Parallel GCC: Status and Expectations

Giuliano Belinassi

giulianob

September 13, 2019

GNU Cauldron 2019
FLUSP
Computer Science Department
Institute of Mathematics and Statistics
University of São Paulo (USP)

Introduction

- **FLUSP – Floss at USP**
  - Student extension group
  - Aims to Contribute to FLOSS
  - Projects we currently contribute: Linux Kernel, GCC, Git, Debian
  - Promote Contribution Events, such as the KernelDevDay

Overview

1. Introduction
2. Where to Start?
3. Parallel Architecture
4. Results
5. TODOs
6. References

Overview

1. Introduction
2. Where to Start?
3. Parallel Architecture
4. Results
5. TODOs
6. References

Introduction

- **How many of you have seen a compiler running in parallel?**

Introduction

- **Parallelization of GCC Internals**
  - Exploring parallelism in a single file

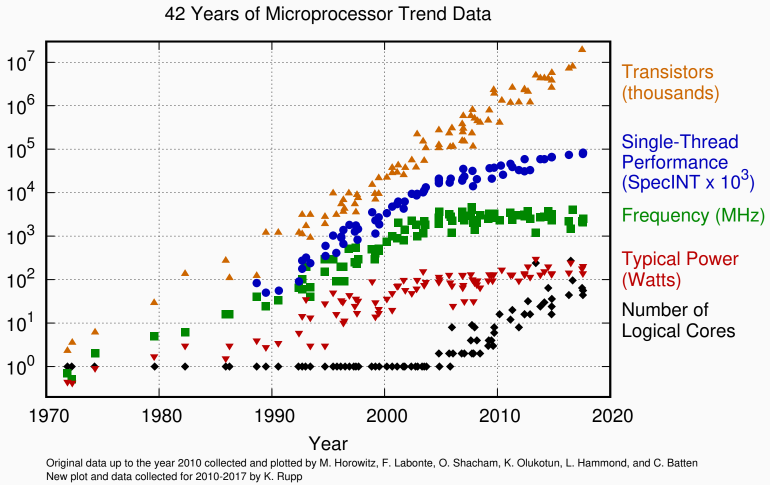
- **Main objectives**
  - Reduction of compilation time
  - Highlight GCC global states

Motivation

- **Compilers are really complex programs**
  - GCC dates from 1987
  - Still sequential
- **Automatic Generation of big files**
  - `gimple-match.c`: 100358 lines of C++ (GCC 10.0.0)
- **Exponencial growth in number of cores of a CPU**

Motivation

Growth of Computational Power (Rupp, 2018)

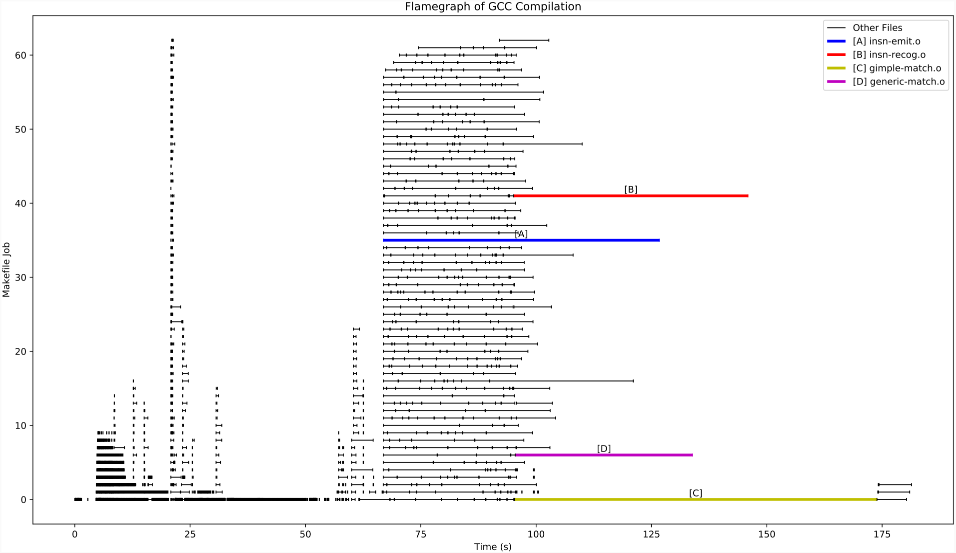


Motivation

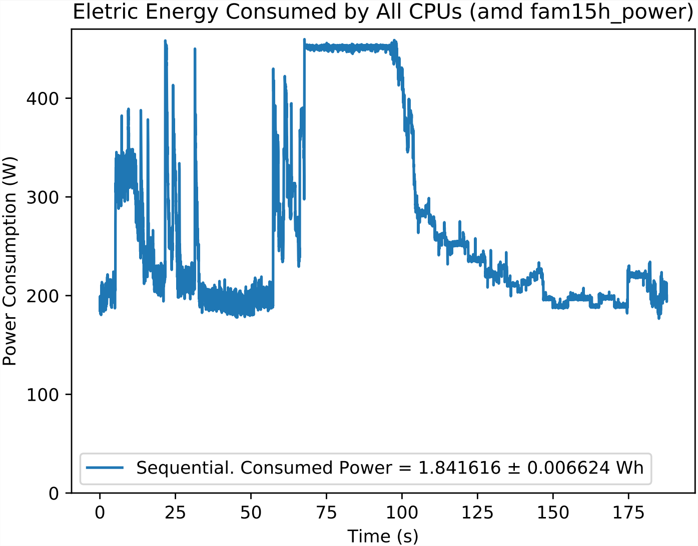
- **Where can we use these parallel processors in a compiler?**
- **How much the improvement is?**
- **Is there projects that can be benefited from this?**

Introduction

- **Experiment 1**
  - GCC compilation on a machine with $4\times$ AMD Opteron 6376
    - $4 \times 16 = 64$ *cores*
  - No *bootstrap* (`--disable-bootstrap`)
  - `$ make -j64`
  - Collected Compilation Time of each file
  - Collected Consumed Energy of all CPUs



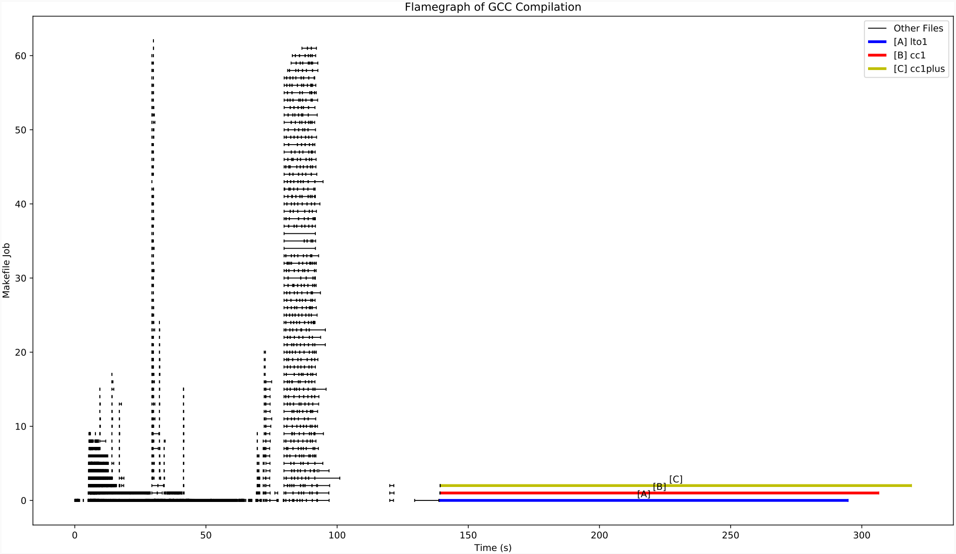
Flamegraph of GCC Compilation
Other Files
[A] insn-emit.o
[B] insn-recog.o
[C] gimple-match.o
[D] generic-match.o
[A]
[B]
[C]
[D]
Makefile Job
Time (s)
0
10
20
30
40
50
60
0
25
50
75
100
125
150
175


Eletric Energy Consumed by All CPUs (amd fam15h_power)
Power Consumption (W)
0
100
200
300
400
0
25
50
75
100
125
150
175
Time (s)
Sequential. Consumed Power = 1.841616 ± 0.006624 Wh

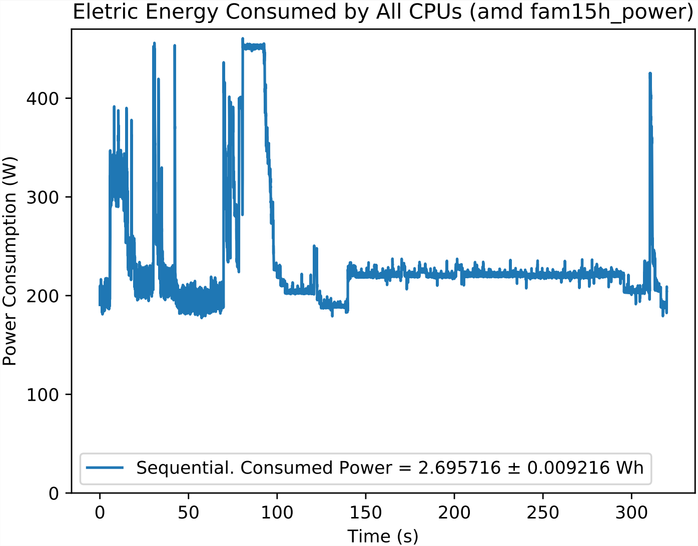
Introduction

- **How about LTO?**

- **Experiment 2**
  - GCC compilation on a machine with $4\times$ AMD Opteron 6376
    - » $4 \times 16 = 64$ *cores*
  - No *bootstrap* (`--disable-bootstrap`)
  - Enabled LTO (`CFLAGS=-flto=64 CXXFLAGS=-flto=64`)
  - `$ make -j64`
  - Collected Compilation Time of each file
  - Collected Consumed Energy of all CPUs



Flamegraph of GCC Compilation
Other Files
[A] lto1
[B] cc1
[C] cc1plus
[A]
[B]
[C]
Makefile Job
0
10
20
30
40
50
60
Time (s)
0
50
100
150
200
250
300



Eletric Energy Consumed by All CPUs (amd fam15h_power)
Power Consumption (W)
0
100
200
300
400
0
50
100
150
200
250
300
Time (s)
Sequential. Consumed Power = 2.695716 ± 0.009216 Wh

Introduction

- **There is a parallelism bottleneck in GCC project**
  - And the compilation uses more electrical power as a consequence

- **Could we improve it by parallelizing GCC?**

Overview

1. Introduction
2. **Where to Start?**
3. Parallel Architecture
4. Results
5. TODOs
6. References

Where to Start?

GCC structure is divided in three parts.

- ***Front End***
  - Parsing
- ***Middle End***
  - Hardware-Independent Optimization
- ***Back End***
  - Hardware-Dependent Optimization
  - Code Generation

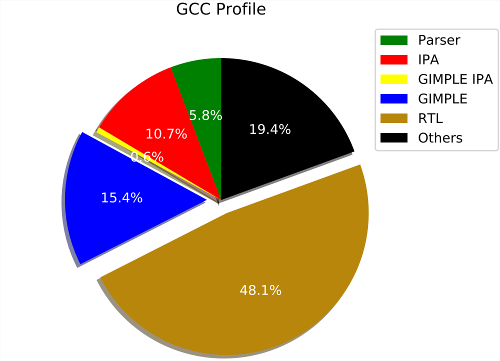
Where to Start?

- **Selected Part: Middle End**
  - Applies Hardware-Independent Optimizations in the code
  - Why? Because it looked easier to start rather than RTL

- **Optimization can be broken into two disjoint sets**
  - Intra Procedural
    - » *Can be applied into a function without observing the interactions with other functions*
  - Inter Procedural
    - » *Interaction with other functions must be considered*
    - » *GCC calls this Inter Process Analysis (IPA)*

GCC Profile

- **We used `gimple-match.c` to profile the compiler**
  - Intel Core i5-8250U (4 cores)
  - 5400RPM 1TB Hard Drive



GCC Profile

- **We used `gimple-match.c` to profile the compiler**
  - 48s is spent compiling this file
  - 63% is spent in Intra Procedural Optimization
  - This can be parallelized
  - Maximum speedup: $2.7\times$ according to Amdahl's Law

Overview

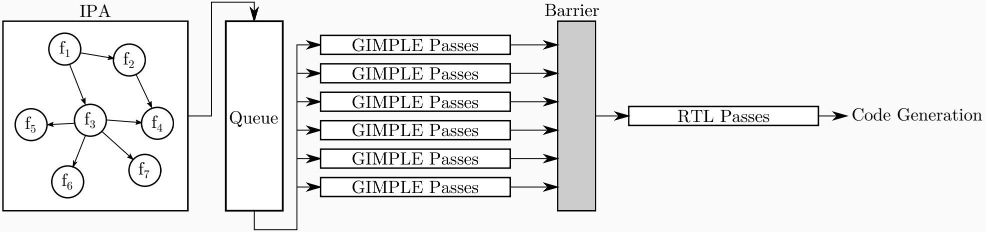
1. Introduction
2. Where to Start?
3. **Parallel Architecture**
4. Results
5. TODOs
6. References

Parallel Architecture

- **Parallelize Intra Procedural Optimizations**
  - By per-function
  - Similarly to Wortman and Junkin (1992)
  - With arbitrary number of threads

Parallel Architecture

- **Architecture of Parallel GCC**
  - The Inter Process Analysis inserts functions into a Producer Consumer Queue
  - Each thread is responsible to remove their work from the Queue
  - When Queue is empty, threads are blocked until work is inserted into it
  - The threads die when the `EMPTY` token is inserted
  - Measured overhead: 0.1s for 2000 functions



Parallel Architecture

- **Advantages**
  - Best candidate to Linear speedup
  - Worst case: A single big function
- **Disadvantage**
  - Map per-pass global states in the compiler

Implementation

- **Split `cgraph_node::expand` into three methods:**
  1. `expand_ipa_gimple`
  2. `expand_gimple` ⟵ Parallelization effort in GSoC 2019
  3. `expand_rtl`
- **Serialized the Garbage Collector**
- **Serialized memory related structures**

Implementation

Memory Pools:

- **Problems:**
  - Linked List of several object instances
  - Can be allocated/released upon request
  - Most race conditions were there
- **Solution:**
  - Create an instance for every thread
  - Merge the memory pools when all threads join
  - Implementing merge feature: Just append the two Linked List

Implementation

Garbage Collection:

- **Problem:**
  - GCC implements a garbage collector
  - Variables can be marked to be watched by it
  - Collection can also be done upon request
- **Partial Solution:**
  - Serialize the entire Garbage Collector
  - Disable collection between passes
- **Research is needed to:**
  - Map the race conditions in the Garbage Collector
  - Make it thread safe

Implementation

`rtl_data` Structure:

- **Problem:**
  - This class represents the current function being compiled in RTL
  - GCC only has a single instance of this class
  - GIMPLE uses it to decide optimization related to instruction costs
- **Solution**
  - Have one copy of this structure for each thread
  - Make GIMPLE not rely on this?

Implementation

tree-ssa.address.c: `mem_addr_template_list`

- **Problem:**
  - Race condition in this structure
- **Partial Solution:**
  - Serialize with a mutex
- **Solution**
  - Replicate for each thread?

Implementation

Integer to Tree Node hash

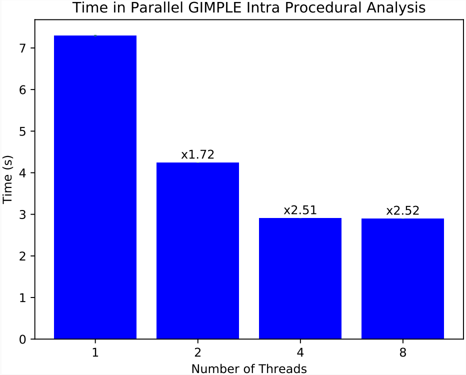
- **Problem:**
  - Race condition in this structure
- **Partial Solution:**
  - Serialize with a mutex
- **Solution**
  - Replicate for each thread?

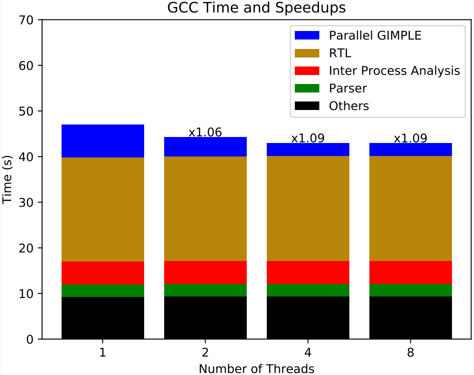
Overview

1. Introduction
2. Where to Start?
3. Parallel Architecture
4. Results
5. TODOs
6. References

Results

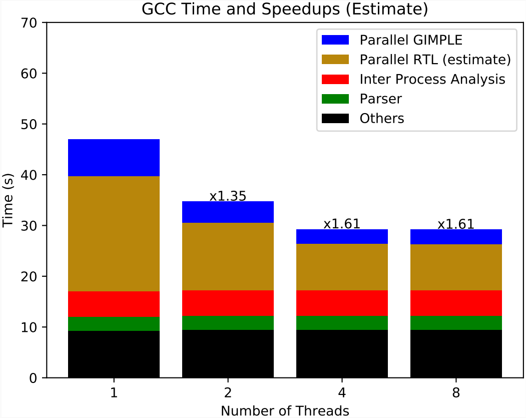
- **Results by parallelizing GIMPLE**
- **Mean of 30 samples**





Results

- The same approach can be used to parallelize RTL
- Using the GIMPLE results



Motivation

- **Where can we use these parallel processors in a compiler?**
  - Intra Process Analysis is a easy answer
- **How much the improvement is?**
  - Without much optimization, $1.6\times$ using 4 threads
  - Better results will require more effort
- **Is there projects that can be benefited from this?**
  - GCC
  - ...What else?

Overview

1. Introduction
2. Where to Start?
3. Parallel Architecture
4. Results
5. **TODOs**
6. References

TODOs

- **What is left to do:**
  - Fix all race conditions in GIMPLE
  - Fix all C11 `__thread` occurrences
    - *Initialize Inter-Pass objects at* `::execute` *time*
    - *Per-thread attributes initialized at spawn time*
  - Make this build pass the testsuite
  - Parallelize RTL
  - Parallelize IPA (useful for LTO?)
  - Communicate with Make for automatic threading

Overview

1. Introduction
2. Where to Start?
3. Parallel Architecture
4. Results
5. TODOs
6. **References**

References i

- Karl Rupp (2018). *42 Years of Microprocessor Trend Data.* URL: https://github.com/karlrupp/microprocessor-trend-data.
- D.B. Wortman and M.D. Junkin (Jan. 1992). "A Concurrent Compiler for Modula-2+". In: 27, pp. 68–81. DOI: 10.1145/143103.120025.

Repositories

1. Introduction
2. Where to Start?
3. Parallel Architecture
4. Results
5. TODOs
6. References

https://github.com/giulianobelinassi/gcc-timer-analysis

https://gitlab.com/flusp/gcc/tree/giulianob_parallel

https://gcc.gnu.org/wiki/ParallelGcc

Thank you!