Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

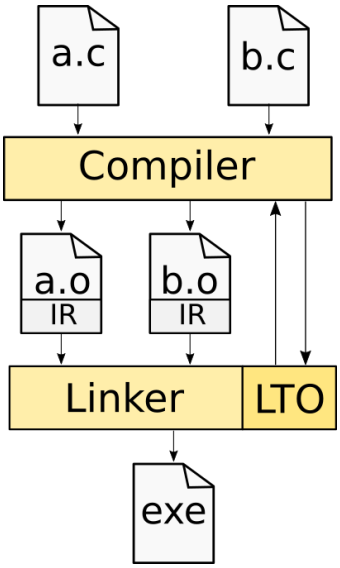
Lehrstuhl Prof. Dr.-Ing. Snelting

Implementation and Evaluation of Link Time Optimization with libFirm

Bachelorarbeit von

Mark Weinreuter

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr.-Ing. Jörg Henkel
Betreuender Mitarbeiter:	Dipl.-Inform. Manuel Mohr

Bearbeitungszeit: 3. August 2016 – 14. November 2016

Abstract

Traditionell werden Programme dateiweise kompiliert und zum Schluss zu einer ausführbaren Datei gelinkt. Optimierungen werden während des Kompilierens angewandt, allerdings nur dateiintern. Mit Hilfe von LTO erhält der Compiler eine ganzheitliche Sicht über alle involvierten Dateien. Dies erlaubt dateiübergreifende Optimierungen, wie das Inlinen von Funktionen oder das Entfernen von unbenutztem Code über Dateigrenzen hinweg. In dieser Arbeit wird libFirm und dessen C-Frontend cparser um LTO erweitert. Durch diese Erweiterung wird eine Laufzeitverbesserung von durchschnittlich 2.3% im Vergleich zur Variante ohne LTO anhand der Benchmarks der SPEC CPU2000 gemessen. Für den `vortex`-Benchmark wird sogar eine Verbesserung um bis zu 25% gemessen.

Traditionally, source files are compiled separately. As a last step these object files are linked into an executable. Optimizations are applied during compilation but only on a single file at a time. With LTO the compiler has a holistic view of all involved files. This allows for optimizations across files, such as inlining or removal of unused code across file boundaries. In this thesis libFirm and its C frontend cparser are extended with LTO functionality. By using LTO an averaged runtime improvement of 2.3% in comparison with the non LTO variant on basis of the SPEC CPU2000 is observed. For the `vortex` benchmark a speedup of up to 25% is achieved.

Contents

1 Introduction

Historically, compilers were vastly constrained in their optimizing capabilities by the available memory. E.g. in early versions of GCC there were only per statement optimizations since GCC processed one statement at a time and had no possibility of accessing anything but the current statement. With the advancement of better and cheaper hardware compilers gradually moved on to optimizing on a per function basis and finally were able to perform optimizations on a whole translation unit. Modern compilers have vast resources at their disposal and a wide range of optimizations to perform. They are able to achieve significant improvements on a per file basis.
However, programs written in common programming languages such as C or C++ usually consist of multiple translation units. The disadvantage of separately compiling parts of the program is that the compiler is unable to perform optimizations that require information about entities contained in other compilation units. If a compiler were to have access to all units at once it could potentially perform more optimizations, remove unused functions, and generally improve performance. This is achieved via link time optimization (LTO). As the name suggests, optimizations are run at link time. Usage information provided by the linker is used to perform cross module optimizations.

1.1 Motivation

Consider the example source code in listing 1, taken from [1]. It exemplifies nicely what benefits can be gained from using LTO.

There are four functions `foo1` to `foo4` defined in two different source files `a.c` and `main.c`. Function `foo3` is declared static and is therefore only visible to its containing file. The function `foo2` is externally visible but never used. Compiling this example with and without LTO produces vastly different results:

a.h

```
extern int foo1(void);
extern void foo2(void);
extern void foo4(void);
```

main.c

```
#include <stdio.h>
#include "a.h"

void foo4(void) {
  printf("Hi\n");
}

int main() {
  return foo1();
}
```

a.c

```
#include "a.h"

static int i = 0;

void foo2(void) {
  i = -1;
}

static int foo3() {
  foo4();
  return 10;
}

int foo1(void) {
  int data = 0;

  if (i < 0)
    data = foo3();

  data = data + 42;
  return data;
}
```

Listing 1: A simple example of three source files where compiling with LTO achieves more optimized results than compiling without it.

1. **Without LTO:** Compiling the input files separately will allow the compiler at best to inline and remove `foo3`. Although `foo2` is never used it cannot be removed, since the optimizer has no knowledge if this function is used in other files. This means that the value of `i` could change and thus `foo1` cannot be simplified.

2. **With LTO:** Compiling these files with LTO allows for a range of optimizations. All compilation units are known and since there is no call to `foo2` it is removed. This in turn guarantees that the value of the variable `i` is never changed. Which makes it impossible for `foo3` to be called, since the condition `i < 0` in `foo1` is never true.
   Therefore `foo3` can be removed. This triggers the removal of `foo4` because the only function call to `foo4` is in `foo3`. This greatly simplifies `foo1`. Functions `foo2`, `foo3`, `foo4` can be removed and the simplified version of `foo1` is inlined into `main` and removed. These optimizations are only possible because the optimizer has detailed usage information from all involved compilation units.

```
  .text
# -- Begin  main
  .p2align 4,,15
  .globl main
  .type  main, @function
main:
  .p2align 4,,10
  movl $42, %eax
  ret
  .size  main, .-main
# -- End  main
```

Listing 2: Resulting assembly code when compiling the example code in listing 1 with LTO.

The resulting assembly code can be seen in listing 2. All functions have been completely removed and `main` simply returns `42`.

Compiling with LTO opens up new possibilities for optimizations as the example in listing 1 shows. The optimizer is no longer restricted by its limited knowledge of the compilation units involved. Instead it has detailed usage information which enables it to remove unused entities or inline functions across modules. The benefit of LTO is that there is no need to develop new compiler optimizations. The existing optimizations can be used as before. They merely operate on a larger code base.

2 Fundamentals and related work

In this chapter basic concepts of a usual compilation workflow are revisited. This allows for a better comparison of the differences in compiling with and without LTO. Furthermore, in order to extend a compiler with LTO support the underlying compiler fundamentals need to be considered.

2.1 Traditional compilation process

Suppose we are given a project that consists of multiple source files. In figure 2.1 the traditional compilation process is shown. Each file is compiled separately into assembly code and then assembled into an object code file. Once all files are compiled they are linked into an executable. The advantage of this approach is that each file is a self contained unit and compilation of multiple files can be done in parallel.

2.2 LTO support in other compilers

In order not to interrupt the traditional, well established compilation process, one of the main objectives when adding LTO support was the requirement to be compatible with existing toolchains. E.g. compiling source files to object files and giving these to the linker to create an executable should still be possible in exactly the same way as before. The only alteration to activate LTO is the `-flto` flag, which tells the compiler and linker to produce and expect LTO-compatible output.

The process of compiling with LTO is illustrated in figure 2.2. The compiler generates special object files which contain its intermediate representation of the

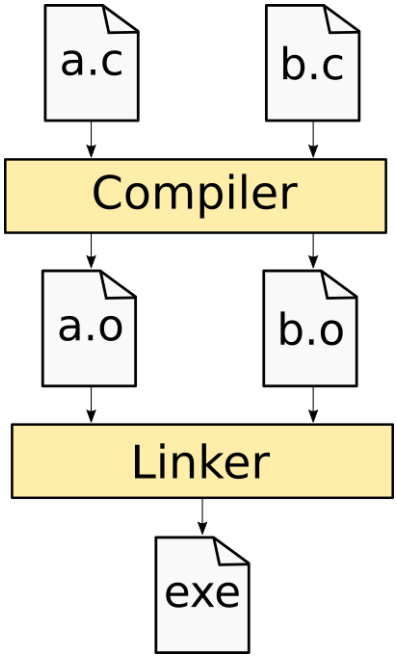


Figure 2.1: Traditional compilation flow. Each source file is compiled into an object file. These are then linked into an executable.

program. In order to deal with these object files the linker is usually extended via a plugin. It interfaces with the compiler to provide it with information about the files to be linked. The linker is aware of all involved files, can resolve the used symbols, which in turn allows for the whole code base to be optimized. Finally, the result of the optimization steps are linked into an executable.

Two options exist how to generate these special object files. The first option is to output regular object code alongside the compiler's native representation. This is known as fat LTO because such files contain both sets of information, which usually doubles the compilation time and file size. The advantage of this format is that it works with LTO compatible tools and those unaware of LTO yet. Alternatively, slim object files can be used, which only contain the compiler's intermediate representation. These rely on the linker and other tools to be able to handle object files which contain intermediate language.

As an example, LLVM [2] has its intermediate language: LLVM bitcode. LTO support is tightly integrated into LLVM's linker. It can deal with regular object files, LLVM bitcode or a mix thereof. The LLVM tools are accessed via `libLTO` to provide the linker access to the optimizer without having to deal with LLVM internals.

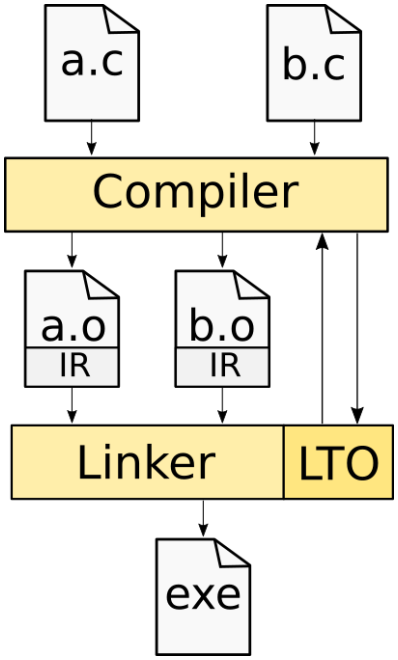


Figure 2.2: LTO implementation using a linker plugin. The compiler generates object files, which contain the compiler's intermediate representation of the source code. The linker detects this and interfaces with the compiler to optimize and link these files as a whole.

2.3 libFirm

Extending a compiler to support LTO requires knowledge of its underlying infrastructure. In this thesis libFirm [3], which is a C library implementation of firm [4] providing a low level graph based intermediate representation of computer programs, is extended. It is developed at the Institute for Program Structures and Data Organisation (IPD) at Karlsruhe Institute of Technology (KIT). libFirm itself provides only a library with which a compiler can be built. In order to use it a frontend is required. Currently, there is a frontend for Java, X10 [5], and C code. For the purposes of this thesis the C frontend cparser is relevant.

The main feature of firm is that it is completely graph based. This means that firm doesn't use instruction lists to hold the compiled source code as other compilers do. Instead, the code is represented in the form of data flow and control flow graphs. This graph representation is in SSA form [6] upon which optimization transforms can be applied to allow the generation of optimized machine code. An

example of a graph representation for a simple function can be seen in figure 2.3. The graph consists of nodes which represent the content of the `add` function. These nodes are connected and form a hierarchy. However, there is no total order, e.g. the operands of the `Add` node are not ordered. A more detailed explanation of the various nodes and their dependencies are beyond the scope of this thesis, but can be found in [7].

```
int add(int x, int y)
{
  return x + y;
}
```

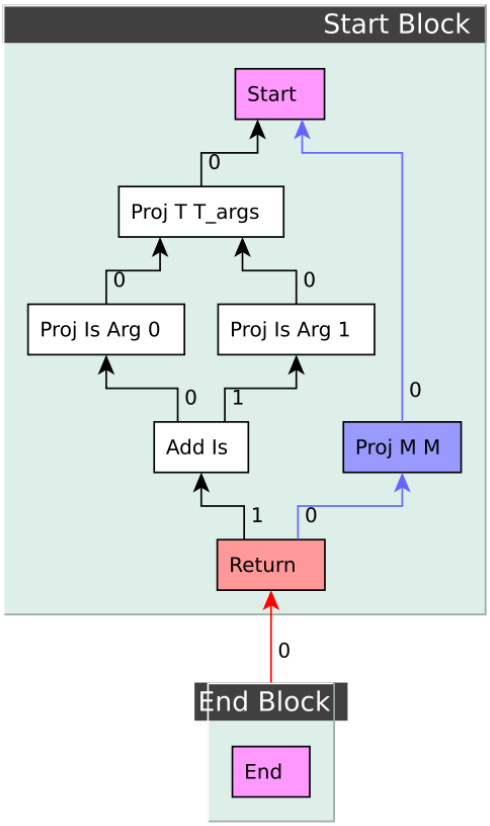


Figure 2.3: The add function on the left is represented as a FIRM graph on the right.

The CPARSER frontend is deeply tied to LIBFIRM and controls the compilation flow responsible for calling upon LIBFIRM internals. It follows the traditional compilation pipeline of reading C files, parsing the source code and construction of the abstract syntax tree (AST). The AST is transformed into an intermediate representation. This representation is constructed by LIBFIRM which holds the information of the provided source code in form of an intermediate representation

program (IRP). An IRP is the central structure which essentially consists of three parts:

- **A list of IRGs:** Each function in the original source code is transformed into an intermediate representation graph (IRG). An IRG is a directed graph which represents the underlying code as nodes and connecting edges.

- **A typegraph:** Each function or variable has a type. Such a type denotes the size, alignment and additional attributes of every function or variable with this type. There are primitive types which represent numbers but also more complex compound types. For example struct types are compounded of other types including other compound types. The types in total make up the typegraph.

- **The symbol table:** Every function or variable is called an entity. These entities are recorded in a symbol table. Entities have a type and furthermore specify information about their visibility within the source code and other attributes such as volatility and linkage.

2.4 IR-files

A central part of this LTO implementation is the usage of source files in an intermediate representation format (IR-files). When a C source file is parsed by the cparser frontend and passed to the libFirm backend it is converted into an intermediate representation program (IRP). This IRP can be exported as an IR-file. This file is an exact, textual representation of the IRP. Hence, it can be imported by cparser to populate the IRP from a file instead of generating it from a C source file. Since these IR-files are essential, their basic structure and relevant parts are outlined in listing 3.

2.4.1 IR-file markup

Every IR-file consists of multiple sections. Most relevant for this implementation is the `typegraph` section. It contains information about the entities and types within the represented IRP. The format of the `typegraph` section is described in the form

```c
#include <stdio.h>

static int foo = 21;

void bar(){
  printf("Value:
   ↪  %d\n", foo *
   ↪  2);
}

int main(){
  bar();
  return 0;
}
```

```
typegraph {
    ...
    entity 107 "foo" "foo" local ... initializer
     ↪  IR_INITIALIZER_CONST 112
    method 109 "bar" "bar" external [constant ] 108
     ↪  22 ...
    method 111 "main" "main" external [constant ]
     ↪  110 22 ...
}
irg 111 140 {
    Anchor 141 [142 144 143 146 147 149 148]
    ...
}
constirg 31 {
    Const 112 "Is" 15
    ...
}
```

Listing 3: A C code example on the left and an excerpt of the corresponding IR-file.

of a grammar as can be seen in grammar 2.3. This grammar uses the entity and type definitions from grammar 2.1 and grammar 2.2.

2.4.2 Entities in IR-files

In listing 3 a simple C code example alongside the basic structure of an IR-file is displayed. There are three entities contained in the source code: the variable `foo` and the functions `bar` and `main`. Within the `typegraph` section of the IR-file there is a definition for each entity as outlined in listing 4. Every line describes an

```
entity 107 "foo" "foo" local [] 50 22 volatility_non_volatile initializer
 ↪  IR_INITIALIZER_CONST 112
method 109 "bar" "bar" external [constant ] 108 22 volatility_non_volatile 0
method 111 "main" "main" external [constant ] 110 22 volatility_non_volatile
 ↪  0
```

Listing 4: An extract of the entity definitions in the IR-file.

entity and is constructed according to the rules in grammar 2.1. First the entity

⟨*entity-def*⟩ ::= ⟨*entity-type*⟩ ⟨num_{id}⟩ ⟨$ident_{name}$⟩ ⟨$ident_{lname}$⟩ ⟨*visibility*⟩ '[' ⟨*linkage-opts*⟩ ']' ⟨num_{type}⟩ ⟨num_{owner}⟩ ⟨*volatility*⟩ ⟨*attributes*⟩

⟨*entity-type*⟩ ::= 'entity' | 'method' | 'alias' | 'label' | 'compound_member' | 'parameter' | 'unknown'

⟨*visibility*⟩ ::= 'external' | 'external_private' | 'external_protected' | 'local' | 'private' |

⟨*volatility*⟩ ::= 'volatility_non_volatile' | 'volatility_is_volatile'

⟨*linkage-opts*⟩ ::= ⟨*linkage-opt*⟩ ⟨*linkage-opts*⟩ | ε

⟨*linkage-opt*⟩ ::= 'constant' | 'weak' | 'garbage_collect' | 'merge' | 'hidden_user'

⟨*attributes*⟩ ::= ⟨*attribute*⟩ ⟨*attributes*⟩ | ε

Grammar 2.1: Simplified entity grammar. The identifiers and numbers have been given a name to better illustrate their semantic meaning.

kind is specified, followed by a unique numerical identifier, the name, linker name, visibility, linkage options, volatility, type id and owner id. Optionally, as in the case of the variable `foo`, an initializer is specified. Since `foo` is declared static its visibility results to `local`. Whereas the other two methods have external visibility and are reported as such in the IR-file.

2.4.3 Types in IR-files

Furthermore, the `typegraph`-section also contains the information about the types of these entities. In accordance with grammar 2.2, a type has a unique identifier, specifies which kind of entity it represents and holds information about its size and

⟨*type-def*⟩ ::= '`type`' ⟨num_{id}⟩ ⟨*opcode*⟩ ⟨num_{size}⟩ ⟨num_{align}⟩ ⟨*layout*⟩ ⟨num_{flags}⟩ ⟨*attributes*⟩

⟨*opcode*⟩ ::= '`struct`' | '`union`' | '`class`' | '`segment`' | '`method`' | '`array`' | '`pointer`' | '`primitive`' | '`code`' | '`unknown`'

⟨*layout*⟩ ::= '`layout_fixed`' | '`layout_undefined`'

⟨*attributes*⟩ ::= ⟨*attribute*⟩ ⟨*attributes*⟩ | ε

Grammar 2.2: Simplified type grammar. Type attributes are dependent on the types opcode and the available attributes have not been reported.

⟨*typegraph-def*⟩ ::= '`typegraph{`' ⟨*tg-content*⟩ '`}`'

⟨*tg-content*⟩ ::= ⟨*type-def*⟩ ⟨*tg-content*⟩ | ⟨*entity-def*⟩ ⟨*tg-content*⟩ | ⟨*empty*⟩

Grammar 2.3: The resulting typegraph grammar, which uses the previously defined `entity-def` in grammar 2.1 and `type-def` in grammar 2.2.

```
type 50 primitive 4 4 layout_fixed 4 "Is"
type 108 method 4 1 layout_fixed 4 0 0 0 0 0
type 110 method 4 1 layout_fixed 4 0 0 0 1 0 50
```

Listing 5: Type definitions for a signed integer and two method types.

alignment. Additionally, depending on the entity kind, further information such as the amount of parameters, return values and their types are reported.

For example, the method `main` in listing 4 has the type `110`, which corresponds to the third type definition in listing 5. This entry describes a method type. The last four numbers `0 1 0 50` specify that a function with this type has zero parameters, one return value, and doesn't have variadic arguments. The type of the return value is denoted as `50`. This matches the type definition in the first line which describes a primitive type for a signed integer. The definition for the `main` method within the IR-file precisely represents the function declared in the C source code.

```
irg 111 140 {
Anchor 141 [142 144 143 146 147 149 148 145 ]
Block 142 [155 ]
Block 144 []
End 143 []
Start 146
Proj 147 146 "P" 1
Proj 149 146 "M" 0
Proj 148 146 "T" 2
NoMem 145
Address 151 109
Call 152 144 149 151 108 op_pin_state_pinned nothrow []
Proj 153 152 "M" 0
Const 154 "Is" 0
Return 155 144 153 [154 ]
}
```

Listing 6: Textual representation of the IRG of the `main` function

2.4.4 Functions as IRGs

However, only the function declaration is specified in the `typegraph`-section. Actual code contained within the `main` function is encapsulated into an IRG. Every function that is defined in the input source file has been converted into an IRG and is represented as such in the IR-file. In listing 6 the IRG of the `main` function is shown. The IRG references the method entity it belongs to, followed by the nodes that make up this IRG. In this example the id `111` refers to the `main` function which matches the entity definition in listing 4.

3 Design and implementation

As outlined in section 2.2 the optimal solution to enable LTO support would be a linker plugin. LibFirm already has an intermediate format, which can be used for LTO. The linker would thus need to detect IR-files and interface with libFirm to compile and optimize the intermediate code. However, creating a linker plugin is a time consuming task and beyond the scope of this thesis. Therefore a simpler but more invasive solution is presented here.

The cparser frontend can generate and read IR-files. This is the base of this LTO implementation. Multiple IR-files are read by cparser. These are merged within libFirm without being passed to the linker. This effectively means that linking is performed on IR-files by the LTO extension instead of object files as is done by the linker. This has the disadvantage that the traditional build process has to be altered. Source files are no longer compiled to object files and linked. Instead, as figure 3.1 illustrates, each source file needs to be compiled to an IR-file. These IR-files are passed to cparser, which in turn constructs a single object file. Finally, this object file is linked into an executable.
The LTO implementation in this thesis provides the core LTO functionality upon which a linker plugin can be built.

3.1 Focus

As previously described, this LTO implementation relies on IR-files as its input format and extends cparser's ability of dealing with these IR-files. Previously, only the following options were available for processing IR-files:

- **Compiling a source file to an IR-file:** A single C source file can easily be compiled into an IR-file with the following command:
  `cparser a.c --export-ir -o a.ir`

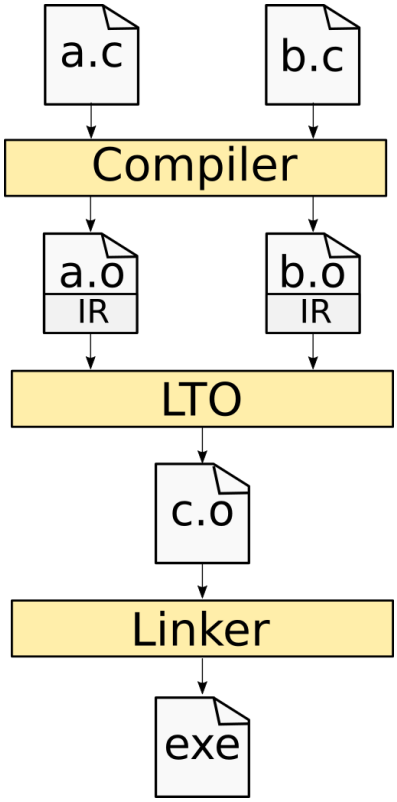


Figure 3.1: Compilation flow of the LTO implementation in this thesis. Each source file is compiled into an IR-file. These IR-files are merged, compiled into a single object file and linked into an executable.

- **Compiling an IR-file:** An IR-file can be treated as any other source file, it can for example be compiled into object code:
  `cparser a.ir -c -o a.o`

These options operate on a single input file. However, LTO requires information about all input files at once to be able to perform cross module optimizations. Consequently, these options have been added:

- **Inputting multiple IR-files:** Multiple IR-files can be given to CPARSER which consecutively reads each file and extends the IRP. Since there is only a single IRP all the previously available CPARSER compile options are still accessible. For instance multiple IR-files can be compiled into an executable named *test*:
  `cparser a.ir b.ir -o test`

- **Merging IR-files:** With the ability to read multiple IR-files there is of course the possibility to export the merged IRP to an IR-file once more. This allows the generation of a single IR output file:
  `cparser a.ir b.ir --export-ir -o merged.ir`

- **Export optimized IR-files:** Platform independent optimizations can also be performed on IR-files. Using this option one can merge and optimize IR-files, exporting an IR-file again:
  `cparser a.ir b.ir -O3 --export-ir optimized.ir`

The focus of this LTO implementation is therefore to extend the usage of IR-files in so far as to make it an independent input format with the added ability to perform cross module optimizations.

3.2 Required adaptions

3.2.1 Multiple input files

In order to handle multiple compilation units the CPARSER frontend has to be modified. Every C source file passed to CPARSER is converted into an IRP. If more than one file were to be loaded, an IRP would be constructed anew for every file. This is currently not supported and there are safeguards to prevent inputting multiple source files. LIBFIRM is designed to process a single compilation unit and has only one IRP at a time. It would therefore be tedious and unforeseeably complex to alter LIBFIRM to construct multiple IRPs. However, IR-files are already in an intermediate format and can be directly integrated into the existing IRP. Consequently, the chosen way of loading more than one input file is to first compile each source file into an IR-file separately. Once all files are translated into IR-files they are passed to the CPARSER frontend, integrated into the same IRP and then compiled into a single executable.

3.2.2 Compilation pipeline

When a C source file is passed to CPARSER for compilation a set of steps is executed to generate the final executable. First the input file is read and preprocessed, the

content is parsed and an abstract syntax tree (AST) is generated. The AST is translated into the intermediate representation and the IRP is constructed. From here on the IRP is optimized and assembly code is generated. Finally, the assembler produces object code from the assembly code. The steps of this process depend on the specified input files and the desired output format. If an IR-file is given, it will be read and an IRP is directly generated without prior parsing or AST generation. Similarly, the code generation steps are not executed if the `--export-ir` flag is set to indicate that an IR-file is the desired output format.
To be able to input multiple IR-files, CPARSER is modified to put the optimization and code generation steps on hold until all files are read and the IRP contains the information from all files. After that the compilation is resumed and the remaining steps depending on the specified flags are executed.

3.2.3 Private entities

One of the first things to consider when combining compilation units are private entities. In C a file scope function or variable marked with `static` is visible only within its containing file as defined in the C standard [8, section §6.2.2]. Therefore it is possible that an entity by the same name marked as `static` exists in two compilation units. If these two units were integrated into the same IRP a conflict would arise since entity names must be unique. To prevent this from happening all static entities get renamed by default. Renaming takes place during the processing of the input IR-file. In this phase entities are referenced by a unique file id and thus renaming can be easily and safely accomplished.
To ensure unique names a global counter value is used. The name of each entity is prefixed by a unique value, e.g `r.<counter value>.<name>`. Since the counter is incremented with each private entity and a dot is not permitted in variable names but allowed for assembly names the name is guaranteed to be unique.

3.2.4 Entity types

Each entity has a type. This type denotes the size, alignment and additional attributes of every entity with this type. Types are not unique, meaning there can be multiple types with the same properties. E.g. two source files include the same header file which has a struct definition. When these two files are parsed and exported into an IR-file both have a type entry for this struct. Usually each file is read separately, compiled and the duplicate types therefore required. However,

with LTO both files are combined and a single definition will be sufficient.
Dealing with this issue, two possibilities exist. Either allow duplicate type definitions or compare types and merge these with the same properties. A type for a primitive, such as an integer, is as the name suggests easy to construct and compare. A compound struct type is more complex and potentially nested. It can contain other nested types. Comparing two struct types therefore requires comparison of all types within this type. Since LIBFIRM internally does not rely on unique types but rather operates on their identifying attributes, it is not necessary for correct operation to merge all types. Hence, the chosen approach is to allow duplicate types.

3.2.5 Entities

Dealing with entities proves to be more difficult. Entities have unique names and there can be multiple declarations across various files and exactly one definition. Take the code in listing 7 for example. Both files have either a declaration or

```c
int bar();

int main(){
    return bar(42);
}
```

```c
int bar(int i);

int bar(int i){
    return i;
}
```

Listing 7: Two source files: *decl.c* on the left and *def.c* on the right which both specify a function `bar`

a definition of the function `bar`. However, in the first file there is a declaration which specifies no parameters meaning the function can have an unspecified number of arguments. This is allowed behavior as defined in the C standard [8, section §6.5.2.2] and merely results in a warning that the function `bar` is not a prototype, but is accepted by the compiler. The definition specifies the correct amount of parameters. This results in different function type definitions if these two files are exported as IR-files. An excerpt of the generated IR-files can be seen in listing 8

The left excerpt corresponds to the function declaration. Here `bar` is defined to have no arguments and a return value. In the right excerpt the function type specifies a return value and one parameter. Compiling these two files separately and then linking them poses no problems. The conflict arises at link time and the linker doesn't check function types. When compiling with LTO things are different.

```
type 45 method 4 1 layout_fixed 4 0
 ↪  0 0 1 0 46
method 47 "bar" "bar" external
 ↪  [constant ] 45 22
 ↪  volatility_non_volatile 0
```

```
type 45 method 4 1 layout_fixed 4 0
 ↪  0 1 1 0 46 46
method 47 "bar" "bar" external
 ↪  [constant ] 45 22
 ↪  volatility_non_volatile 0
```

Listing 8: Two IR-files which both have a declaration or definition of the function `bar`

Here part of the linking process is done in the compiler. If the incorrect definition were used, libFirm would raise an exception since the method invocation with one argument would not correspond to the amount of parameter specified in the function type.

The example above illustrates why it is important to distinguish between the declaration and definition of functions. The desired behavior during the merge is to keep the correct definition and ignore any incorrect declarations. However, when an IR-file is read it is not apparent if the given entity is a definition or a declaration. This information is not part of the IR-file. In contrast, the LTO implementation of LLVM for example has builtin functionality to check if an entity is a declaration or a definition. Extending the IR format to provide this information would simplify the merging process, but is beyond the scope of this implementation. To circumvent this issue some simple heuristics have to be used.

- Dealing with normal entities, such as global variables there will be exactly one entity with an initializer. This entity will be treated as the definition during a merge and its type and initializer is kept in the resulting entity.

- Functions pose a bigger problem. Additionally to non prototype function declarations, a compilation unit might contain an implicitly declared function as can be seen in listing 9. During compilation, the compiler issues the warning: „`implicit declaration of function 'foo'`“, but the code can be successfully compiled and linked. If the compiler detects an implicit declaration a simple placeholder is included in the IR-file. This substitute specifies no parameters and an integer return type. It most likely doesn't match the actual type of the function definition.
  Merging is achieved by comparing parameters and return types, giving precedence to types that do not represent a placeholder type.

main.c

```c
int main(){
    return foo();
}
```

foo.c

```c
int foo(){
    return 42;
}
```

Listing 9: The function `foo` is used but not declared in the file `main.c`. The compiler issues a warning during compilation but both files can be successfully compiled and linked.

utils.h

```c
extern void *safe_malloc(
long bytes);
```

okmalloc.c

```c
char *safe_malloc(unsigned size);
{  /* ... */ }
```

Listing 10: Definition of `safe_malloc` and an incorrect declaration thereof.

Finally, entities could specify different visibilities, volatilities and linkage options. Linkage options are simply accumulated, whereas in the case of volatility and visibility the more restrictive one is chosen where possible. E.g. if an entity is once specified as volatile and as non volatile in another definition it is treated as volatile.

3.2.6 Common problems

Since this LTO implementation links all involved compilation units within libFirm some unexpected errors occur that usually arise in the linker. These problems are caused due to two things:

- **Contradictory method definitions and declarations:** For example the method `safe_malloc` is defined to expect an unsigned value. However, a declaration in a separate header file specifies long values as can be seen in listing 10.
  Compiling and linking these files in the traditional manner doesn't raise any errors. This is due to the fact that each source file is separately compiled into an object file and when the linker is invoked there are no more type checks. However, the LTO variant merges the files beforehand and libFirm has a verifier which checks that correct parameter types are specified and thus aborts with the following message:

```
Verify warning: Proj Iu[67331:21](safe_malloc[67318]):
expected mode Is but found Iu.
```

- **Mismatching declarations:** The function `bc_expand_increment` is declared returning a non void value while being defined as returning void. This causes libFirm to raise a similar error, terminating the compilation process:
  ```
  Verify warning: Return X[1..1](bc_expand_increment[1..3]):
  number of inputs does not match method type (0 inputs,
  1 declared)
  ```

Problems like these usually do not surface since the code runs just fine and these mistakes are at worst simple coding errors. Only due to libFirm's verifier and the merging behavior of LTO do these issues show up. To resolve these problems the source code of the involved function declarations are adjusted.

3.2.7 Removal of unused entities

During a regular linking phase the linker resolves the symbols of all involved entities. The linker therefore has complete knowledge which entities are used and whether they are referenced outside of their containing file. A linker plugin can communicate this information to the compiler which can remove unused entities or change their visibility where possible. Since this LTO implementation is directly integrated into libFirm this detailed information is not readily available.

To simulate this behavior a simplifying assumption is made: All compilation units have been merged into a single IRP, which means there is no compilation unit left to externally reference any of these entities. Thus visibilities can be adjusted to the effect that all entities now have file local visibility. An exception has to be made for the main function, since it will be called externally.
This is a rather crude assumption. If the compile target were a library, visibility adjustment would not be possible since the library functions will be referenced externally. A solution to this problem would be to allow specifying which methods to adjust and those to be kept. However, due to the additional implementation overhead a simpler solution is chosen. The main function is never adjusted and other functions are only altered if the compile target is not a library.

4 Evaluation

In this chapter we analyse the benefits and drawbacks of this LTO implementation. Compiling with LTO means the compiler has to process more data and hold it in memory. An obvious initial assumption is that the compile time and memory usage will increase. Yet, the additional information can, e.g. be used to inline more methods and to enable further improvements. Inlining more functions can potentially increase the code size and thus in turn the amount of nodes in its IR graph and the overall memory requirements. However, the additional possibilities for optimization should have a positive effect on the runtime of the compiled binary.

4.1 Benchmarking tools

4.1.1 SPEC CPU2000

The programs used in this evaluation are taken from the Standard Performance Evaluation Corporation (SPEC) CPU 2000 [9]. This suite contains a range of standardized programs in order to produce comparable benchmarking results. Benchmarks in different programming languages are available, but only those written in the C programming language are considered here.
The compile issues described in section 3.2.6 arise with the `gcc` and `twolf` benchmarks. The two affected files are modified to correct the function declarations which resolves the problem.

4.1.2 Benchmarking tool: temci

Another useful benchmarking tool is temci [10]. It is developed with the intention to produce reliable results and provides a statistical analysis of these findings. Futhermore, temci has builtin support to work with the SPEC suite and thus the combination of these two tools is used to produce reproducible and comparable results. Additionally, temci provides randomization support [11], to scramble the assembly code every time the program is compiled to randomize the order in which functions are placed in the resulting executable. This minimizes possible caching effects to ensure statistically relevant results. Unfortunately, this option requires recompilation of each benchmark for each run and is disabled to avoid the additional time overhead.

4.1.3 GNU time

Interesting compile metrics are the time required to build each benchmark and amount of main memory (RAM) used during compilation. The GNU time [12] program can report on both of these properties. By simply running the compile command through GNU time, it records among other things the elapsed wall clock time and the maximum resident set size (RSS). This RSS denotes the amount of main memory (RAM) that is currently held by a process excluding swapped out memory or unloaded parts. As long as no swapping occurs the RSS is a valid indicator for the amount of memory used.

4.1.4 Inlining

The optimizer has a wide range of optimizations but the inline optimization is expected to gain the most. Without LTO only functions within the same file can be inlined since it is impossible to know if a function is called from another file. With LTO enabled the compiler has the required information and can inline across files. When a function call is inlined, the call is removed and the function content is inserted instead. If all function calls are removed the function can be discarded as well. This effectively results in fewer functions but possibly larger code size. Since libFirm is graph based the size change can be measured as an increase in node count. A function call can only be inlined if the amount of nodes of the

containing function plus the node count of the function to be inlined is less than the current inlining limit. To adjust this limit the `-finline-max-size` flag with the desired limit is passed to CPARSER. Since this has a significant influence on the optimization performance the benchmarks are conducted for various inline limits: 375, 750, 1500, 3000, 6000. The default value is 750.

4.1.5 Benchmarking setup

The benchmarks are conducted on a desktop computer with an Intel Core i7-2600 3.4GHz CPU and 16GB RAM, running a Ubuntu 16.04 with a 4.4.0-45-generic kernel. All of the SPEC benchmarks are compiled for each of the specified inlining limits, once for the LTO variant and once for the non LTO version. All these executables are then run using temci to handle the SPEC benchmarking process. Every benchmark is conducted 20 times. We perform the following measurements:

- **Compile time:** The compile time for every benchmark is reported to distinguish the time overhead when compiling with LTO.
- **Peak memory usage:** Insightful observations about the compile process might be gained by comparing the memory requirements to examine how much more memory LTO uses and if this could cause issues.
- **Amount of methods:** To observe the behavior of the inliner and the removal of functions, the amount of methods and in the resulting executable is registered. The results of the different inline limits can then be compared to ideally find an optimal limit.
- **Nodes count:** To determine if inlining has an effect on the resulting code size, the final amount of nodes is recorded. Furthermore, due to the potentially high node count when compiling with LTO previously unobserved problems might occur.
- **Run time measurements:** The changes induced by LTO are expected to have a positive effect on the run time. To determine a possible improvement the execution time of each benchmark is measured and compared.

The results of all the above listed measurements are displayed in this chapter. The absolute values are shown next to the relative differences. This allows to gain

an understanding of the actual values and simultaneously see if and how big the changes are. With the exception of the runtime measurements all relative values show a slowdown factor of the LTO implementation. Since it is expected to take up more memory and compile longer the quotient of the LTO variant divided by the non LTO version is shown: $\frac{value_{LTO}}{value_{non-LTO}}$.
For example if the reported relative value for the compilation time of a benchmark was 2×, this signifies that the LTO version took twice as long to compile.

4.2 Compile times measurements

The compile time measurements with and without LTO can be found in table 4.2. The table shows the absolute values for the compile duration. Alongside these the slowdown factor for the LTO version is displayed.
The quantity of source files and their size directly influences the compilation time. Therefore benchmarks, such as `art`, `mcf`, or `equake` which consist only of a few files, can be compiled within milliseconds. Compilation times for these benchmarks are difficult to compare and benchmarks that took less than one second are marked accordingly. For reference, the `cc1` benchmark is by far the biggest and thus takes the longest time to compile. Comparing the results, it is apparent that compiling with LTO requires decisively more time across all benchmarks. Additionally higher inline limits result in an increased compilation time. Only with the highest inline setting some of the SPEC projects show less of a slowdown than lower limits.
A significantly higher compilation time for the LTO benchmarks were to be expected. Since the LTO variant operates on the combined contents of all compilation units, analysis and optimization of the source code take up more time. By increasing the inline limit the compiler has a greater reach and can inline functions which previously exceeded the limit.

4.2.1 Load-store optimization

Passing the `-time` flag to CPARSER produces an overview of the time each optimization takes up. The optimization that stood out the most is the load-store optimization. Some optimizations depend on the amount of functions within the IRP. The load-store optimization has, at the time of writing, a quadratic worst case complexity in the number of IRGs. The reason for this is that every load or store operation that is removed invalidates the information about the use of entities. By

SPEC	Limit 375	Limit 750	Limit 1500	Limit 3000	Limit 6000
ammp	1.0×	0.9×	1.0×	1.0×	1.1×
art	–	–	–	–	1.0×
bzip2	1.0×	2.0×	1.0×	1.0×	1.0×
crafty	1.1×	1.1×	1.2×	1.5×	1.8×
equake	–	–	–	–	–
gzip	1.0×	0.7×	1.0×	1.0×	1.2×
mcf	1.0×	1.0×	1.0×	1.0×	1.5×
mesa	1.0×	1.0×	1.1×	1.2×	1.6×
parser	1.3×	1.1×	1.2×	1.1×	0.8×
perlbmk	1.4×	1.5×	1.8×	1.7×	1.6×
twolf	1.1×	1.0×	1.0×	0.8×	0.8×
vortex	1.0×	1.3×	1.7×	2.5×	4.3×
vpr	1.2×	1.0×	1.3×	1.4×	1.3×

Table 4.1: Relative compilation slowdown of the LTO variant in comparison to the non LTO version. In contrast to the compile time measurements in table 4.2 the load-store optimization is disabled.

SPEC	Limit 375			Limit 750			Limit 1500			Limit 3000			Limit 6000		
	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel
ammp	6	5	0.8×	6	6	1.0×	6	7	1.2×	7	9	1.3×	8	11	1.4×
art	<1	<1	–	<1	<1	–	<1	<1	–	<1	<1	–	1	1	1.0×
bzip2	1	1	1.0×	1	1	1.0×	2	1	0.5×	3	2	0.7×	5	4	0.8×
cc1	60	289	4.8×	74	413	5.6×	107	633	5.9×	185	1041	5.6×	552	2107	3.8×
crafty	9	8	0.9×	9	9	1.0×	9	11	1.2×	9	16	1.8×	11	23	2.1×
equake	<1	<1	–	<1	<1	–	<1	<1	–	<1	<1	–	<1	<1	–
gap	22	57	2.6×	27	84	3.1×	36	130	3.6×	59	181	3.1×	137	378	2.8×
gzip	3	1	0.3×	2	2	1.0×	3	3	1.0×	3	4	1.3×	4	6	1.5×
mcf	<1	<1	–	<1	<1	–	1	1	1.0×	1	1	1.0×	2	2	1.0×
mesa	21	42	2.0×	25	51	2.0×	28	63	2.2×	33	82	2.5×	37	133	3.6×
parser	8	8	1.0×	8	11	1.4×	12	16	1.3×	21	28	1.3×	90	78	0.9×
perlbmk	24	74	3.1×	30	105	3.5×	45	165	3.7×	89	280	3.1×	194	474	2.4×
twolf	13	11	0.8×	15	13	0.9×	19	22	1.2×	31	63	2.0×	64	176	2.8×
vortex	21	46	2.2×	22	71	3.2×	25	114	4.6×	30	218	7.3×	45	416	9.2×
vpr	6	7	1.2×	6	8	1.3×	7	11	1.6×	9	13	1.4×	11	20	1.8×

Table 4.2: The absolute compile times in seconds for the SPEC benchmarks in the LTO and non LTO version. The third sub column shows the relative slowdown factor of the LTO version. A value of „–“ represents compilation times that took only few milliseconds and are not representative.

removing a store operation a variable might become readonly and thus needs to be treated differently. This vastly increases the compile time of programs with a large amount of functions. The compile time measurements might thus be influenced by the incorrect behavior of this optimization. In order to investigate the effect this optimization has, it is disabled by specifying the `-fno-opt-load-store` flag. The benchmarks are compiled once more for the different inline limits.
In table 4.1 the relative compilation slowdown with LTO for different inline limits is shown. The slowdown values are reported with the load-store optimization disabled. The results for `gcc` and `gap` are not reported here, since disabling the load-store optimization causes an error for these two benchmarks which, at the time of writing, had not been resolved.
These values clearly show that the load-store optimization has an influence on the compilation time. Compiling with LTO still requires more time, but the additional compilation time might be reduced if such bottlenecks as the load-store optimization are removed.

4.3 Memory usage

The traditional approach of compiling source files separately has the obvious advantage that it only needs to have the source code of a single file in memory. Holding all compilation units at once and performing optimizations thereon puts a noticeable strain on the available memory. In table 4.3 the maximum allocated amount of RAM is reported for both the LTO and non LTO version. The third sub column shows the relative increase in memory usage. A first conclusion that can be drawn from these measurements is that the inline limit has an immense impact on the required memory. Doubling the limit, leads to an additional demand in RAM. For every replaced function call the callees content is inserted, effectively duplicating the original code for every call. With higher limits there are fewer functions to be inlined, thus a reduction in the relative memory usage is visible for some of the SPEC projects.
Since `cc1` is the largest benchmark in regard to code size, it requires the greatest amount of resources. The contrast between compiling with and without LTO manifests starkly when comparing the RAM usage for its 6000 inline limit. The absolute difference of required memory is roughly $7.5\,GB$. A further observation is that the memory requirements for the smaller benchmarks such as `art`, `equake`, `bzip2` with only one or few files is actually less than the non LTO variant. This is due to the fact that by enabling LTO unused functions can be removed even if they are not file private. Unexpected behavior is reported for the `vortex` benchmark. It has by far the greatest relative increase in RAM usage. For the 3000 inline limit the LTO variant requires up to 34 times more memory than the non LTO version.

SPEC	Limit 375			Limit 750			Limit 1500			Limit 3000			Limit 6000		
	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel
ammp	22	76	3.5×	22	80	3.6×	22	89	4.0×	25	103	4.1×	48	130	2.7×
art	13	11	0.8×	15	11	0.7×	17	15	0.9×	27	25	0.9×	40	38	0.9×
bzip2	27	22	0.8×	32	26	0.8×	41	33	0.8×	61	51	0.8×	83	85	1.0×
cc1	74	1117	15.1×	97	1542	15.9×	133	2348	17.7×	226	3760	16.6×	582	8064	13.9×
crafty	46	115	2.5×	46	126	2.7×	46	151	3.3×	46	200	4.3×	76	271	3.6×
equake	24	22	0.9×	24	22	0.9×	24	22	0.9×	31	28	0.9×	37	34	0.9×
gap	30	403	13.4×	52	602	11.6×	98	947	9.7×	196	1170	6.0×	366	2086	5.7×
gzip	11	22	2.0×	14	29	2.1×	20	43	2.1×	37	65	1.8×	41	107	2.6×
mcf	10	11	1.1×	12	15	1.2×	20	23	1.1×	34	37	1.1×	62	66	1.1×
mesa	42	329	7.8×	43	387	9.0×	51	455	8.9×	58	548	9.4×	94	869	9.2×
parser	22	98	4.5×	27	127	4.7×	43	176	4.1×	73	290	4.0×	208	550	2.6×
perlbmk	92	511	5.6×	150	744	5.0×	277	1145	4.1×	558	1877	3.4×	974	3097	3.2×
twolf	37	106	2.9×	37	114	3.1×	37	129	3.5×	37	156	4.2×	51	219	4.3×
vortex	34	471	13.9×	36	712	19.8×	38	1075	28.3×	54	1857	34.4×	126	3036	24.1×
vpr	19	91	4.8×	20	104	5.2×	26	119	4.6×	46	135	2.9×	72	198	2.8×

Table 4.3: Peak memory usage in megabytes (MB) when compiling with and without LTO. The third sub column value shows the factor by which the memory usage has changed for the LTO version relative to the non LTO variant.

Such a huge demand indicates that the LTO variant is able to perform a great deal more optimizations than its non LTO counterpart.

4.4 Remaining functions

To determine how many methods have been inlined the amount of functions remaining after optimization are calculated. Without LTO every translation is compiled separately and linked later on. The recorded amount of functions is therefore the sum total of all remaining functions as reported by libFirm for each compilation unit. Table 4.4 shows the quantity of remaining functions. The relative difference is once more displayed in the third sub column.
Unsurprisingly, the LTO variant outperforms the non LTO version significantly. With increased inline limits the inliner becomes more aggressive and can inline even more functions. Since LTO has information about all methods within the source code it can inline functions across file boundaries. The non LTO version is, however, limited to functions within the same file and changing the inline limit hardly changes the amount of functions inlined. By pushing the limit up to 6000 the small benchmarks `art` and `equake` even get reduced to a single function. Reducing so many functions is not necessarily an improvement. By inlining function contents the code layout in memory is affected, which can have both, negative and positive effects on caching behavior during execution. Therefore these measurements are foremost an indicator how well the inliner is able to perform. These results clearly show that the LTO extension is able to provide the inliner with more information and indicate that the LTO extension is operating correctly. Inlining inserts code instead of function calls, this code has to be optimized at every occurrence.

4.5 Node count

The actual code size in libFirm is measured in the amount of nodes that make up the program. Since LTO is able to inline more methods the node count is expected to differ from the non LTO version. Table 4.5 illustrates the absolute node counts and the relative change in the amount of nodes. The node counts are measured after the optimizations are run and before the code generation.
Most of the benchmarks have a higher node count with increasing limit. This corresponds with the assumptions that more functions can be inlined and thus the program contains more nodes. The smaller benchmarks show results similar to the memory usage measurements. The LTO variant contains fewer nodes than its non LTO counterpart. This supports the assumption, that some unused functions have been removed, which the LTO version is able to do, due to its holistic view of all

SPEC	Limit 375			Limit 750			Limit 1500			Limit 3000			Limit 6000		
	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel
ammp	179	152	85%	179	133	74%	179	118	66%	179	90	50%	179	74	41%
art	26	21	81%	26	13	50%	26	12	46%	26	10	38%	26	1	4%
bzip2	74	37	50%	74	28	38%	74	22	30%	74	17	23%	74	11	15%
cc1	2051	1831	89%	1972	1728	88%	1900	1660	87%	1835	1559	85%	1805	1489	82%
crafty	109	98	90%	109	95	87%	109	85	78%	109	75	69%	109	70	64%
equake	27	18	67%	27	16	59%	27	12	44%	27	3	11%	27	1	4%
gap	853	826	97%	852	795	93%	852	778	91%	852	760	89%	852	752	88%
gzip	77	38	49%	69	30	43%	66	25	38%	62	20	32%	62	11	18%
mcf	26	15	58%	26	9	35%	26	7	27%	26	3	12%	26	3	12%
mesa	1069	693	65%	1043	623	60%	1017	543	53%	1002	512	51%	990	480	48%
parser	323	207	64%	323	176	54%	323	150	46%	323	132	41%	323	119	37%
perlbmk	1039	908	87%	1016	885	87%	995	842	85%	982	819	83%	982	832	85%
twolf	190	144	76%	190	135	71%	190	98	52%	190	67	35%	190	52	27%
vortex	923	528	57%	923	507	55%	923	463	50%	923	443	48%	923	418	45%
vpr	212	168	79%	177	117	66%	156	86	55%	141	61	43%	130	52	40%

Table 4.4: The amount of functions remaining after optimization for both the LTO and non LTO version. The third sub column shows the percentage of the remaining methods in the LTO version in comparison to the non LTO version.

SPEC	Limit 375			Limit 750			Limit 1500			Limit 3000			Limit 6000		
	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel
ammp	107	107	1.0×	114	114	1.0×	128	124	1.0×	138	143	1.0×	155	165	1.1×
art	11	10	0.9×	12	10	0.8×	15	11	0.7×	18	12	0.7×	25	14	0.6×
bzip2	35	27	0.8×	44	34	0.8×	58	42	0.7×	90	51	0.6×	120	53	0.4×
cc1	1622	1740	1.1×	2066	2419	1.2×	2894	3705	1.3×	4458	5955	1.3×	7390	10666	1.4×
crafty	149	151	1.0×	158	169	1.1×	165	201	1.2×	171	264	1.5×	186	348	1.9×
equake	16	12	0.8×	17	12	0.8×	17	12	0.7×	19	12	0.6×	24	11	0.5×
gap	522	621	1.2×	679	955	1.4×	900	1513	1.7×	1304	1894	1.5×	2021	3326	1.6×
gzip	33	25	0.8×	38	34	0.9×	46	51	1.1×	53	70	1.3×	59	85	1.4×
mcf	8	7	0.9×	9	7	0.8×	10	8	0.8×	13	11	0.8×	19	17	0.9×
mesa	443	466	1.1×	490	559	1.1×	542	665	1.2×	614	811	1.3×	674	1272	1.9×
parser	153	146	1.0×	202	196	1.0×	299	277	0.9×	456	467	1.0×	763	867	1.1×
perlbmk	665	779	1.2×	891	1190	1.3×	1311	1889	1.4×	2196	3145	1.4×	3450	4897	1.4×
twolf	167	156	0.9×	179	169	0.9×	197	192	1.0×	234	234	1.0×	295	324	1.1×
vortex	597	715	1.2×	722	1091	1.5×	815	1662	2.0×	922	2900	3.1×	1094	4750	4.3×
vpr	134	139	1.0×	145	165	1.1×	155	190	1.2×	176	212	1.2×	225	274	1.2×

Table 4.5: The node count in thousands for every SPEC benchmark for the LTO as well as the non LTO version. The third sub column once more shows the relative increase in nodes for the LTO variant.

compilation units. The relative increase in memory is at a maximum for `vortex` as table 4.3 shows. This fits the node count measurements since the quantity of nodes is also the highest for every inline limit.

4.6 Run time measurements

Usually a program is compiled only once. The resulting executable, however, can be run as often as required. Therefore it is a small price to pay if the compilation process is expensive, but the compiled executable runs faster. To investigate if the LTO extension is able to improve the runtime, the execution time of each benchmark is recorded. To determine the relative run time speedup, the runtime of every non LTO program is divided by the execution time of the corresponding LTO binary. Table 4.6 shows the averaged runtimes of all SPEC benchmarks and resulting speedup factors. These results are generated by temci. Temci reports the absolute and speedup values alongside the standard deviation. For statistically significant results, temci requires the standard deviation to be less than one percent. This was the case for all benchmarks except `ammp`. Here the standard deviation exceeded this limit for all but the 1500 inline limit. The results for this benchmark might not be reliable and should be run more often. However, the results don't seem to be out of the ordinary and aren't run more often due to time constraints. The speedups and slowdowns show mixed results. The LTO version does not always produce better results. For `crafty` there is up to a 5.2% slowdown. The greatest improvement is registered for `vortex`. A maximum speedup of 25.3% is achieved for the 1500 limit. An improvement of this magnitude is unexpected. Most benchmarks show a similar pattern when the inline limit is changed. The first limit shows longer runtimes. Within the next or second next limit a minimum runtime is reached. Increasing the limit further causes an increase in the resulting runtimes. This trend is visualized in figure 4.1. In the upper half the geometric means of the absolute runtime for all SPEC benchmarks in the LTO and non LTO version are plotted. In the lower half the relative speedup factor measured by the quotient of the geometric mean of the non LTO over the LTO variant is displayed. According to this graph the greatest speedup can be found for an inline limit of 1500. Here the LTO version achieves a speed up of 2.6% percent.
One might now conclude that an inline limit of 1500 produces the best results for the limits used. However, this it not necessarily the case. If the absolute runtimes are considered it is apparent that the 1500 limit has the highest speedup in comparison with the non LTO version, but the absolute runtimes for this limit are actually higher. As can bee seen in the upper half of figure 4.1, the minimum absolute runtime is at the 3000 limit. For this inline limit the speedup is only 2.3% but the absolute runtime for the LTO as well as the non LTO version is the lowest.

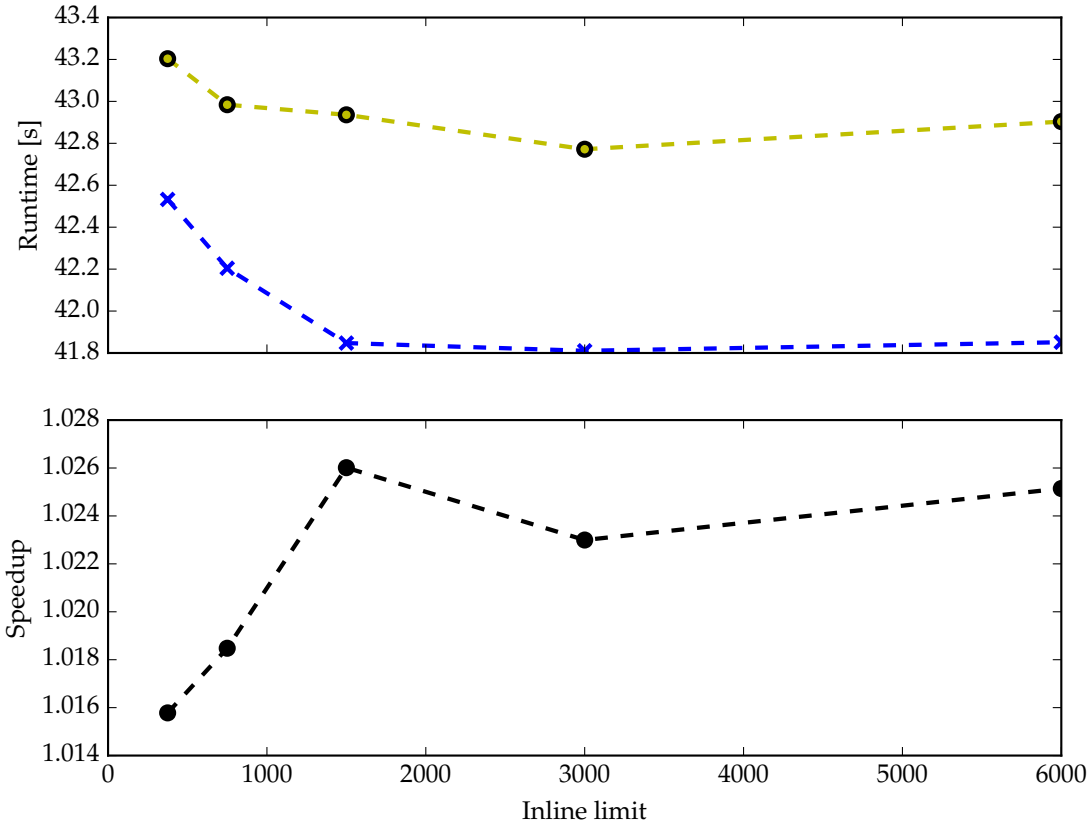


Figure 4.1: The upper plot shows the geometric means of the absolute runtimes over all benchmarks for the LTO and non LTO variant. In the lower the relative runtime speedup, the quotient of the non LTO over the LTO variant, is shown.

SPEC	Limit 375			Limit 750			Limit 1500			Limit 3000			Limit 6000		
	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel	Base	LTO	Rel
ammp	93.6	96.7	0.97	95.1	96.4	0.99	93.1	95.0	0.98	93.8	95.9	0.98	94.5	96.2	0.98
art	28.4	28.9	0.98	28.3	29.3	0.97	29.3	29.5	0.99	29.4	29.4	1.00	28.0	28.2	0.99
bzip2	52.0	49.5	1.05	50.7	48.4	1.05	50.6	47.3	**1.07**	50.1	48.4	1.04	50.2	47.6	1.05
cc1	23.5	23.9	0.98	23.9	23.7	1.01	23.8	24.0	0.99	24.0	24.3	0.99	24.5	24.5	1.00
crafty	28.9	29.8	0.97	28.8	29.7	0.97	28.7	29.6	0.97	28.8	29.9	0.96	28.9	30.5	0.95
equake	30.9	30.6	1.01	30.9	30.6	1.01	29.4	28.9	1.02	29.0	28.7	**1.01**	29.0	28.8	1.01
gap	28.7	28.2	1.02	28.7	27.8	1.03	28.8	27.7	**1.04**	28.7	27.7	**1.04**	30.3	27.9	1.09
gzip	62.8	63.4	0.99	62.2	61.3	1.01	62.6	60.3	**1.04**	62.2	60.8	1.02	62.2	60.4	1.03
mcf	22.6	22.5	**1.00**	22.6	23.0	0.98	22.6	22.6	1.00	22.7	22.6	1.00	22.7	22.6	1.00
mesa	55.8	55.3	1.01	55.6	54.6	1.02	54.6	54.7	1.00	54.5	54.1	**1.01**	54.7	54.1	**1.01**
parser	61.6	58.9	1.05	60.7	58.3	1.04	60.4	58.1	**1.04**	60.1	58.1	**1.03**	60.1	58.4	1.03
perlbmk	56.9	56.8	**1.00**	56.8	57.3	0.99	59.1	58.4	1.01	56.8	58.6	0.97	57.8	59.0	0.98
twolf	67.6	65.3	1.04	67.7	65.4	1.04	67.3	63.2	1.06	67.4	62.4	**1.08**	67.6	62.5	1.08
vortex	43.4	36.3	1.20	43.5	35.3	1.23	43.6	34.8	**1.25**	43.7	35.2	1.24	43.7	36.6	1.19
vpr	50.3	50.7	0.99	47.8	49.4	0.97	48.1	50.4	0.95	47.7	47.6	1.00	46.9	46.9	**1.00**
mean	43.2	42.5	1.02	43.0	42.2	1.02	42.9	41.8	1.03	42.8	41.8	**1.02**	42.9	41.9	1.03

Table 4.6: Averaged execution times of all SPEC benchmarks for different inline limits in seconds. Furthermore the relative run time speedup of the LTO version compared to the non LTO version is shown in the third sub column. Values greater than one indicate that the LTO variant is faster. For easier comparison each speedup is highlighted in green and a slowdown is colored in red. Measurements that show a speedup and also have the shortest execution time are marked in bold green.

For this reason some of the values in table 4.6 have been printed in bold. These measurements show a speedup while simultaneously being the fastest in terms of absolute runtime.

4.7 Observations and conclusions

Big projects, such as the SPEC benchmarks are usually split in multiple files. Functions that logically belong together are placed in the same file. Because of this, functions in the same module are more likely to be called by other methods from the same file. This scenario is optimal for a non LTO compiler since it is able to inline those functions; each file is a self contained unit. However, if there are function calls across file boundaries, a non LTO compiler cannot inline these functions.

This helps to explain why most benchmarks show little or no improvements. There is not much room for improvement since existing optimizations already perform well. However, small benchmarks such as `ammp` or `crafty` consistently achieve worse results when compiled with LTO. Inlining functions can have positive or negative effects depending on the caching behavior of the resulting executable. Inlining more functions is no guarantee that the program runs faster. Quite the contrary is true for the smaller SPEC benchmarks. If by chance the cache layout becomes beneficial a slight improvement is registered or vice versa. A conclusion that could be drawn from these results is that LTO is not beneficial for small projects.

4.7.1 The vortex benchmark

The `vortex` benchmark is by far the greatest anomaly with a 25% speedup. Simple caching effects or other small improvements are not capable of producing such a change. Therefore `vortex` is examined closer to determine what causes this difference in runtime performance. `Vortex` is a database application and one of the larger benchmarks. An initial assumption is that the code is split in an unfortunate way, which prevents a non LTO compiler from doing critical optimizations.

Inspection with the `perf` [13] command line tool indicates that a performance hotspot is the function `ChkGetChunk` in the `mem00.c` file. Since `vortex` performs a lot of memory operations it includes its own memory management code. The corresponding source files are: `mem00.c` which provides low level memory operations,

mem01.c and mem10.c which provide higher level features. The ChkGetChunk function is exclusively called from functions defined in mem10.c. An example function is shown in figure 4.2. These functions are all similar, they call ChkGetChunk to load a chunk of memory and cast it to the desired primitive value, e.g. MemGetShort to retrieve a short value. These memory management functions are excessively called throughout vortex.

```
boolean    MemGetShort  (numtype       Chunk,  indextype   Index,
ft F,lt Z,zz *Status, shorttype   *ShortValue)
{
char     *Base  = 0;
indextype Inset = 0;
addrtype  Ptr   = 0;
if (ChkGetChunk (Chunk, Index, sizeof(shorttype), McStat))
        if (*Status != Set_EndOfSet) {
                Base     = (char    *)Chunk_Addr(Chunk);
                Inset    = (indextype )(Index * sizeof(shorttype));
                Ptr      = (addrtype )(Base + Inset);
                *ShortValue = *(((short *)(Chunk_Addr(Chunk))) + Index);
                if (*(short *)Ptr != *ShortValue)
                        *Status = Err_BadStackMath;
        }

TRACK(TrackBak,"MemGetShort\n");
return(STAT);
}
```

Figure 4.2: An example function from vortex which calls ChkGetChunk.

This strengthens the previous assumption that some functions are repeatedly called across file boundaries which cannot be inlined. In order to test if this has an impact on the runtime performance the following steps are executed:

- mem00.c and mem10.c files are compiled into IR-files and linked via the LTO extended CPARSER into a single object file: mem.o
- vortex is recompiled but the mem00.o and mem10.o files are replaced by the newly generated object file mem.o.
- The benchmark is run again with the modified executable.

Limit	Base	Modified	LTO	Rel
375	43.4	38.7	36.3	1.07
750	43.5	39.4	35.3	1.11
1500	43.6	39.2	34.8	1.12
3000	43.7	39.6	35.2	1.12
6000	43.7	39.9	36.6	1.09

Table 4.7: Absolute runtime in seconds of `vortex` for the non LTO version, the modified non LTO variant and the LTO version. The last column shows the runtime speedup of LTO in comparison to the modified non LTO version.

In table 4.7 the runtimes of the modified `vortex` benchmark in comparison to the LTO version are displayed. The results show a definite improvement. The LTO variant is still faster by up to 12%, but the difference is halved in comparison to the previous value of 25%. We strongly suspect that the difference could possibly be minimized further if more files were combined, but this requires detailed knowledge of the vortex benchmark and was therefore skipped due to time constraints.

5 Conclusions and future work

The implemented LTO solution for libFirm is able to produce visible results. An overall speedup of 2.3% is achieved when compiling the SPEC benchmarks with LTO. For the `vortex` benchmark a speedup of up to 25% is reached. While this is an exception it proves that there are real world programs which hugely benefit from compiling with LTO. The drawback of LTO is that it takes up more memory and time during compilation. This should pose no problem since modern computers have vast resources and LTO only needs to be activated for the final build.
Nevertheless, due to the limited time there are still some issues which need to be addressed to improve the LTO support for libFirm.

5.1 Linker plugin

The current LTO implementation relies on compiling source code to IR-files. These IR-files are directly merged within libFirm and only a single object file is created and linked into an executable. This interrupts the customary build process and is therefore not an optimal solution.
More desirable is a linker plugin. Such a plugin extends the linker to recognize IR-files and takes care of interfacing with libFirm to compile and optimize the intermediate code. This setup doesn't require changes to the toolchain and can be easily integrated into the existing build process.

5.2 Definition vs. declaration

One difficulty during the implementation of the LTO functionality was to distinguish between type definitions and declarations. From the IR-file alone it is not apparent if a type denotes a declaration or a definition. It would be helpful if libFirm

provided a way to query this information. The current implementation relies on simple heuristics to determine which might prove insufficient.

Bibliography

[1] LLVM Website, "LLVM link time optimization: Design and implementation." `http://llvm.org/docs/LinkTimeOptimization.html`.

[2] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[3] G. Lindenmaier, "libFIRM – a library for compiler optimization research implementing FIRM," Tech. Rep. 2002-5, Sept. 2002.

[4] M. Braun, S. Buchwald, and A. Zwinkau, "Firm—a graph-based intermediate representation," Tech. Rep. 35, Karlsruhe Institute of Technology, 2011.

[5] M. Braun, S. Buchwald, M. Mohr, and A. Zwinkau, "An X10 Compiler for Invasive Architectures," Tech. Rep. 9, Karlsruhe Institute of Technology, 2012.

[6] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pp. 12–27, ACM, 1988.

[7] "Firm - Website." `http://pp.ipd.kit.edu/firm/`.

[8] ISO/IEC Committee, "ISO/IEC 9899:201x Programming languages — C." `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`, Apr. 2011.

[9] Standard Performance Evaluation Corporation, "SPEC CPU2000." `https://www.spec.org/cpu2000/`.

[10] J. Bechberger, "Besser benchmarken." `http://pp.ipd.kit.edu/publication.php?id=bechberger16bachelorarbeit`, Apr. 2016.

[11] J. Bechberger, "temci documentation." `http://temci.readthedocs.org/en/latest/`.

[12] Free Software Foundation (FSF), "time - GNU Project - Free Software Foundation (FSF)." `https://www.gnu.org/software/time/`.

[13] Perf developers, "Perf Wiki." `https://perf.wiki.kernel.org/index.php/Main_Page`.

Erklärung

Hiermit erkläre ich, Mark Weinreuter, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum

Unterschrift