Code optimization in GCC

Sébastian Pop

Université Louis Pasteur
Strasbourg
FRANCE
Introduction

GCC : GNU Compiler Collection

- C, C++, Java, Ada, Fortran, Mercury, ...
Introduction

GCC : GNU Compiler Collection

- C, C++, Java, Ada, Frotran, Mercury, ...
- Generates code for 43 different architectures: i386, ia64, m68k, sparc, ...
Introduction

GCC : GNU Compiler Collection

- C, C++, Java, Ada, Fortran, Mercury, ...
- Generates code for 43 different architectures: i386, ia64, m68k, sparc, ...
- Main compiler in GNU world
Introduction

GCC : GNU Compiler Collection

- C, C++, Java, Ada, Fortran, Mercury, ...
- Generates code for 43 different architectures: i386, ia64, m68k, sparc, ...
- Main compiler in GNU world
- Apple’s system compiler.
Introduction

GCC: GNU Compiler Collection

- C, C++, Java, Ada, Fortran, Mercury, ...
- Generates code for 43 different architectures: i386, ia64, m68k, sparc, ...
- Main compiler in GNU world
- Apple’s system compiler.
- Industrial compiler.
Front-ends / back-end

GCC

gcc  g++  gcj  g77

Front-ends
Front-ends / back-end

GCC
- gcc
- g++
- gcj
- g77
- i386
- ia64
- m68k
- sparc

RTL

Front-ends

Machine Description

Back-ends
Exemple: cross-compilation

- Suppose that I want to generate Sparc code:
  
  -target=sparc
Exemple: cross-compilation

- Suppose that I want to generate Sparc code: 
  \[-\text{target} = \text{sparc}\]

- I build GCC on my laptop: \[-\text{build} = \text{i586}\]
Exemple: cross-compilation

- Suppose that I want to generate Sparc code:
  -target=sparc

- I build GCC on my laptop: -build=i586

- and I run the compiler on my laptop:
  -host=i586
Exemple: cross-compilation

- Suppose that I want to generate Sparc code:
  -target=sparc

- I build GCC on my laptop: 
  -build=i586

- and I run the compiler on my laptop:
  -host=i586

..gcc/configure -target=sparc -build=i586
-host=i586
Exemple: cross-compilation

 GCC
 gcc g++ gcj g77
 i386 ia64 m68k sparc

 RTL

 Front-ends

 Machine Description

 Back-end

 GAS

 Assembler
Exemple: cross-compilation

1. Select SPARC machine description

Front-ends

Machine Description

Back-end

GCC

gcc  g++  gcj  g77

SPARC specific
RTL

sparc

Code optimization in GCC – p.5
Exemple: cross-compilation

1. Select SPARC machine description
2. Compile

Front-ends
Machine Description
Back-end

SPARC specific RTL

SPARC assembler code

GCC

gcc  g++  gcj  g77

sparc
RTL Optimizations

- An optimization pass optimizes all front-ends.
RTL Optimizations

- An optimization pass optimizes all front-ends.
- Machine dependent optimizations.
RTL Optimizations

- An optimization pass optimizes all front-ends.
- Machine dependent optimizations.
- Types and memory structures after lowering to RTL contain less information.
  Memory accesses are under their canonical form: <start address + offset>
RTL Optimizations

- An optimization pass optimizes all front-ends.
- Machine dependent optimizations.
- Types and memory structures after lowering to RTL contain less information. Memory accesses are under their canonical form: <start address + offset>
- Idea: we’d like to have
An optimization pass optimizes all front-ends.

Machine dependent optimizations.

Types and memory structures after lowering to RTL contain less information. Memory accesses are under their canonical form: `<start address + offset>`

Idea: we’d like to have

architecture independent optimizations.
RTL Optimizations

- An optimization pass optimizes all front-ends.
- Machine dependent optimizations.
- Types and memory structures after lowering to RTL contain less information. Memory accesses are under their canonical form: `<start adress + offset>`

Idea: we’d like to have
- architecture independent optimizations.
- on high level representations.
Intermediate Representations

GCC
- gcc
- g++
- gcj
- g77

RTL

Translation follows machines specificities
Machine description
Intermediate Representations

Progressive transition from AST to RTL
Architecture independent IR

Machine description

Translation

GCC

gcc  g++  gcj  g77

Mid–RTL

RTL
Intermediate Representations

 GCC
  gcc    g++    gcj    g77
  Simplify

  Simple
  Translation

  Mid-RTL
  Machine description

  RTL

Imperative Normal Form
Language independent representation

Progressive transition from AST to RTL
Architecture independent IR
Abstract Syntax Trees

- Simple linked list for statement nodes.
Abstract Syntax Trees

- Simple linked list for statement nodes.
- Manipulation of nodes through a macro interface: `TREE_CHAIN`, `TREE_OPERAND`, `TREE_CODE`, ...
Abstract Syntax Trees

- Simple linked list for statement nodes.
- Manipulation of nodes through a macro interface: \texttt{TREE_CHAIN}, \texttt{TREE_OPERAND}, \texttt{TREE_CODE}, ... 
- Data structures hidden.
Abstract Syntax Trees

- Simple linked list for statement nodes.
- Manipulation of nodes through a macro interface: `TREE_CHAIN`, `TREE_OPERAND`, `TREE_CODE`, ...
- Data structures hidden.
- AST nodes are typed: allows tree-checking during development.
AST: example

\[
a = (\neg b) \ast 7;
\]
\[
x = y+z;
\]
AST: example

```c
a = (--b) * 7;
x = y+z;
```

Code optimization in GCC – p.9
AST: example

\[
a = (-b) * 7; \\
x = y + z;
\]
a = (--b) * 7;
x = y+z;
a = (–\(\neg b\)) \times 7;
x = y+z;
a = (--b) * 7;
\( x = y + z; \)
a = (−−b) * 7;
x = y+z;
Simple: overview

- SIMPLE’s grammar defines an imperative normal form:
Simple: overview

- SIMPLE’s grammar defines an imperative normal form:
  - Reduced number of expressions.
Simple: overview

- SIMPLE’s grammar defines an imperative normal form:
  - Reduced number of expressions.
  - Reduced number of control structures.
Simple: overview

- SIMPLE’s grammar defines an imperative normal form:
  - Reduced number of expressions.
  - Reduced number of control structures.
- SIMPLE AST has a regular structure.
Simple: overview

- SIMPLE’s grammar defines an imperative normal form:
  - Reduced number of expressions.
  - Reduced number of control structures.
- SIMPLE AST has a regular structure.
- Systematic AST analysis is possible.
Simple: overview

- SIMPLE’s grammar defines an imperative normal form:
  - Reduced number of expressions.
  - Reduced number of control structures.
- SIMPLE AST has a regular structure.
- Systematic AST analysis is possible.
- Common intermediate representation for all front ends.
Simple: exemple

\[ a = \text{--}b*7; \]
<table>
<thead>
<tr>
<th>a = --b*7;</th>
<th>b=b-1;</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=b*7;</td>
<td></td>
</tr>
</tbody>
</table>
Simple: exemple

\begin{align*}
a &= --b*7; \\
b &= b-1; \\
a &= b*7;
\end{align*}
### Simple: exemple

<table>
<thead>
<tr>
<th>a = --b*7;</th>
<th>b = b-1;</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b*7;</td>
<td>a = b*7;</td>
</tr>
</tbody>
</table>

```plaintext
if (i++ && --k)
{
    j = f(i+3*k);
}
```

```plaintext
if (i)
{
    k = k-1;
    if (k)
    {
        i = i+1;
        T1 = 3*k;
        T2 = i+T1;
        j = f(T2);
    }
    else
    {
        i = i+1;
    }
else
{
    i = i+1;
}
```
while(i++ && --k)
{
   A[i]=A[i+3*k];
}

Simple: exemple
```c
while(i++ && --k)
{
    A[i]=A[i+3*k];
}

if(i)
{
    k=k-1;
    if (k)
        while(1)
        {
            i=i+1;
            T1=3*k;
            T2=i+T1;
            A[i]=A[T2];
            if(i)
            {
                k=k-1;
                if(k)
                    i=i+1;
                else
                    break;
            }
            else
                break;
        }
    else
        break;
}
```

```c
if(i)
{
    k=k-1;
    if (k)
        while(1)
        {
            i=i+1;
            T1=3*k;
            T2=i+T1;
            A[i]=A[T2];
            if(i)
            {
                k=k-1;
                if(k)
                    i=i+1;
                else
                    break;
            }
            else
                break;
        }
}
```

An optimizing compiler

- Source code
- Front-end
  - Analyses
  - Optimizations

Diagram showing the process of a compiler, with source code flowing through the front-end to undergo analyses and optimizations.
An optimizing compiler

Source code

Front-end

Analyses
Call graph

Optimizations
Inlining
Recursivity suppression
Call Graph

(node, edge) => (declaration, call)
Call Graph

- (node, edge) \( \Rightarrow \) (declaration, call)
- Graph representation:
Call Graph

- (node, edge) => (declaration, call)
- Graph representation:
  - pointers: P-Space.
Call Graph

- (node, edge) => (declaration, call)
- Graph representation:
  - pointers: P-Space.
  - in a file under parenthesized form: EXP-Space.
Call Graph

- (node, edge) => (declaration, call)
- Graph representation:
  - pointers: P-Space.
  - in a file under parenthesized form: EXP-Space.
- Use metrics for controlling inlining.
Call Graph

- (node, edge) => (declaration, call)
- Graph representation:
  - pointers: P-Space.
  - in a file under parenthesized form: EXP-Space.
- Use metrics for controlling inlining.
- GCC’s analysis is limited to a single translation unit.
Call Graph: solution

- Perform call graph optimizations outside GCC.
Call Graph : solution

- Perform call graph optimizations outside GCC.

- Problems:
Call Graph : solution

- Perform call graph optimizations outside GCC.

- Problems:
  - Extract information, decide, then apply optimizations: 3 passes.
Call Graph : solution

- Perform call graph optimizations outside GCC.

- Problems :
  - Extract information, decide, then apply optimizations: 3 passes.
  - Knowledge base’s size.
Call Graph: solution

- Perform call graph optimizations outside GCC.

- Problems:
  - Extract information, decide, then apply optimizations: 3 passes.
  - Knowledge base’s size.
  - What informations to be stored in KB?
An optimizing compiler

Source code

Front-end

Analyses

Call graph

Optimizations

Inlining

Recursivity suppression
An optimizing compiler

Source code

Front-end

Analyses
- Call graph
- Control flow graph

Optimizations
- Inlining
- Recursivity suppression
- CFG normalization
CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.
CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.
- CFG normalization is based on Simple.
CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.
- CFG normalization is based on Simple.
- Why normalizing CFG?
CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.
- CFG normalization is based on Simple.
- Why normalizing CFG?
  - It is difficult to optimize programs containing gotos.
CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.
- CFG normalization is based on Simple.
- Why normalizing CFG?
  - It is difficult to optimize programs containing gotos.
  - Break and continue translation to RTL generates gotos.
CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.
- CFG normalization is based on Simple.
- Why normalizing CFG?
  - It is difficult to optimize programs containing gotos.
  - Break and continue translation to RTL generates gotos.
  - Simplification generates irregular code.
Flow Out

Loop:

<table>
<thead>
<tr>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop’s body</td>
</tr>
</tbody>
</table>

Code optimization in GCC – p.18
Flow Out

Loop:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Normal exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop’s body</td>
<td>Irregular exit</td>
</tr>
</tbody>
</table>

if (c)
break;
Flow Out

**Loop:**

<table>
<thead>
<tr>
<th>b_c &amp; Condition</th>
<th>Normal exit</th>
</tr>
</thead>
</table>

- **Loop’s body**

```
if (c) {b_c = true;
else {
    ...
```
Break Elimination

while (a)
{
    stmt1;
    if (b)
        break;
    stmt2;
}

Code optimization in GCC – p.19
int c_b = 0;
while (c_b == 0 && a)
{
    stmt1;
    if (b)
        {c_b = 1;}
    else
        {
            stmt2;
        }
}
Goto Elimination
Goto Elimination

goto

label
Goto Elimination
Goto Elimination
Goto Elimination
An optimizing compiler

Source code

Front-end

Analyses
Call graph
Control flow graph

Optimizations
Inlining
Recursivity suppression
CFG normalization
An optimizing compiler

Source code

Front-end

Analyses
- Call graph
- Control flow graph

Optimizations
- Inlining
- Recursivity suppression
- CFG normalization
- Loop unrolling / blocking / fusion ...
- Spatial / temporal locality
An optimizing compiler

Source code

Front-end

Analyses
- Call graph
- Control flow graph
- SSA
- Induction variables
- Array access functions
- Pointers and alias analysis
- Dependence analysis

Optimizations
- Inlining
- Recursivity suppression
- CFG normalization
- Loop unrolling / blocking / fusion ...
- Spatial / temporal locality
Loop Optimizations

- Loops are normalized after detection of induction variables.

Geometric representation of array accesses can be then constructed. Dependence analysis is necessary for validating loop transformations. These points are still under development.
Loop Optimizations

- Loops are normalized after detection of induction variables.
- Geometric representation of array accesses can be then constructed.
Loop Optimizations

- Loops are normalized after detection of induction variables.
- Geometric representation of array accesses can be then constructed.
- Dependence analysis is necessary for validating loop transformations.
Loop Optimizations

- Loops are normalized after detection of induction variables.
- Geometric representation of array accesses can be then constructed.
- Dependence analysis is necessary for validating loop transformations.
- These points are still under development.
An optimizing compiler

Source code

Front-end

Analyses
- Call graph
- Control flow graph
- SSA
- Induction variables
- Array access functions
- Pointers and alias analysis
- Dependence analysis

Optimizations
- Inlining
- Recursivity suppression
- CFG normalization
- Loop unrolling / blocking / fusion ...
- Spatial / temporal locality
An optimizing compiler

**Source code**

**Front-end**

**Analyses**
- Call graph
- Control flow graph
- SSA
- Induction variables
- Array access functions
- Pointers and alias analysis
- Dependence analysis

**Optimized code**

**Unparser**

**Optimizations**
- Inlining
- Recursivity suppression
- CFG normalization
- Loop unrolling / blocking / fusion ...
- Spatial / temporal locality
Remerciements

Merci à tous ceux qui ont contribué à la réussite de ce projet :
Remerciements

Merci à tous ceux qui ont contribué à la réussite de ce projet :

- l’équipe ICPS pour l’excellente ambiance,
- Philippe Clauss, Vincent Loechner et Benoît Meister pour leur travail de recherche,
- Catherine Mongenet pour le cours de compil,
- Frédéric Wagner et Diego Novillo pour m’avoir accompagné dans ce projet,
- the FSF for GCC, Debian, Linux and Prosper
- et ma famille.