A link-time optimization (LTO) approach in the EMCA program domain

Master of science degree project in embedded systems
Stockholm, Sweden 2013
TRITA-ICT-EX-2013:233

Khalil Saedi

KTH Royal Institute of Technology
School of Information and Communication Technology

Supervisor: Patric HEDLIN, FlexTools, Ericsson AB
Examiner: Prof. Christian SCHULTE, ICT, KTH
# Contents

Nomenclature vii

Acknowledgments vi

Abstract vii

1. Introduction 1
   1.1. Terminology .............................................. 1
   1.2. Background .................................................. 1
   1.3. Problem ........................................................ 1
   1.4. Related work ................................................ 2
   1.5. Organization ............................................... 2

2. Problem statement and methodology 5
   2.1. LLVM Standard LTO in conventional programming model .......... 5
   2.2. EMCA program domain ........................................... 5
   2.3. Link time optimization in EMCA .................................. 6
   2.4. Method .......................................................... 7

3. LLVM 10
   3.1. LLVM IR (Intermediate representation) .......................... 10
   3.2. LLVM modules .................................................. 11
      3.2.1. Front-end ................................................ 12
      3.2.2. Optimizer ............................................... 12
         3.2.2.1. Compile time optimization (static) .................. 12
      3.2.3. Back-end ................................................ 12
      3.2.4. Tools .................................................... 13
   3.3. LLVM advantages .............................................. 13
      3.3.1. Modern design ......................................... 13
      3.3.2. Performance ............................................ 14
      3.3.3. Modularity ............................................. 14
      3.3.4. License ................................................ 14

4. Link time optimization 16
   4.1. Linker ........................................................ 17
      4.1.1. Linker for embedded systems ................................ 17
   4.2. LLVM LTO advantages .......................................... 17
      4.2.1. Dead code elimination .................................. 17
      4.2.2. Interprocedural analysis and optimization .................. 18
      4.2.3. Function inlining ....................................... 20
<table>
<thead>
<tr>
<th>4.3. Link time optimization in LLVM</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1. Standard LTO by using the Gold linker</td>
<td>21</td>
</tr>
<tr>
<td>4.3.1.1. Gold and optimizer interaction</td>
<td>21</td>
</tr>
<tr>
<td>4.3.2. LTO Without Gold linker</td>
<td>22</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5. Results</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1. Standard LTO on X86 architecture</td>
<td>24</td>
</tr>
<tr>
<td>5.1.1. Compiling Clang</td>
<td>25</td>
</tr>
<tr>
<td>5.1.1.1. With Clang (Bootstrapping)</td>
<td>25</td>
</tr>
<tr>
<td>5.1.1.2. With GCC</td>
<td>25</td>
</tr>
<tr>
<td>5.1.2. Mibench</td>
<td>26</td>
</tr>
<tr>
<td>5.1.3. Discussion</td>
<td>30</td>
</tr>
<tr>
<td>5.2. LTO in EMCA (single core)</td>
<td>30</td>
</tr>
<tr>
<td>5.2.1. Discussion</td>
<td>32</td>
</tr>
<tr>
<td>5.3. Future works</td>
<td>32</td>
</tr>
<tr>
<td>5.4. Conclusion</td>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A. LTOpopulate module</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1. passManagerBuilder</td>
<td>35</td>
</tr>
<tr>
<td>A.2. LLVM LTO core, populateLTOPassManager</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B. Gold (Google linker)</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1. How to build</td>
<td>37</td>
</tr>
<tr>
<td>B.1.1. LLVM</td>
<td>37</td>
</tr>
<tr>
<td>B.1.2. GCC</td>
<td>38</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C. LLVM optimization options</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1. Mibench results</td>
<td>39</td>
</tr>
</tbody>
</table>

| Bibliography | 41 |
# List of Figures

2.1. EMCA memory structure ......................................................... 6

3.1. LLVM modules ........................................................................... 11

4.1. Separate compiling model .......................................................... 16
4.2. Dead code elimination ................................................................. 18
4.3. Sample program ......................................................................... 19
4.4. Program flowchart ..................................................................... 19
4.5. LTO without Gold linker .............................................................. 22

5.1. Program size in optimization level -Os with and without LTO, X86 ........ 27
5.2. Program size in optimization level -O2 with and without LTO, X86 ........ 28
5.3. Program size in optimization level -O3 with and without LTO, X86 ........ 29
5.4. Program size compiled by flacc with and without LTO ..................... 31
List of Tables

3.1. LLVM optimization levels ........................................ 13
4.1. Interprocedural optimization .................................... 20
5.1. Clang bootstrapping by Clang ................................. 25
5.2. Clang bootstrapping by GCC 4.8 ............................... 25
5.3. Program characteristics in Mibench ............................ 26
5.4. Program size in optimization level -Os with and without LTO, X86 .... 27
5.5. Program size in optimization level -O2 with and without LTO, X86 .... 28
5.6. Program size in optimization level -O3 with and without LTO, X86 .... 29
5.7. Object file size in optimization level -Os with and without LTO ....... 31
C.1. Optimization -O1 passes ........................................... 39
C.2. Mibench, llvm optimization option (code size) ................ 40
## Nomenclature

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EMCA</td>
<td>Ericsson Multi Core Architecture</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>IPO</td>
<td>Inter Procedural Optimization</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>JIT</td>
<td>Just In Time compiler</td>
</tr>
<tr>
<td>LLVM</td>
<td>Low Level Virtual Machine (compiler framework)</td>
</tr>
<tr>
<td>LTO</td>
<td>Link Time Optimization</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment</td>
</tr>
</tbody>
</table>
I would like to thank my supervisor Patric Hedlin, his knowledge and help have been invaluable.

I would also like to thank all of my group members in Flextools, in particular my manager Per Gibson, and my examiner professor Christian Schulte at KTH Royal Institute of Technology. They made this great opportunity a precious experience for me. I received assistance and very useful advice from Patrik Hägglund in Linköping during my thesis work.

My most sincere thanks also to Claes Sandgren, with whom I was lucky enough to share an office room with. I don’t remember exactly how many questions I asked him during work on my project- certainly more than a thousand! This research would have been impossible without his generous help.
Abstract

Multi-core systems on chip with a high level of integration are used in high performance network devices and parallel computing systems. Ericsson is using its own multi-core system (EMCA) for various high performance mobile network systems. EMCA, like most embedded multiprocessor systems, is a memory constrained system. Each core has limited amount of local and shared memory for code and data. To achieve high computational density on the system, it is very important to optimize code size to reduce both shared memory access and context switching costs for each computation node.

This thesis evaluates the link time optimization (LTO) approach based on a new LLVM back-end for EMCA architecture. Link time optimization (interprocedural optimization) is performed with the entire program code available all at once in link time, or immediately after linking the programs object files. The research carried out during this thesis proves that the LTO approach can be used as a solution for code size reduction in the EMCA program domain. The thesis also evaluates the link time optimization mechanism itself and shows its advantages in general. As for the experimental part, it provides implementation of LTO based on the LLVM framework, compatible with the current programming tool-chain for EMCA.
1. Introduction

This thesis is based on the need to investigate the potential of both link time optimization (LTO) as a vehicle for solving postponed build system decisions and proper link-time transformations, as well as the raw LTO capabilities provided by the LLVM compiler framework in EMCA\textsuperscript{1} programming domain. EMCA is Ericsson ASIC design. It is Ericsson’s concept of integrating several Digital Signal Processor (DSP) cores on a single Application Specific Integrated Circuit (ASIC).

This thesis has been done:

1. To evaluate the link time optimization feature in the LLVM compiler framework.
2. To implement LTO on the compatible LLVM framework for EMCA.

It is part of an ongoing project to retarget a LLVM compiler for EMCA.

1.1. Terminology

The thesis is both research about link time optimization and implementation of link time optimization in the EMCA program domain. To understand the terminology and expressions which are used throughout this thesis, the reader is supposed to have some knowledge in compiler technology and program compile and build procedure. Basic knowledge in code optimization concept and multi-core architectures will be helpful.

1.2. Background

The LLVM project in Ericsson is under substantial development. Its goal is to provide a high performance and modern compiler tool-chain for EMCA programming environment. EMCA architecture, like most embedded systems, is a memory constrained system. It has a limited amount of local and common memory for code and data. Common (shared) memory is a very precious resource for EMCA. It is very important to optimize the code size in order to reduce expensive context switching or load/store costs on the common and local memories. This thesis evaluates link time optimization (LTO) as a solution to optimize and reduce the code size on EMCA programming domain.

1.3. Problem

The program code size is very important in the EMCA program domain. It was necessary to evaluate link time optimization as a potential solution to reducing the code

\textsuperscript{1}Ericsson Multi Core Architecture
size. Standard link time optimization in LLVM supports a few hardware architectures only. EMCA is a unique heterogeneous multi-core architecture that has quite a different program build procedure. A small operating system is running on EMCA, which is responsible for resource allocation and processes and threads scheduling.

A program in EMCA may have multiple entry point functions. An entry point function is an execution starting point for a program\(^2\). They are defined as entry processes in OS. Each entry process is a running unit (Runblock) in the system. To avoid code duplication, Runblocks share parts of the program code. The code sharing concept is handled by the EMCA linker statically at link-time.

It was necessary to find a way to implement link time optimization for such a system, using EMCA’s link and build procedure. For several entry processes, it means several running units which are running in parallel. It was necessary to find a way to make LTO work for a whole program while considering parallel execution issues and possible dependencies between the programs running units.

### 1.4. Related work

The Western Research Laboratory (WRL) has done broad research about link-time code modification and optimization in the 1990s. This research showed that link time optimization of address calculation on a 64-bit architecture reduces code size about 10 percent\(^1\). Global register allocation at link-time considerably improved run-time speed about 10 to 25 percent\(^2\). David Wall from WRL showed impressive achievement in the run time speed with link-time code modification\(^3\).

It took a while for compiler developers to implement all those techniques in a standard build routine. LTO support proposal in GCC\(^3\) began in 2005. The first version was released in 2009 in GCC 4.5. Earlier in 2003, a whole program optimization project began, and was released in 2007. It extended the scope of optimization to the whole program\(^4\). LTO was one of the main goals for LLVM from the beginning. LLVM supports link time optimization since early releases.

Link time optimization promises to reduce code size as well as improving run-time performance. Compiling Firefox with link time optimization reduced the main Firefox module from 33.5MB to 31.5MB (%6)\(^5\). Considerable code size reduction occurred for SPECint 2006, and for SPECfp 2006 also\(^5\). Code size decreased about 15 percent when doing LTO for ARM executables\(^6\).

Since the focus in this thesis is program code size, all tests and evaluations have been carried out in order to measure code size in generated executable binary.

### 1.5. Organization

Chapter 2 deals with problem statement and methodology in more details. Chapter 3 explains LLVM modular structure and LLVM optimizer mechanism. Chapter 3 illustrates the standard optimization levels which are available in LLVM.

\(^{2}\)Starting point function is normally main function in conventional programming model

\(^{3}\)GNU Compiler Collection
In chapter 4, the link time optimization concept and LTO advantages are explained in detail. Chapter 4 illustrates LTO implementation in the LLVM framework. It will explain the standard LTO method in LLVM which is a target dependent method. Chapter 4 shows a target independent method too which is developed during the thesis. In the EMCA programming environment, since we use an in-house linker, LTO is feasible with the second method.

The chapter entitled “Results” illustrates results for LTO impact on code size. There are three tests; bootstrapping Clang (X86), Mibench testbenches(X86), Mibench testbenches (EMCA). The Conclusion section will summarize the report.
Problem statement and methodology
2. Problem statement and methodology

The thesis is both research about link time optimization and implementation of link time optimization in the EMCA program domain. This chapter discusses about problem description and methodology which is used to deal with the problem.

2.1. LLVM Standard LTO in conventional programming model

Standard link time optimization in LLVM supports a few hardware architectures only. It is implemented for Intel X86 (32 and 64 bit), PowerPC, SPARC and ARM architectures. LLVM requires a special linker to perform LTO, which is Gold Linker\(^1\). Standard LTO in LLVM is implemented for the conventional programming model.

In a conventional programming model, programs are organized into several source files; where each source file is compiled separately into an object file. The linker links object files and generates the executable binary. A program in the standard model is written in such a way that it is supposed to be a unique quantity on the system during execution time. Parallel execution for a program is implemented by using threads, or it is provided by an operating system which runs the program on more than one core at the same time (a needs task scheduler). In the conventional programming model programmer, compiler and linker suppose that the program is a single entity. There is a main (entry point) function, and program execution starts from this function. The entry point function has a decisive role in this model to do link time optimization.

2.2. EMCA program domain

The EMCA programming domain is a different environment compared to the conventional model in many ways. EMCA is a real time multi-processor system; processes and threads must start and finish in an expected time. It is not using cache memory to be a predictable system; all memory accesses are direct access. A tiny operating system is running on the system which manages all task scheduling and resource allocation. It is a nonpreemptive multitasking system. Programs are running in parallel on real dedicated computing cores. Function overlaying technique is used when the program size is larger than the local program memory size. Function overlaying provides ability for the operating system to keep part of the program in the common memory and load it to the computing unit whenever it is needed.

\(^1\)Google linker, a release of GNU linker
EMCA adapts a unique build system to satisfy all of those architectural requirements. EMCA is using a static linker. All linking rules and program attributes are defined statically for the program in advance. The size of the program is very important therefore overlay parts in the program must be defined in advance. Since there is no dynamic link in the system, programs must be compiled to be self-contained for the running purpose. Programs are sharing codes to avoid code duplication. The EMCA linker must provide a semi dynamic linking structure to provide code sharing between programs. This is done by interpreting programs linking rules statically at link-time.

In the compiler back-end, the code generator and linker produce programs as an application on top of the operating system. Basically, the operating system and programs are compiled together. All programs, after compilation, are turned to be processes and threads that are hooked into the OS. In this programming model since programs are running in parallel, they have more than one entry point. Each entry point is an entry process in the operating system that a computing core will be assigned to it in the run time.

2.3. Link time optimization in EMCA

It was necessary to find a way to implement link time optimization for such a system, using EMCA’s link and build procedure. As it is explained in above, EMCA has quite a different build and compile procedure. Programs start to run in parallel from multiple starting points. It was necessary to find a way to make LTO work for a whole program while considering parallel execution issues and possible dependencies between the programs parallel running units. LTO must be implemented in a way to work with in-house linker for EMCA.
2.4. Method

Step 1:

To evaluate LTO impact on the code size, first LLVM compiler and Gold linker are installed for X86 machine. Three different tests have been carried out, compiling Clang\(^2\) with GCC, compiling Clang with Clang itself, and finally some test cases from Mibench testbench\(^7\). Clang is compiled by GCC with and without LTO. Bootstrapping for Clang is done as well, with and without LTO. According to my supervisor’s recommendation, some test cases (10 tests) from Mibench testbench are selected. We use Mibench testbench in Ericsson for many compiler functionality tests for EMCA. In general, LTO showed that it can considerably reduce the code size in many test cases. This part has been completed to prove that LTO can be considered as a solution to reduce program code size.

Step 2:

As the second step, I started to hack on LLVM to understand compile processes, its optimizer mechanism, and its interaction with the linker during LTO. As previously explained in the problem section, LTO in LLVM is highly target dependent. We needed to find a way to implement link time optimization on LLVM for EMCA which satisfies all programming constrains and EMCA hardware architecture requirements.

Finally I found a method to complete interprocedural optimization (IPO passes) on the program which is relatively similar to what LLVM together with Gold linker are doing during LTO. For this reason intermediate representation (IR) of the code is used. In this method, IPO passes are running on the program IRs before delivering object files to the linker. For this reason, IPO passes need access to the whole code all at once. A LLVM tool (llvm-link) is used to combine the program’s intermediate representation files. Then, running IPO passes on the combined file manually.

This method has been tested for X86 architecture for the same test cases, since results for LTO with the standard method were available to compare with. The method has exactly the same optimized results as performing LTO with Gold linker. This step provided a reliable platform for me to implement LTO for EMCA during the next phase. The method can be used not only for EMCA but also for other non-supported targets.

Step 3:

Programs on EMCA are running in parallel at the same time on real dedicated computation nodes. Each program will change to become processes and threads on OS and may have more than one starting point. Each starting point is a running unit which will be assigned to a DSP or more. It is for this reason that we do not have a single entry function in programs for EMCA.

The main function, or a function, no matter what its name, is needed to perform link time optimization. LTO needs it to build program flow and to internalize other functions in the program. In EMCA architecture there is more than one function with the same

---

\(^2\)LLVM standard front-end for C family languages

\(^7\)Mibench testbench
role. We cannot pick one of them as a starting point for the whole program– running LTO passes on the whole program based on a single starting point removes other starting points, and all code belonging to these starting points, because they are unreachable from the selected starting point.

To solve the issue about multiple starting point, I changed my approach from LTO on whole program to LTO on each running block. For this reason, intermediate representation of the code is used. The program IR files are combined as explained in step 2. However, instead of running IPO passes once on the combined file, they ran on the combined file several times based on the number of the program starting points. A command line option 'dynMain' in LLVM is defined, which it can accept a function name and replace the main with it for LTO reasons. Running IPO passes with all starting points on the whole code, delivers optimized running blocks and guarantees preservation of any dependency between them.

As a conclusion, in the EMCA programming domain, LTO must be carried out on running blocks rather than on the program itself. This concept can be scaled up for any other optimizations on EMCA which require access to the whole code before generating executable binary.
LLVM
3. LLVM

LLVM (Low Level Virtual Machine) despite its name is not just a virtual machine. LLVM is “an umbrella project that hosts and develops a set of close-knit low-level tool-chain components (e.g., assemblers, compilers, debuggers, etc.), which are designed to be compatible with existing tools typically used on Unix systems”[8]. The project is under University of Illinois/NCSA Open Source License. It began as a research project in 2000 by Vikram Adve and Chris Lattner[9, 10]. The main goal was to have aggressive multi-stage lifelong optimization[11]. LLVM infrastructure provides compile time, link-time (interprocedural), run time and idle time optimization as well as whole program analysis and aggressive restructuring transformations[12, 8].

LLVM is a modern, open source and re-targetable compiler infrastructure in which all components are designed as a collection of modular libraries and tool-chains (parser, IR builder, optimizer passes, linker, assembler and code generator, etc.). This feature offers the ability to develop a compiler for new hardware architecture very quickly. LLVM has a powerful intermediate representation in SSA\(^1\) format. It is like a common currency for all optimizations, transformations and LLVM modules. LLVM has a just-in-time (run time virtual machine, JIT) that can execute LLVM IR directly.

3.1. LLVM IR (Intermediate representation)

The LLVM infrastructure is developed around its strong intermediate representation (IR). "It was designed with many specific goals in mind, including supporting lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, and aggressive restructuring transformations, etc. The most important aspect of it, though, is that it is itself defined as a first class language with well-defined semantics”[8].

LLVM IR’s assembly format is similar to an abstract RISC\(^2\) instruction set, with additional high level structures. LLVM has just 31 “opcodes”, so it is relatively easy to read and understand the code. Instructions have a three addresses format which means it can accept one or two input registers as the source and produce a result in the same or a different destination register. For example in instructions like add and subtract:

```
%x = add i32 1, %x
%poison = sub i32 4, %var
```

It uses load/store instruction to transfer data from/to memory[8].

A large part of LLVM IR structure is the same for all of the supported targets. However, it is not completely target independent, some features and data types are valid for a

---

\(^1\)Static Single Assignment- each variable in the code is assigned exactly once[13].

\(^2\)Reduced Instruction Set Computing
specific target only. For example, i40 or i24 in the EMCA architecture; they are 40 and 24 bit integer types that are used to hold fixed-point variables. One of the most important features of LLVM IR is its powerful type system; all functions and variables must have a type[14].

LLVM assembly is in three isomorphic representations. Human readable assembly (.ll), In-memory format and On-disk dense bit-stream (.bc) format. All three formats are equivalent, and LLVM can transform them easily without losing data.

### 3.2. LLVM modules

Almost all LLVM’s modules are implemented in C++ as set of libraries. Modularity allows LLVM to mix a static compilation mechanism with a virtual machine concept for optimization and code generation.

![LLVM modules diagram](image)

**Figure 3.1.:** LLVM modules

The compile process in LLVM framework starts with a front-end that produces LLVM IR. In the next step, LLVM optimizer transforms and optimizes the IRs (bitcode). Then LLVM static compiler (llc) translates optimized bitcode into the target assembly code. Finally, a native assembler produces an object file from the assembly code, and the linker links object files with (shared) libraries to generate the final executable binary. LLVM modules can be categorized as tools and work flow tool-chains (Fig. 3.1).
3.2.1. Front-end

Basically, any programming languages’ front-end can use LLVM as an optimizer or as a back-end, if they can produce LLVM IR in a proper format. There are many projects implemented as LLVM front-end but three standard front-ends for LLVM are: Clang, DragonEgg and llvm-gcc

- **Clang (Clang++)**: Clang is the native front-end which supports C, C++ and Objective C. It is fully GCC\(^3\) compatible[15, 16].

- **DragonEgg**: DragonEgg is a GCC plugin that uses LLVM tool-chain for optimization and code generation. The goal is to compile all other programming languages that are supported by GCC where Clang and LLVM together are not able to compile them[17].

- **llvm-gcc**: llvm-gcc is a modified version of GCC that is a C front-end for LLVM. It uses LLVM optimizer and code generator as back-end[18].

3.2.2. Optimizer

LLVM provides lifelong optimization in any possible stages, including compile time, link time, install time, run time and idle time (profile-driven) optimizations. These last three optimization stages are strictly target dependent, and in some cases they are not easily deployable in embedded systems such as EMCA architecture.

3.2.2.1. Compile time optimization (static)

All analysis and transformation passes are target independent\(^4\). LLVM optimizer is a launcher that holds and organizes the analysis and transformation passes. It runs passes from the pass list during optimization time. It checks their dependencies among other passes, and invokes and runs all other dependent passes as well. It is also possible to write a new user defined optimization pass.

LLVM standard optimization levels are illustrated in Table 3.1.

3.2.3. Back-end

LLVM back-end consists of a native assembler (llc) and a system linker. LLVM does not have a native linker yet, and relies on the system linker. It uses GNU linker for normal link process, and Gold linker for link time optimization. LLVM’s native linker project (lld) is still under development. LLVM can generate code for ARM, Hexagon, MBlaze, Mips, NVPTX, PowerPC and SPARC[19].

---

\(^3\)GNU Compiler Collection

\(^4\)This concept is a bit complicated because it is target independent for LLVM supported architecture, and not for a new or undefined system.
### 3.2.4. Tools

- **lli**: Low level interpreter or lli directly compiles and executes LLVM bitcode instructions. It uses LLVM in time compiler only for supported architecture.
- **llc**: It is LLVM static compiler which generates native assembly or object file.
- **llvm-link**: Links several bitcodes into one single bitcode file.
- **opt**: It simulates Clang optimization levels. It is a wrapper for optimizer. Input and output for opt is in IR format.

### 3.3. LLVM advantages

LLVM began as an academic compiler research project at the Illinois University. It rapidly became an open source umbrella project to develop a modern compiler infrastructure. An increasing number of IT companies are developing and using LLVM in a variety of different projects, and many research groups are working on LLVM optimization techniques. A very active developer community in industry and universities improve and enrich the LLVM framework. LLVM has a quite bright future and it can be considered as a replacement for GCC in the near future.

#### 3.3.1. Modern design

LLVM is written in standard C++ and is portable to most UNIX-based operating systems. It has a reusable source code which can be easily modified, and this allows LLVM to be

---

<table>
<thead>
<tr>
<th>option</th>
<th>optimization level</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>No optimization, fast compile time, considerably large executable size, easy to debug</td>
</tr>
<tr>
<td>-O1</td>
<td>Many optimizations passes, aims to compile fast and not adding to the size</td>
</tr>
<tr>
<td>-O2</td>
<td>Default level, it is mid-level optimization, all -O1 passes plus '-unroll' '-gvn' '-globaldce' '-constmerge' passes. It discards expensive optimizations in terms of compile time and size</td>
</tr>
<tr>
<td>-O3</td>
<td>All -O2 optimizations passes plus '-argpromotion'. It aims to produce faster executable program, in general it increases to the size and takes longer time to compile</td>
</tr>
<tr>
<td>-O4</td>
<td>Its equal to -O3 with -flto option, for supported platforms it can perform link time optimization</td>
</tr>
<tr>
<td>-Os</td>
<td>It is similar to -O2 with additional passes. Aims to keep binary size as small as possible</td>
</tr>
<tr>
<td>-Oz</td>
<td>Aims to aggressively reduce the size in any possible way, it may cause performance loss</td>
</tr>
</tbody>
</table>

**Table 3.1.:** LLVM optimization levels
used for different purposes. LLVM supports Just-In-Time compiling which is very useful for debugging. LLVM provides warning and error messages with accurate details. LLVM, together with Clang, provide a static code analyzer. A static analyzer can find possible bugs and give the suggestion to fix the bugs. LLVM has full support for a new accurate garbage collector[20]. It provides a mature and stable link time optimization. The LTO feature is a new option in compiler evaluation history.

3.3.2. Performance

LLVM, in many testbenches, compiles faster than other compilers. Lantter[21] showed that LLVM together with Clang can compile about 3 times faster than GCC for Carbon and PostgreSQL in analysis time[16]. LLVM-gcc 4.2 compiled SPEC INT 2000 42% and 30% faster in -O3 and -O2 optimization levels, compared to GCC 4.2. The generated executable binaries for SPEC INT 2000 by LLVM-gcc 4.2 run 4% and 5% respectively faster than the GCC generated binaries (-O3, -O2)[22].

3.3.3. Modularity

Modular design provides lots of freedom for compiler developers to build their custom compiler. The LLVM IR has a major role in this design as it is the common format for all LLVM components. LLVM modular design means that it is the best choice for educational purpose in universities to teach compiler technology. Students and researchers can easily modify or test LLVM tools for a specific goal.

3.3.4. License

LLVM is under Open Source Initiative 'three-clause BSD' license[23]. LLVM is available free of charge to commercial uses.
Link time optimization
4. Link time optimization

In a standard separate compilation model, like most projects in C language, programs are divided into several functions which are stored in source code files. Compiler compiles the program, file by file, and produces separate object files for each source file (Fig. 4.1). Writing programs in this model has many advantages:

- It provides maintainable code, which enables several developers to work on different parts of the code simultaneously.
- Each file is compiled independently. The re-compiling process after any changes is only necessary for the modified file, and not for all source files in the project.

![Separate compiling model](image)

Nonetheless, this model has a serious drawback. The scope of compiler optimizations is limited to a single source file at the time. It is not possible to have whole program optimization in this model.

For example a C source file may contain several functions, header files and external function calls to other source files or libraries. Since the compile process is in a file-by-file style, compiler has no idea how a function in a source file will be called externally. After static compiling, each object file has its own code and data address. It is the linker’s job to link all object files and reorder the code blocks to produce a final address table for program.

Link time optimization can help to carry out whole program optimization. It needs a close interaction between the linker and compiler. During the link time, it is possible for the compiler to have a vision of the whole program all at once. During LTO, optimization scope is extended to the whole program where interprocedural transformations can be done for all code modules. Previous researches about link time optimization[24, 6, 25] have shown that LTO has significant improvement in run time performance and code size reduction. However, link time optimization is useful when the program is rather large with many functions (external function calls) that are distributed in several source files.
4.1. Linker

The Linker combines a program’s object files and (shared) libraries. It generates final executable binary or another (shared) library. Generally, linkers are integrated into the compiler tool-chain as a back-end part, but they can be used separately too. The technology and concept of linker have not changed much since it was implemented for the first time decades ago.

"It binds more abstract names to more concrete names which permits programmers to write code using the more abstract names"[26]. Linker carries out symbol resolution and generates a final symbol table. It relocates addresses for the program and data memory addresses from all object files. Linker produces final relative address for the whole program.

4.1.1. Linker for embedded systems

Embedded systems generally have limitations in memory space for both program and data. Extra care is needed to generate executable program for embedded systems. Programmer, compiler, assembler, linker and loader have to consider memory limitations, parallel execution on multiple cores and other issues that rarely occur in programming for a general purpose processor. Developers for embedded systems often need more control over the compilation and code generation process. Some embedded systems don’t have a loader or operating system to carry out a dynamic link stage. For most embedded systems, all code relocation and binding must be done statically by the linker before generating the final program.

In the EMCA programming model, there is a limited amount of common memory, and each computation node has limited program and data memories. To have code size larger than the local program memory, EMCA uses a functions overlay technique. Overlaying allows part of the program to be kept in the common memory. The developer must consider this issue and define link rules to linker in advance. All programs will be part of the operating system which runs on the EMCA, and it is another constrain for the linker to build a proper executable.

4.2. LLVM LTO advantages

Since the optimizer has access to the whole program during link time optimization, it can extend analysis and transformation scopes to the whole program. LLVM has a couple of LTO passes both to analyze and transform the code. Here, some advantages of LTO are illustrated that are not achievable with other optimization levels.

4.2.1. Dead code elimination

Dead code elimination is one of the interesting aspects of the LTO optimizer. It is implemented by a set of analysis passes.
Link time optimization

- Propagating extracted values from the caller function on the callee function makes it possible to predict the outcome of the conditional branches. It results in the findings of unreachable branches.

- Unreachable code analysis pass traverses control flow path (for the whole program). It tracks unreachable marked sections to determine whether it is reachable from other code blocks or not. If it was safe to remove, then dead-code elimination pass will eliminate unreachable parts and remaining branches form the procedures and conditional statements[25].

![Image 1](image1.png)  
**Figure 4.2.**: Dead code elimination

Figure 4.2 shows two functions: `foo1` and `foo2` that are in 'b.c' and 'a.c'. Function `foo1` is defined externally and it is called from `foo2`. A traditional file-by-file compile model will generate an object file for 'a.c' which contains both if and elseif conditions. The elseif branch itself calls some mathematical functions. If, in the program, the only place that those mathematical functions are called is in `foo1`, then it will cause the compiler to include all those mathematical libraries and functions as well. It may result in significant increase of the code size.

In an LTO enabled compiler, it will find out that the value which goes to `foo1` is always positive and elseif condition is always false. It marks it as unreachable branch and the next pass (dead code stripper) will remove it from the final binary.

4.2.2. Interprocedural analysis and optimization

Link time optimization extends interprocedural optimization (IPO) through whole code. Figure 4.3 shows a very simple program\(^1\) which consists of two source files and a header file. The program starts with a main function in 'main.c', it calls `foo1` which is defined as an external function in 'a.c'. `Foo1` has a conditional branch that either calls `foo3` or just prints a message and returns. If `foo3` is called, it has a function call to `foo4`, which is defined as an external function in 'main.c'. `Foo4` has a print function only.

\(^1\)This example is obtained from llvm.org.
Figure 4.3.: Sample program

Figure 4.4 shows a flowchart for the program. It shows how the conditional branch is not taking the *yes* branch. It is obvious that function *foo2* is never called in the program and the conditional branch in function *foo1* is always false and function *foo3* will not be called in the program either.

A traditional file-by-file compiler will generate object files which contain native assembly code for all functions. Final executable binary has instructions for all functions. Column 1 (Non LTO) in Table 4.1 shows a symbol table for generated program that has *foo1*, *foo2* and *foo4*.

The compiler with LTO, acts completely differently:

Because *foo3* merely calls *foo4*, it is not generating any code, and so it is deleted in the link time.
Link time optimization

Table 4.1.: Interprocedural optimization

- Linker will inform the optimizer that foo2 in the symbol table is never called. It marks foo2 as dead code.
- A value propagator will find out that the condition is always false in foo1. It means foo3 will not be called in the program. It will mark foo3 as dead code.
- The compiler will find that foo4 is not called from anywhere else. It marks foo4 as dead code too.
- A dead code stripper pass removes all the dead marked codes.

Column 2 (LTO) in Table 4.1 shows a symbol table for the generated program with LTO. foo1, foo2, foo3 and foo4 are just removed and printf function is inlined. The program is compiled with Clang, with and without the LTO option. Without considering other optimization levels, the code size reduced from 550 Byte in NonLTO to 396 Byte when LTO is applied. About 30% less code size with expected better run-time performance3.

4.2.3. Function inlining

To eliminate function call and return overheads, caller function can include the called function’s body inside its own function body. It is called function inlining. Without LTO, function inlining can just happen inside the source file scope.

A compiler with LTO support can perform extensive inlining on the whole program regardless of whether a function is defined externally or internally. There are some factors that determine if it is beneficial to do inlining or not. Compiler uses heuristics on function calls. For example, how a function is called and how many times it is called, how large the function instructions size is or if it is called in a loop or not, etc. It is a tradeoff between code size and run-time performance when compiler is doing inlining. Boundary

3It is not making sense in such a simple and small program to see the execution time difference.
is not clear, while profile-guided post link optimization may provide accurate information to take an accurate decision about inlining a function.

4.3. Link time optimization in LLVM

LLVM is using the Gold linker to carry out standard LTO. Since link process in EMCA is a bit different to a regular link system, LTO must be carried out using the EMCA native linker. I found another way to carry out LTO in LLVM by moving optimization procedure one step before generating the assembly code. This method is using IR format of the code and is target independent. The method is tested for X86 architecture, it generates exactly the same results as the LLVM standard LTO method. Both methods are explained here.

4.3.1. Standard LTO by using the Gold linker

During LTO, LLVM postpones optimizations to link-time. All optimizations in LLVM to operate need intermediate representation of the code. LLVM generates bitcode (IR) format inside the object files during LTO. It needs a special linker to understand IR format inside object files for this reason, LLVM needs the Gold linker. “This is achieved through tight integration with the linker. In this model, the linker treats LLVM bitcode files like native object files and allows mixing and matching among them. The linker uses libLTO, a shared object, to handle LLVM bitcode files. This tight integration between the linker and LLVM optimizer helps to carry out optimizations that are not possible in other models”[27].

-O4 is the standard flag to carry out LTO in LLVM which is equal to -O3 -flto. Linkers with LTO support, make it simple for developers to carry out link time optimization without any significant change in the project’s make files or build process.

4.3.1.1. Gold and optimizer interaction

LLVM tightly integrates linker with its optimizer to take advantage of collected information by linker during the link stage. This information contains a list of defined symbols which is more accurate than any other data collected by other tools in a normal build process. All LTO functions are implemented in the libLTO library[27].

- Building global symbol table: Gold reads all object files which are in bitcode format to collect information about defined and referenced functions. The linker builds a global symbol table from the collected information.

- Symbol resolution: After building a symbol table for the whole program, Gold resolves symbols.

- Optimization: The optimizer finds live code modules and items in the table and starts optimization passes. If dead code stripping is enabled, then it removes unreachable parts. The code generator produces the final native object files after optimization.

- Final symbol resolution: Gold updates the symbol table after optimization and dead code stripping. It is the final step, and Gold links native object files just like the normal linking process, and generates the final executable binary.
4.3.2. LTO Without Gold linker

In this method, link time optimization is running on the program’s combined bitcodes (IR). LLVM framework has a tool that combines several bitcode files into a single bitcode file. This tool is llvm-link. After linking all bitcode files by llvm-link, the combined file is passed to the optimizer (opt).

In this stage, opt runs interprocedural optimization passes on the combined file. The output of this optimization will be an LTO optimized bitcode file. This file is passed to llc to produce native assembly code. Figure 4.5 illustrates this procedure.

![Figure 4.5: LTO without Gold linker](image)

1- Front end, compiles the source files and emits bitcode formats
   
   ```bash
   clang -emit-llvm -c foo1.c -o foo1.bc
   clang -emit-llvm -c foo2.c -o foo2.bc
   ```

2- llvm-link combines all bitcode files
   
   ```bash
   llvm-link foo1.bc foo2.bc -o all.bc
   ```

3- Opt is running intermodular optimization on the combined file
   
   ```bash
   opt -Ox -std-link-opts all.bc -o lto_optimized.bc
   -Ox = 0,2,3,s,z
   ```

4- llc produces native assembly (target dependent)
   
   ```bash
   llc lto_optimized.bc -o lto_optimized.s
   clang -Ox lto_optimized.s -o lto_binary
   ```

The method has the same performance as the standard method. This method is used with some modifications to evaluate LTO in EMCA.
Results
5. Results

This chapter illustrates results for tests that have been carried out to evaluate the LTO effect on code size reduction. Since the focus of this thesis is on the code size, all data presents the code size of final executable binaries. Tests are in three categories:

1- Evaluating standard LTO on X86 as a potential solution to reduce the code size

The X86 architecture is selected because LLVM fully supports X86 and standard LTO is available for it. These tests helped me to understand LLVM structure and optimization mechanism.

- Bootstrapping Clang: Clang is compiled by itself. An extra test has been done to compile Clang with GCC to evaluate LTO efficiency in other compilers too.
- Mibench testsuite: It is a standard testbench for embedded systems and we use it in Ericsson for many compiler functional tests.

2- Functional test for my LTO method (without Gold) on X86

To be sure that my LTO method is a reliable method, it is used to compile Mibench testsuite for X86. Since it had exactly the same results as the standard LTO method, results are not include here.

3- Functional test for LTO implementation on EMCA

LTO implementation is a little different on EMCA. For LTO impact on code size reduction, Mibench testcases are selected. Some of Mibench testcases are modified to be compilable by the EMCA C compiler

5.1. Standard LTO on X86 architecture

The X86 architecture is selected because it is the only platform that LLVM currently fully supports. It was suitable to get familiar with the different components of LLVM, and also for understanding LTO mechanism. Since standard LTO supports X86, it was best choice to develop and test my LTO method on it, and compare it with standard LTO.
5.1.1. Compiling Clang

Tests have been carried out with optimization levels -Oz and -O3 in both LLVM and GCC. The test platform was SUSE Linux 64 bit. LLVM 3.2 and GCC 4.8 are used. The Gold linker version is binutils-2.22-gold.

5.1.1.1. With Clang (Bootstrapping)

The “bootstrapping” term in compilers refers either to the development of a compiler for a special programming language by that programming language itself, or the generating of the compiler by compiling with itself again. Clang has built by itself- the normal procedure is building Clang by GCC.

<table>
<thead>
<tr>
<th></th>
<th>-Oz</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non LTO</td>
<td>17164168</td>
<td>25586920</td>
</tr>
<tr>
<td>LTO</td>
<td>16333464</td>
<td>28025336</td>
</tr>
<tr>
<td>size</td>
<td>-5%</td>
<td>+9%</td>
</tr>
</tbody>
</table>

Table 5.1.: Clang bootstrapping by Clang

Applying LTO with optimization level -Oz reduces the code size about 5%, 830KB of reduction. However a code size increase occurs while carrying out LTO with optimization level -O3. The size goes from 255MB to 280MB, 9% more. -O3 is the maximum level of optimization for run-time speed and in most cases generates larger code size. It is doing aggressive inlining that results in a code size increase.

5.1.1.2. With GCC

Table 5.2 shows results for compiling Clang by GCC 4.8 with and without LTO.

<table>
<thead>
<tr>
<th></th>
<th>-Os</th>
<th>-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non LTO</td>
<td>15415032</td>
<td>27471864</td>
</tr>
<tr>
<td>LTO</td>
<td>14167456</td>
<td>23186840</td>
</tr>
<tr>
<td>size</td>
<td>-8%</td>
<td>-18%</td>
</tr>
</tbody>
</table>

Table 5.2.: Clang bootstrapping by GCC 4.8

GCC produces smaller code size when LTO is applied both in -Os and -O3 optimization levels, 8% and 18% smaller respectively.
5.1.2. Mibench

Mibench is a free, commercially representative embedded benchmark suite[7]. It has standard tests for automation tool, operating systems, network, office tools, security, and telecommunication protocols. The platform that is used for this test was Ubuntu 12.04 (64 bit) running as a virtual machine on Intel Core i5 (2.4 GHz).

For this test LLVM version 3.2 and Gold linker from binutils-2.22-gold are used. The Gold linker installation is explained in Appendix B.

Tests are selected from Automotive, Network, Office and Telecoms suites. Table 5.3 shows the number of source files and functions for each test. We will see in the next pages how the program size and the number of source files are important for link time optimization.

<table>
<thead>
<tr>
<th>*</th>
<th>number of source files</th>
<th>number of functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath_large</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>bitcount</td>
<td>14</td>
<td>28</td>
</tr>
<tr>
<td>susan</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>dijkstra</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>patricia</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>ispell(buildhash)</td>
<td>25</td>
<td>47</td>
</tr>
<tr>
<td>rsynth(say)</td>
<td>43</td>
<td>63</td>
</tr>
<tr>
<td>adpcm(timing)</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>adpcm(rawaudio)</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>adpcm(rawdaudio)</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

*Table 5.3.: Program characteristics in Mibench

* It is the number of C files and header files in the program directory

All optimization levels are tested with and without LTO. Figure 5.1 and Table 5.4 show optimization for level s (size). Since the focus of this thesis is on the code size, data for optimization level -Os is the best result for code size. Figure 5.2 and Table 5.5 illustrate optimization results with level 2 (default) and finally Figure 5.3 and Table 5.6 show results for optimization level 3 (speed).

As mentioned earlier, the Mibench testbench is tested with my method also; these results are identical for that method as well.
Figure 5.1: Program size in optimization level -Os with and without LTO, X86

Table 5.4: Program size in optimization level -Os with and without LTO, X86
**Figure 5.2.** Program size in optimization level -O2 with and without LTO, X86

<table>
<thead>
<tr>
<th>program</th>
<th>-O2</th>
<th>-O2 -flto</th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath_large</td>
<td>3212</td>
<td>2878</td>
<td>-10%</td>
</tr>
<tr>
<td>bitcount</td>
<td>1884</td>
<td>1308</td>
<td>-31%</td>
</tr>
<tr>
<td>susan</td>
<td>23836</td>
<td>24956</td>
<td>+5%</td>
</tr>
<tr>
<td>dijkstra</td>
<td>1580</td>
<td>1212</td>
<td>-23%</td>
</tr>
<tr>
<td>patricia</td>
<td>1820</td>
<td>1708</td>
<td>-6%</td>
</tr>
<tr>
<td>ispell(buildhash)</td>
<td>23580</td>
<td>21584</td>
<td>-8%</td>
</tr>
<tr>
<td>rsynth(say)</td>
<td>27372</td>
<td>20300</td>
<td>-26%</td>
</tr>
<tr>
<td>adpcm(timing)</td>
<td>1404</td>
<td>1340</td>
<td>-4%</td>
</tr>
<tr>
<td>adpcm(rawaudio)</td>
<td>1244</td>
<td>956</td>
<td>-23%</td>
</tr>
<tr>
<td>adpcm(rawdaudio)</td>
<td>1228</td>
<td>860</td>
<td>-30%</td>
</tr>
</tbody>
</table>

**Table 5.5.** Program size in optimization level -O2 with and without LTO, X86
Figure 5.3.: Program size in optimization level -O3 with and without LTO, X86

<table>
<thead>
<tr>
<th>program</th>
<th>-O3</th>
<th>-O3 -flto</th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath_large</td>
<td>3212</td>
<td>2878</td>
<td>-10%</td>
</tr>
<tr>
<td>bitcount</td>
<td>2028</td>
<td>1452</td>
<td>-28%</td>
</tr>
<tr>
<td>susan</td>
<td>24972</td>
<td>25644</td>
<td>+3%</td>
</tr>
<tr>
<td>dijkstra</td>
<td>1580</td>
<td>1212</td>
<td>-23%</td>
</tr>
<tr>
<td>patricia</td>
<td>1820</td>
<td>1708</td>
<td>-6%</td>
</tr>
<tr>
<td>ispell(buildhash)</td>
<td>24220</td>
<td>21532</td>
<td>-11%</td>
</tr>
<tr>
<td>rsynth(say)</td>
<td>27900</td>
<td>20524</td>
<td>-26%</td>
</tr>
<tr>
<td>adpcm(timing)</td>
<td>1404</td>
<td>1340</td>
<td>-4%</td>
</tr>
<tr>
<td>adpcm(rawcaudio)</td>
<td>1244</td>
<td>956</td>
<td>-23%</td>
</tr>
<tr>
<td>adpcm(rawdaudio)</td>
<td>1228</td>
<td>860</td>
<td>-30%</td>
</tr>
</tbody>
</table>

Table 5.6.: Program size in optimization level -O3 with and without LTO, X86
5.1.3. Discussion

As the results showed, LTO has significant impact on code size reduction in most tests. Illustrated sizes in the above figures are size of ‘.text’ section in the executable binary, with and without LTO.

Table 5.4, 5.5 and 5.6 show that in all tests, applying link time optimization reduces code size except in a “susan” test. LLVM has 4 different standard optimization levels: -O2 is default optimization level (moderate), -O3 is for run time speed and -Os and -Oz are for code size optimization. In all optimization levels when LTO is applied, the code size is reduced considerably. Even when it was applied together with -O3 it decreases the code size. Normally -O3 increases the code size a lot.

In the test “susan”, LTO is reducing the code size in optimization level -Os just a little. However it increases the code size, 5 and 3 percents when LTO is applied with level -O2 and -O3. This is because in the “susan” test there is just one source file. Compiler has access to the entire program. The Added size in the “susan” testcase occurred because of aggressive inlining when LTO is applied.

Test rsynth (say) in Mibench testsuit is a very good example of how LTO reduces the code size for a large program. The rsynth (say) is a text of speech program written in C language in 43 source and header files. It is a rather large test in comparison to other tests in Mibench. Applying LTO together with -Os reduces the code size from 20700 to 15100 byte, a 27% of reduction. Applying LTO with other optimization levels has the same effect.

Link time optimization largely depends on the program structure. It generally depends on how a program is organized into several source files and how many external functions are defined in the code. There are no logical trends between the size of the program (number of source files and functions) and code size reduction during LTO. It depends how program developers wrote the codes in the source files.

Before having the standard link time optimization option, it was down to developers to take care of intermodular dependencies in the program. Because of program size increasing very quickly in the last years, it is almost impossible to keep track of intermodular dependencies in the program. For example, a typical web browser program like FireFox has about 6 million lines of code in more than 1000 source files.

Overall, these tests proved that LTO can be considered an efficient solution to reduce the code size.

5.2. LTO in EMCA (single core)

Ten testcases from Mibench testbench are picked. Small modifications have been done on the source codes to adjust them with the EMCA programming environment. Since some of the test cases can’t be compiled by flacc (EMCA C compiler), instead of tests susan, ispell and rsynth (say), other tests are replaced. All tests are compiled by LLVM version of falcc with and without LTO.

To have an accurate view about LTO impact on the code size, (.text) for the produced object file is measured. As can be seen in figure 5.4 and table 5.7, in all testcases applying link time optimization reduces the code size.
Programs compiled for Phoenix III

Figure 5.4.: Program size compiled by flacc with and without LTO

*For basicmath all data type changed to float. Cubic function solving section is discarded.

Table 5.7.: Object file size in optimization level -Os with and without LTO
5.2.1. Discussion

Results for EMCA in single core mode shows that LTO approach has considerable impact in code size reduction. Mibench testcase results for EMCA and X86 architecture can not be compared together. Each machine has its own instruction set. For the EMCA C compiler, some testcases are modified and some of them were failed to compile properly in the EMCA compiler. For example, EMCA dose not support file handling or dynamic memory allocation functions. Some other testcases are selected for EMCA from Mibench testsuit, however X86 and EMCA have 6 similar testcases for LTO test.

As can be seen LTO reduces the code size relatively more compare to the code size reduction in same testcases for X86 architecture. The main focus in this stage at Ericsson is to implement initial version of LLVM back-end for EMCA to work properly and correctly. The code size maybe is not optimized well in this version of implementation, but LTO approach showed that it can help to cover this issue in a good way.

5.3. Future works

Extra research can be carried out to compare the quality of the assembly code generated with and without LTO. Link time optimization promises to deliver optimized program which most likely has a better run time speed. Further research can evaluate the LTO impact on the run time speed. LLVM has two optimization levels for code size optimization-optimization level s and z. They are different in their degree of function inlining. Further research is necessary to evaluate their impact on the program when they are used together with LTO.

5.4. Conclusion

Nowadays, multicore systems on chip with high level integration are used in high performance network devices and intensive parallel computation systems. Ericsson is using its own ASIC designed multicore system on chip (EMCA) for various high performance mobile network systems. EMCA, like most embedded multiprocessor systems, is a memory constrained system. Each core has a limited amount of local and shared memory for code and data. To achieve higher computational density on the system, it is very important to optimize code size to reduce both shared memory access and context switching costs for each computation node.

A new LLVM back-end for EMCA architecture is under development in Ericsson. This thesis has evaluated link time optimization (LTO) features on LLVM compiler as a solution to reduce code size. As the experimental part, the thesis showed a LTO implementation model on LLVM back-end for EMCA.

Link time optimization needs a close interaction between linker and compiler. Just in the link time or immediately after linking, it is possible for the compiler to have a vision of the whole program all in once. During LTO, optimization scope is extended to the whole program. As shown in the results section, LTO can be considered as a potential solution to reduce the code size.
LLVM compiler framework was originally designed for conventional (standard) programming models. Standard link time optimization in LLVM is implemented based on the conventional programming structure. It is designed in such a way that developers don’t need a significant change in program compile and build procedure. It simply needs, in the back-end part, the system linker to be changed with a specific linker (Gold linker). So in standard LTO they kept build procedure the same, but changed the back-end part.

Since EMCA is a multiprocessor system that has quite a different build and compile procedure, problems show up when we want to implement link time optimization in this architecture. EMCA is using a special linker that is systematically different to a normal linker. Despite standard link time optimization, I implemented LTO on EMCA by keeping back-end (linker) the same, but changed compile and build procedure for this reason.

Link time optimization had a significant impact on code size reduction for EMCA in single core mode. It needs to be tested with some real network applications for multi core mode soon.
Appendices
A. LTO populate module

A.1. passManagerBuilder

Function populateLTOPassManager is a member of PassManagerBuilder class that adds LTO passes to the pass list. PassManager will make sure to run each analysis and optimization pass in a correct time sequence.

A.2. LLVM LTO core, populateLTOPassManager

PopulateLTOPassManager is the core function which creates LTO passes. 'LibLTO.so' is the standard LTO library, used by Gold linker, calls this function to do interprocedural optimization. It is a function in passManagerBuilder class, located in llvm/lib/Transforms/IPO/passManagerBuilder.cpp. Its prototype is:

```cpp
void PassManagerBuilder::populateLTOPassManager(
    PassManagerBase &PM,bool Internalize,bool RunInliner,bool DisableGVNLoadPRE)
```

Here are all passes which populateLTOPassManager creates:

```cpp
// Provide AliasAnalysis services for optimizations.
addInitialAliasAnalysisPasses(PM);

if (Internalize)PM.add(createInternalizePass);
// Propagate constants at call sites into the functions they call.
PM.add(createIPSCCPPass());
PM.add(createGlobalOptimizerPass());

// Remove duplicated global constants.
PM.add(createConstantMergePass());
PM.add(createDeadArgEliminationPass());

// Reduce the code after globalopt and ipsccp.
PM.add(createInstructionCombiningPass());

// Inline small functions
if (RunInliner) PM.add(createFunctionInliningPass());

// Remove unused exception handling info
PM.add(createPruneEHPass());

// Optimize globals again
```
if (RunInliner) PM.add(createGlobalOptimizerPass());

// Remove dead functions.
PM.add(createGlobalDCEPass());

PM.add(createArgumentPromotionPass());

// The IPO passes may leave cruft around. Clean up after them.
PM.add(createInstructionCombiningPass());
PM.add(createJumpThreadingPass());

// Break up allocas
if (UseNewSROA) // Scalar Replacement Of Aggregates
    PM.add(createSROAPass());
else PM.add(createScalarReplAggregatesPass());

// Run IP Alias Analysis driven optimizations.
PM.add(createFunctionAttrsPass());
PM.add(createGlobalsModRefPass());

// Hoist loop invariants.
PM.add(createLICMPass());

// Cleanup the code, remove redundancies and dead code.
PM.add(createGVNPass(DisableGVNLoadPRE));
PM.add(createMemCpyOptPass()); // Remove dead memcpys.
PM.add(createDeadStoreEliminationPass());

// Cleanup and simplify the code after the scalar optimizations.
PM.add(createInstructionCombiningPass());
PM.add(createJumpThreadingPass());

// Delete basic blocks, which optimization passes may have killed.
PM.add(createCFGSimplificationPass());

// Discard unreachable functions.
PM.add(createGlobalDCEPass()); [28]
B. Gold (Google linker)

Gold (Google release of system linker) is a new open source linker that is developed by Lance Taylor at 2008. Gold aims to be a drop-in replacement for GNU linker (ld-bfd). It is part of the standard GNU binutils package. Gold is compatible with GCC 4.0+ releases and just supports ELF format on Unix base platforms.

LLVM needs the Gold linker to perform link time optimization. LTO for LLVM is just available in Unix base platforms. GCC uses Gold to do LTO too. “As an added feature, LTO will take advantage of the plugin feature in gold. This allows the compiler to pick up object files that may have been stored in library archives”[29].

B.1. How to build

You need to obtain binutils package by GIT or SVN. Here the LLVM build process is included[30].

```
$ mkdir binutils
$ cd binutils
$ cvs -z9 -d:pserver:anoncvs@sourceware.org:/cvs/src login
   {enter "anoncvs" as the password}
$ cvs -z9 -d:pserver:anoncvs@sourceware.org:/cvs/src co binutils
$ mkdir build
$ cd build
$ ../src/configure --enable-gold --enable-plugins
$ make all-gold

B.1.1. LLVM

- Rebuild LLVM with:

  `./configure --with-binutils-include=/path/to/binutils/src/include` .......

LLVM (–with-binutils-include) will generate LLVMgold.so in $DIR/lib directory, this is the shared library that the Gold plugin uses over libLTO to do LTO.

- set up bfd-plugins:

  `cd path/to/lib; mkdir bfd-plugins; cd bfd-plugins; ln -s ../LLVMgold.so ../libLTO.so`
B.1.2. GCC

Rebuild GCC with:

```
./configure --enable-gold=default --enable-lto ......
```

Most GCC (4.5 +) releases come with LTO wrapper.

```
-fuse-linker-plugin
```

gcc -fuse-linker-plugin will call gold with linker plugin to perform LTO instead of collect2.
C. LLVM optimization options

LLVM is designed to be fully compatible with GCC. It inherited all compilation syntax from GCC, but it is not essentially equal in compile and optimization mechanism.

- **O0**: It is basically no optimization.
- **O1**: It is first level of optimization. it skips expensive transformations

<table>
<thead>
<tr>
<th>targetlibinfo</th>
<th>-no-aa</th>
<th>-basicaa</th>
<th>-preverify</th>
<th>-globalopt</th>
</tr>
</thead>
<tbody>
<tr>
<td>-tbaa</td>
<td>-ipsccp</td>
<td>-deadargelim</td>
<td>-instcombine</td>
<td>-functionattrs</td>
</tr>
<tr>
<td>-dse</td>
<td>-adce</td>
<td>-scpd</td>
<td>-notti</td>
<td>-indvars</td>
</tr>
<tr>
<td>-simplifycfg</td>
<td>-basiccg</td>
<td>-prune-eh</td>
<td>-always-inline</td>
<td>-simplify-libcalls</td>
</tr>
<tr>
<td>-lazy-value-info</td>
<td>-correlated</td>
<td>-tailcallelim</td>
<td>-reassociate</td>
<td>-loops</td>
</tr>
<tr>
<td>-lcssa</td>
<td>-loop-rotate</td>
<td>-licm</td>
<td>-loop-umswitch</td>
<td>-scalar-evolution</td>
</tr>
<tr>
<td>-loop-idiom</td>
<td>-loop-deletion</td>
<td>-loop-unroll</td>
<td>-memdep</td>
<td>-memcpyopt</td>
</tr>
<tr>
<td>-inline</td>
<td>-domtree</td>
<td>-strip-dead-p</td>
<td>-jump-threading</td>
<td>-loop-simplify</td>
</tr>
</tbody>
</table>

Table C.1.: Optimization -O1 passes

- **O2**: Has all O1 passes plus `-gvn` `-globaldce` `-constmerge`.
- **O3**: Has all O2 passes plus `-argpromotion`. "This pass promotes “by reference” arguments to be “by value” arguments. In practice, this means looking for internal functions that have pointer arguments"[31]. In O3 level OptimizeSize= false, UnitAtATime= true, UnrollLoops= true, SimplifyLibCalls= true, HaveExceptions= true, InliningPass).
- **O4**: The Clang driver translates it to -O3 and -flto. It will call Gold linker to do link time optimization.
- **O5, O6**: are similar to -O2

This command in prints passes:

```
echo "" | opt -OX -disable-output -debug-pass=Arguments
```

C.1. Mibench results

Here are results for some tests from Mibench. Test platform is Ubuntu12.04. Results show the size of `.text` section in executable program. Command below extracts `.text` size:

```
size -A XXX | grep .text | awk '{ print $2 }`
```
LLVM optimization options

Table C.2.: Mibench, llvm optimization option (code size)
Bibliography


Bibliography


[27] [Online]. Available: http://llvm.org/docs/LinkTimeOptimization.html


[31] [Online]. Available: http://llvm.org/docs/Passes.html