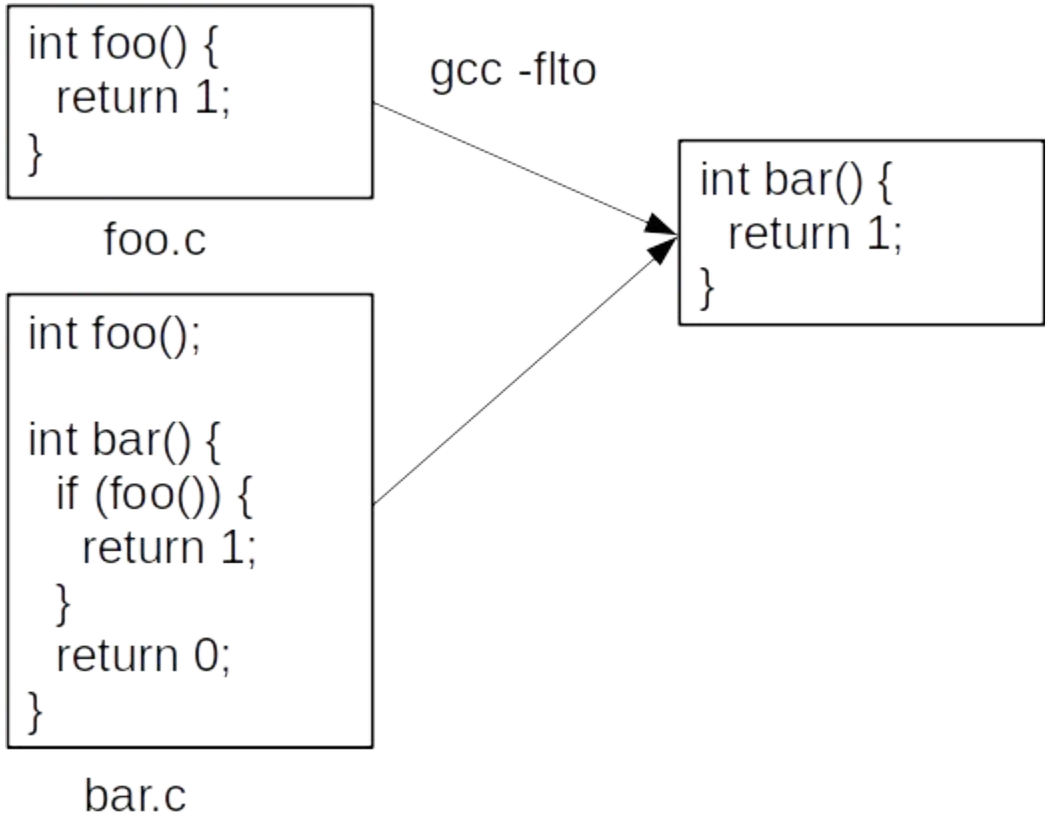
Project building without LTO

- Object files can be generated in parallel
- Building process can be well-scaled on multi-core machines
- Modification of one source file does not trigger rebuilding the whole project
- Linking phase is minimized



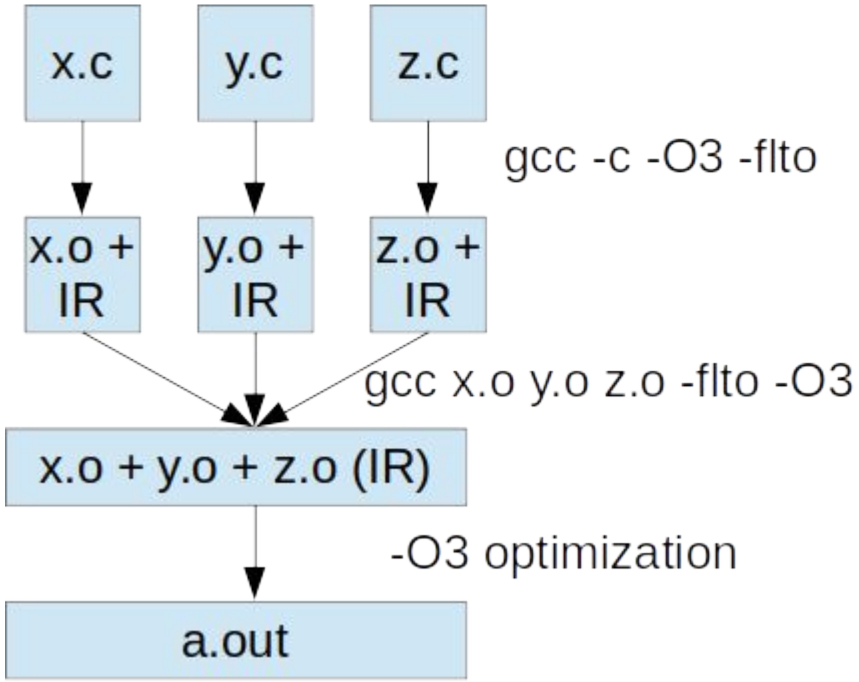
Missed optimization opportunities

- No knowledge how given code will be used by other modules causes that compiler cannot fully optimize the target application
- Motivating example: if optimizer knows what value is returned by `foo` function, then it can optimize `bar` function



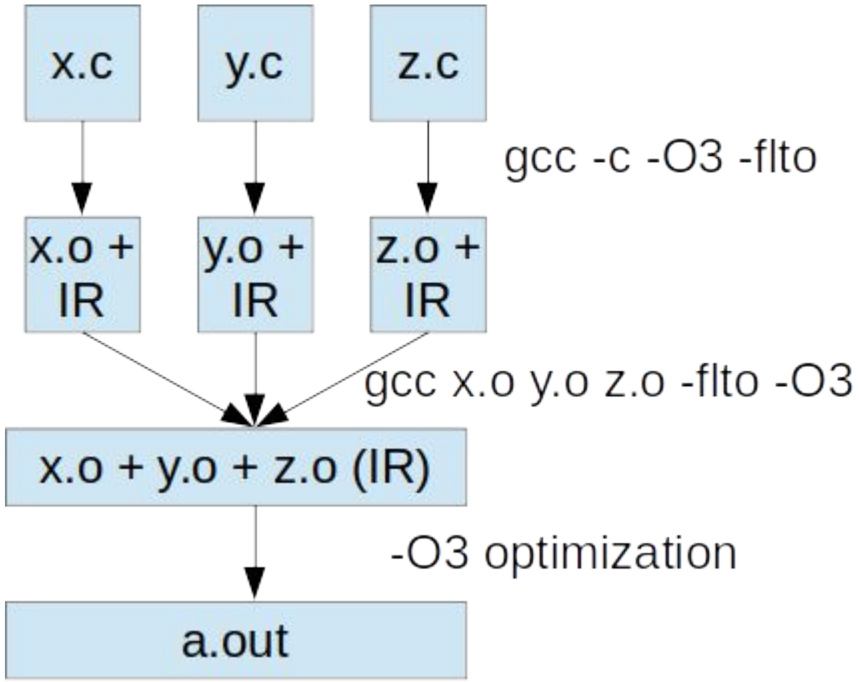
Link Time Optimization

- Extends optimization possibilities
- Partial IR code merged together (linking) and then optimized once again by standard optimization passes
- Standard analysis and optimization passes can work on the whole-program source code



Drawbacks of Link Time Optimization

- Time and resource consuming
- Prone to compiler bugs
- Modification of any source file causes that the whole optimization needs to be executed once again

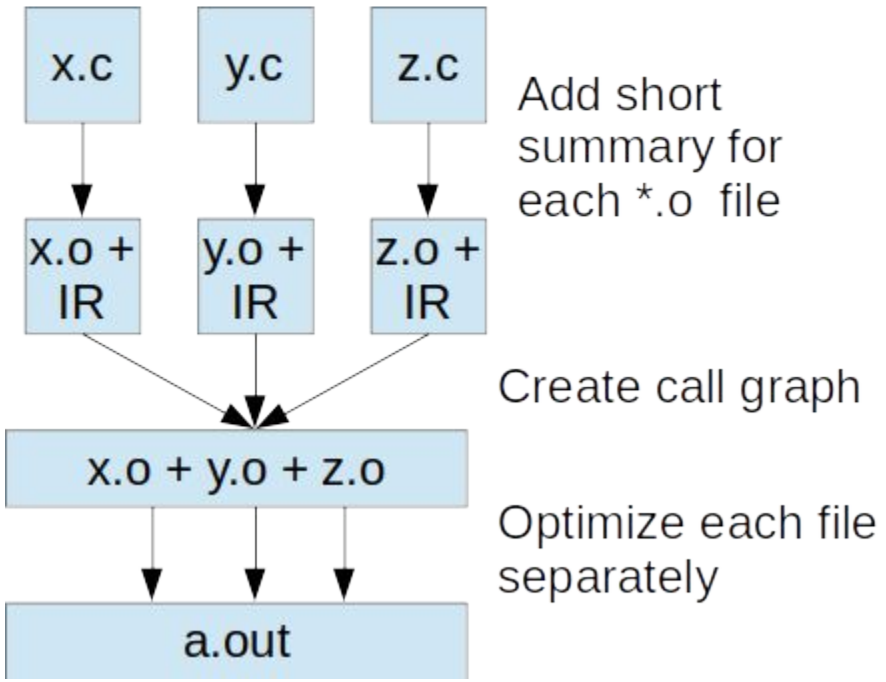


clang thin LTO

- Implemented only for clang (similar solution is done for gcc)
- Allows to parallelize link time optimization procedure
- Small modification inside one source file does not invalidate the whole previous optimization results
- Reduced memory requirements

clang thin LTO - implementation

- Every compilation module is described by small summary
- Link time analysis is done only for summaries not for the whole IR code
- Optimization is based on analysis results
- Every compilation module is optimized separately



Profile Guided Optimization – Motivation

mobica

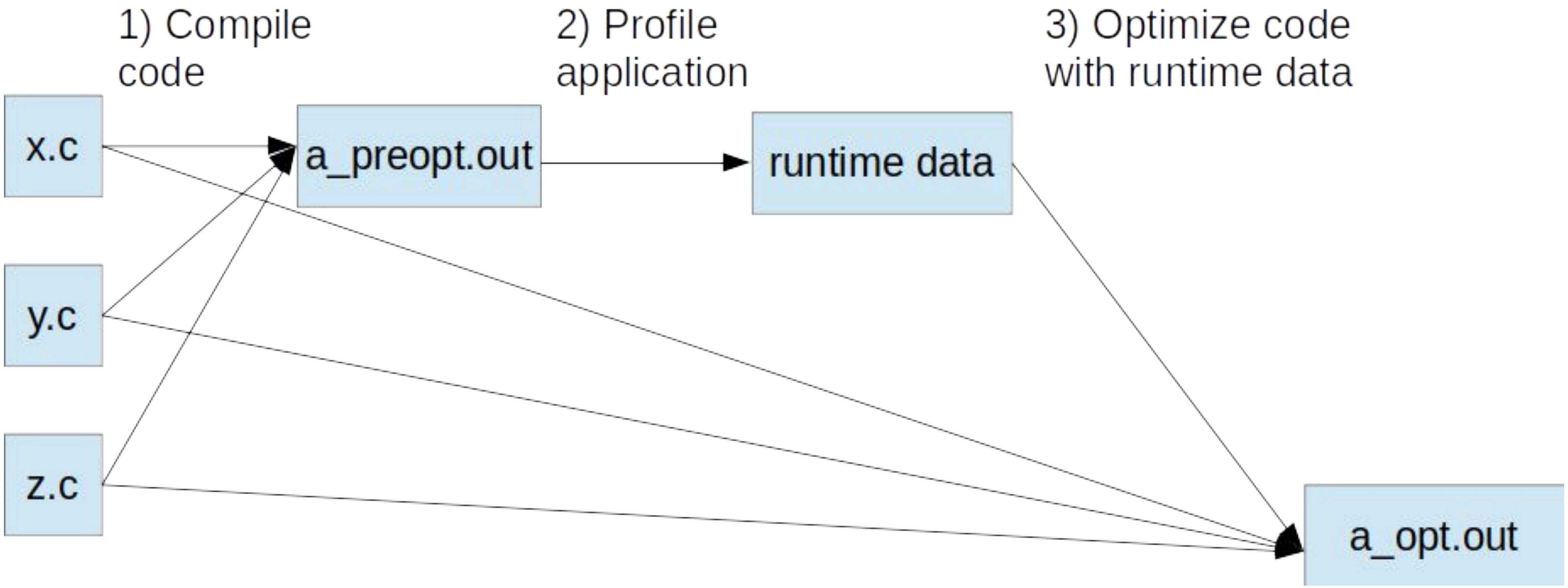
Problems to solve:

- Compiler cannot always predict in compile time which part of the code should be heavier optimized
- Trade-offs between binary size and code efficiency

Recommendations:

- Provide runtime data for compiler analysis
- Optimize for speed only “hot” parts of the code

Profile Guided Optimization – Introduction

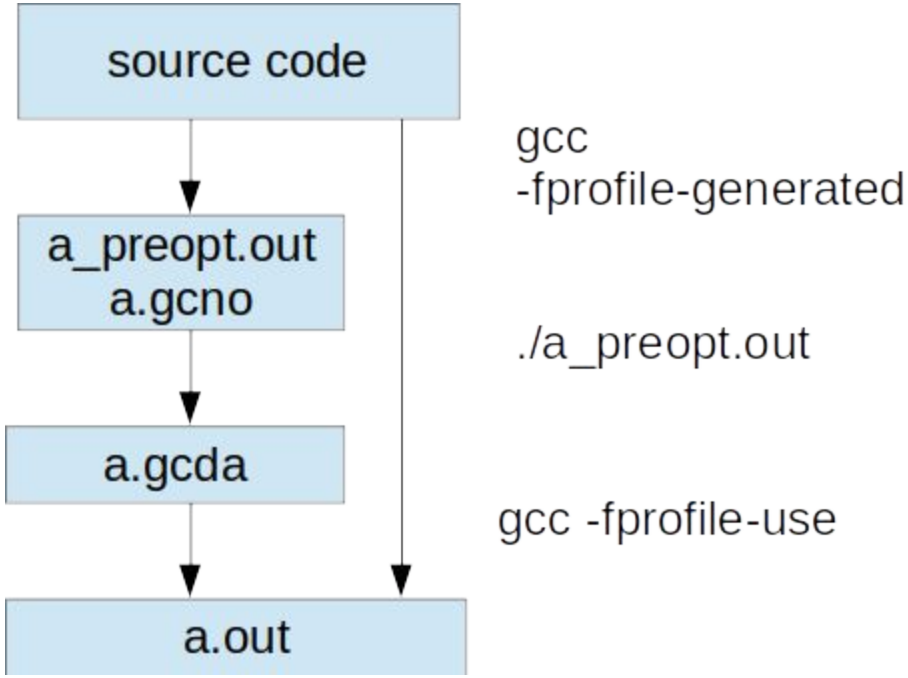


Code profiling

- The most important part of the profile guided optimization
- Code profiling is not fully transparent for building process
- Input data must reflect real usage of application
- Wrong input data can deteriorate output performance
- Time consuming
- Change in source code require new profiling data

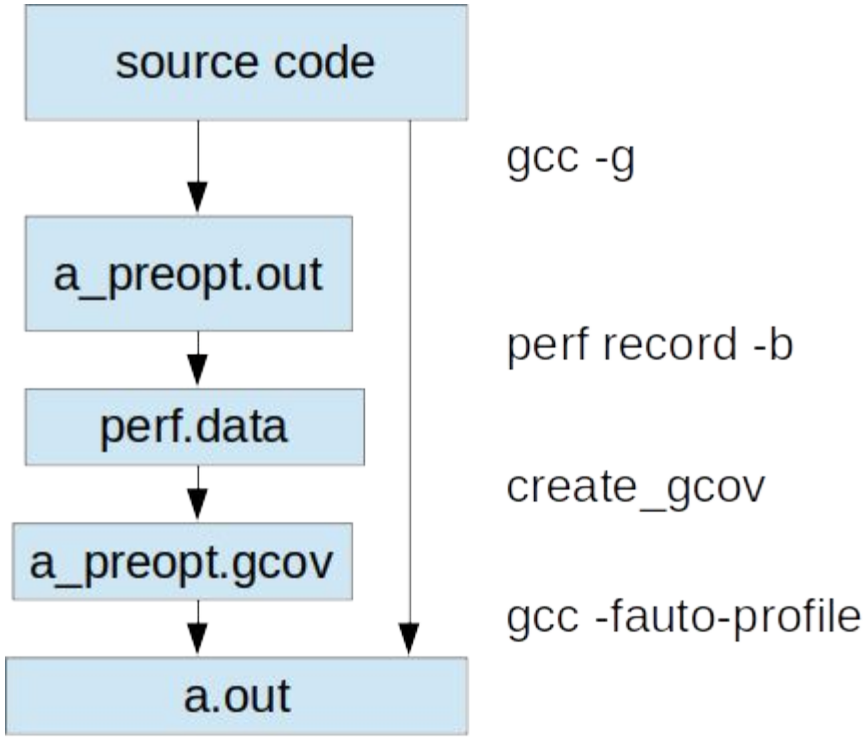
Code instrumentation

- Detailed data of code execution
- Code coverage analysis
- High runtime overhead
- May be not applicable for production code



Code sampling

- Coarse method
- Low runtime overhead
- May be executed for software in production environment
- No applicable for Linux kernel optimization
- perf used as sampling tool
- gcc and clang rely on open source AutoFDO project for converting perf data to compiler input



Effectiveness of PGO and LTO

- Combined PGO and LTO give the best results
- PGO with instrumentation can give 10-15% of improvement in comparison to standard optimisation [D. Chen, D. Xinliang Li, T. Moseley “AutoFDO: Automatic Feedback-DirectedOptimization for Warehouse-Scale Applications”, CGO’16]
- PGO with sampling can achieve 8-13% speedup [D. Chen, D. Xinliang Li, T. Moseley “AutoFDO: Automatic Feedback-DirectedOptimization for Warehouse-Scale Applications”, CGO’16]
- LTO provides averagely 10% speedup [M. Amini & T. Johnson “ThinLTO: Scalable and Incremental LTO”, 2016 LLVM Developers’ meeting]

Projects configured with LTO/PGO

- Google (internal projects)
- OpenSUSE Tumbleweed binaries are averagely 5% smaller with LTO enabled (work in progress)
- Firefox built with LTO (+ PGO for Linux)
- Gentoo, Mandriva (LTO in experimental stage)
- LibreOffice supports LTO build

Example

- Proof that PGO and LTO actually works

Recommendations

- Good software design and proper algorithmic choice are more profitable than the most advanced optimization
- Use the newest version of the compiler
- Use LTO and PGO for final release product
- Make sure that you use right input data for PGO

Few words about bibliography

- https://gcc.gnu.org/onlinedocs/gcc/ → gcc manual
- https://clang.llvm.org/docs/UsersManual.html → clang manual
- https://hubicka.blogspot.com/ → gcc developer blog
- https://llvm.org/devmtg/2016-11/#talk12 → clang thin LTO explained
- D. Chen, D. Xinliang Li, T. Moseley; Automatic Feedback-Directed Optimization for Warehouse-Scale Applications; CGO'16 → AutoFDO and sampling PGO explanation

Thank you!