

Link-Time Optimization

CodeSourcery
May 2, 2006

Introduction

Design/Requirements:

 <http://gcc.gnu.org/projects/lto/lto.pdf>

Authors:

 Mark Mitchell

 Kenny Zadeck

Participants:

 AMD

 HP

 IBM

Why Link-Time Optimization?

```
/* program.h */  
extern int i;  
extern int f();
```

```
#include "program.h"  
int i = 3;  
int f() { return 7; }
```

```
#include "program.h"  
int main () {  
    return 3 + f();  
}
```

Optimizations

- ⚡ Inlining
- ⚡ Constant propagation
 - Only in WPO mode!
- ⚡ Many more

Definitions

Inter-Module Optimization

- Optimizations that require multiple translation units
- Example:
 - Inlining of a function from one translation unit into another

Whole-Program Optimization

- Assumption:
 - All code is available to the optimizer
- Permits additional simplification
- Example:
 - If a variable is not modified in the code available, then it is never modified.

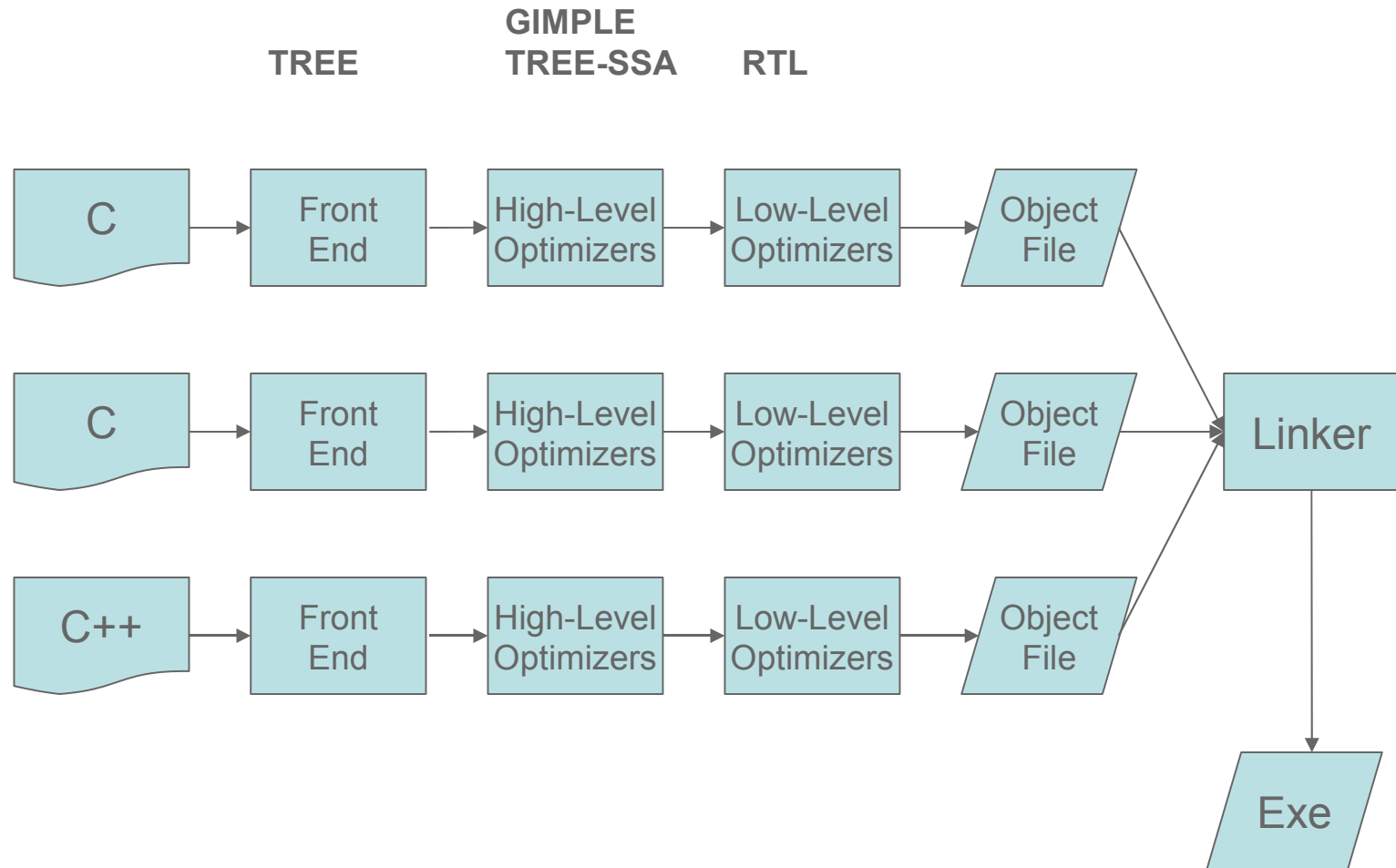
Link-Time Optimization

- IMO/WPO performed when the program is linked.

Requirements

- ⚡ Support all architectures
 - From microcontrollers to supercomputers
- ⚡ Support existing source languages
 - Including GNU extensions
- ⚡ Minimize risk-adjusted effort-to-solution
 - Avoid building things we already have
 - Avoid wholesale changes to proven infrastructure
- ⚡ Minimize risk of a “fork”
 - Avoid division of effort (KDE/GNOME)

Unoptimized Pipeline

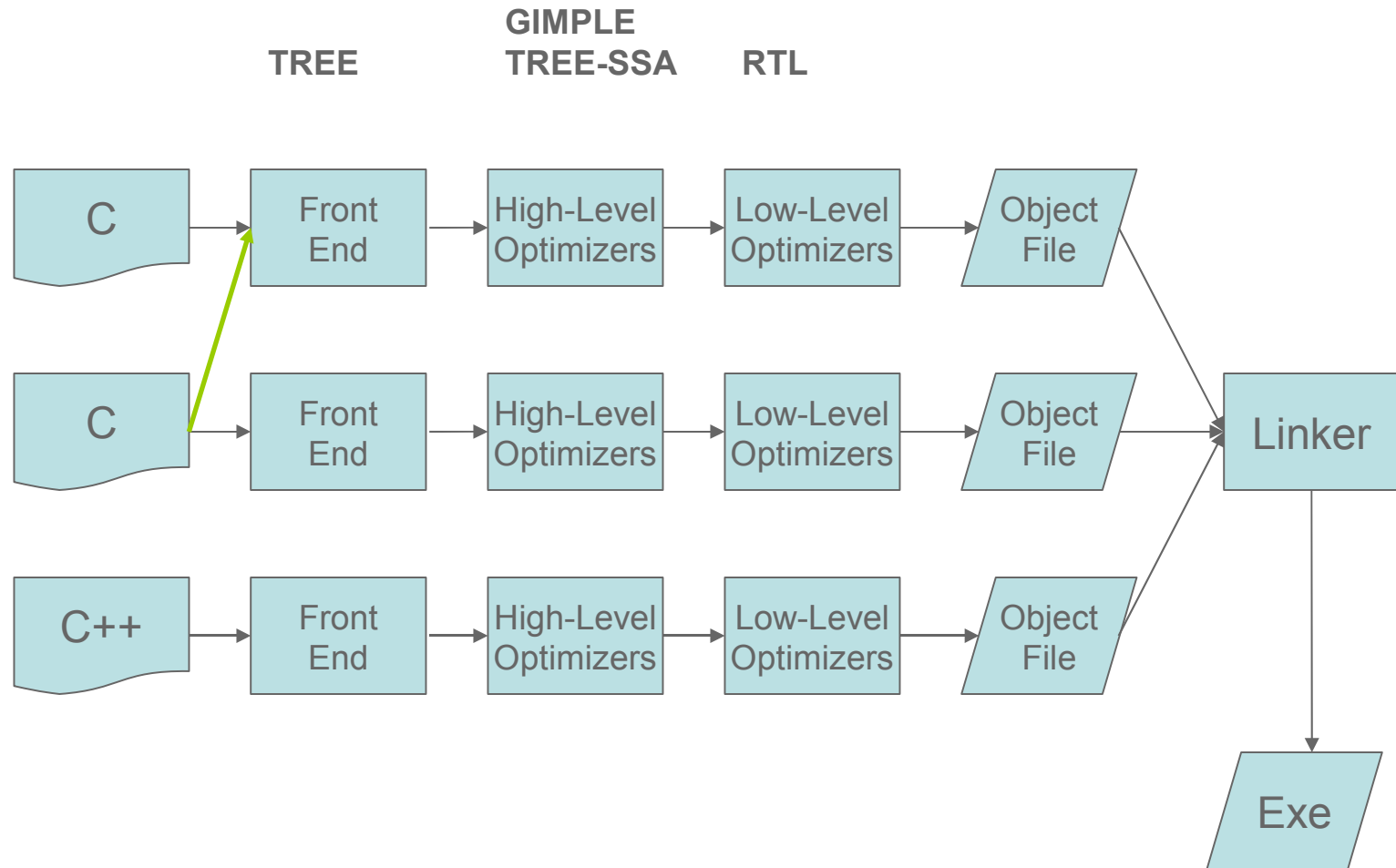


Compile-Time Approach

- ⚡ Read multiple source files at once
- ⚡ Merge translation units as files are read
- ⚡ Use existing processing pipeline to generate code

- ⚡ This approach is presently implemented in GCC – for C.

Compile-Time IMO Pipeline



CTO Tradeoffs

Strengths

- ⚡ Does not require reading/writing IL
- ⚡ Relatively easy to implement
- ⚡ Easy to give coherent diagnostics

Weaknesses

- ⚡ Does not support multiple programming languages
- ⚡ Requires all source be available when optimization is performed
- ⚡ Impacts contents of object files

LTO

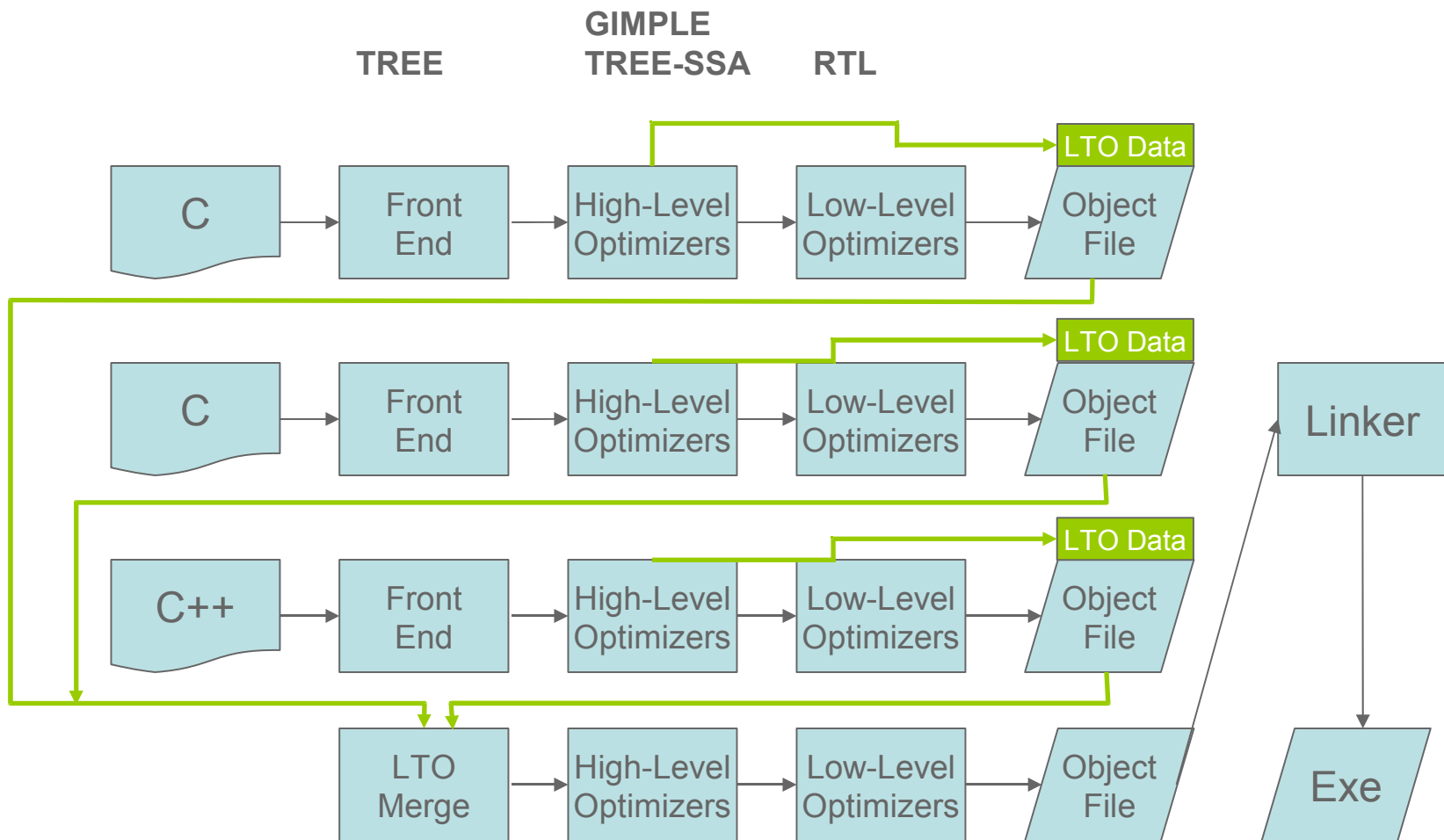
Compilation Time

- ⚡ Save program representation
 - Code: bodies of functions
 - Declarations: variables, functions, types, ...

Link Time

- ⚡ Restore saved state
- ⚡ Merge
 - Match declarations from multiple translation units
- ⚡ Optimize
 - Use existing optimizers
- ⚡ Link
 - Use existing linker

LTO Pipeline



Data Representation

Code

Bytecode Format

- Minimize storage requirements

Level of GIMPLE

- Full type information
- Array information still available
- Common format across languages

Declarations

DWARF3

- Expresses all key language features
- Extensible
- Also useful as debug information

LTO Tradeoffs

Strengths

- ⚡ Leverages existing infrastructure
 - Same optimizers
 - Same code-generators
 - Same debug generators
- ⚡ Hard parts are essential complexity
 - Serializing information
 - Merging translation units
- ⚡ Reusable infrastructure
 - IDEs
 - Link-time checking tools
 - C++ `export` keyword

Weaknesses

- ⚡ Memory usage
 - May require some “tree trimming” to support compilation of large programs ...
 - ... but that is a good thing in any event.

Conceptual Challenges

- ⚡ Language Specifications
- ⚡ When are two types or declarations the same?
- ⚡ What if the definitions do not match?
- ⚡ What if the entities come from different programming languages?
- ⚡ Should invalid combinations be diagnosed?

Problematic Examples

```
extern int i;
struct S {
    char c;
}

inline void f() {
    return 3;
}
```

```
extern long i;
struct S {
    char c;
    void g();
}

inline void f() {
    return 7;
}
```

- ⚡ Are the two is the same?
 - What if `sizeof(int) != sizeof (long)`?
- ⚡ Can a class match a struct?
 - What if they are not layout compatible?
- ⚡ Can there be two definitions of the same function?
 - Which one should be called?
 - Does it make a difference which file does the calling?

Link-Time Optimization

CodeSourcery
May 2, 2006