Advanced Topics in Compilers

Gino

Simone Campanoni
simone.campanoni@northwestern.edu
Outline

• Introducing Gino

• Gino’s compilation pipeline

• Debugging Gino
Gino

• Gino is a parallelizing compiler for LLVM IR

• Standalone codebase
  https://github.com/arcana-lab/gino

• To compile it
  1. Compile and install NOELLE
  2. Source NOELLE/enable
  3. Go to Gino’s codebase and compile Gino
     cd gino ; make
A typical parallelizing compiler

Parallelizing compiler

Front-end

IR

Parallelizer

Middle-end

Parallel IR

Back-end
A typical parallelizing compiler

Source code → Front-end → IR → Identify potential parallelism → IR → Mapping parallelism onto the target architecture → IR → Optimizing parallel code → Back-end

Parallelizing compiler

- Front-end
  - IR
  - Identify potential parallelism
- Middle-end
  - IR
  - Mapping parallelism onto the target architecture
  - Parallelism enablers
  - Memory alias analysis
- Back-end
  - Parallel IR
  - Optimizing parallel code
  - Parallelizer
  - Parallelization technique

Parallelism enablers

Memory alias analysis

Parallelizer

Parallelization technique
Gino: the parallelizing compiler upon NOELLE
Gino: the parallelizing compiler upon NOELLE

Source code

Front-end

IR

Gino

Identify potential parallelism

Parallelism enablers

Parallelization planner

Parallelizer

Parallelization technique

Parallel IR

Middle-end

Mapping parallelism onto the target architecture

Optimizing parallel code

Back-end

IR

NOELLE

PDG generator and SCC analysis

Abs

Profilers

Memory alias analysis

Alias analysis frameworks (SCAF, SVF, LLVM)

Synchronizing data structures

Task engine

VIRGIL

Parallel IR
Outline

• Introducing Gino

• Gino’s compilation pipeline

• Debugging Gino
Compilation pipeline

• Let’s assume test.cpp is the whole program
  (otherwise, if multiple source files exist, then
  use gclang if you run commands manually
  or use NOELLEGym to automate everything)
Gino: the parallelizing compiler upon NOELLE

**Source code** → **Front-end** → **IR** → **Gino** → **Middle-end** → **Parallelization planner** → **Parallelizer** → **Back-end** → **Parallel IR**

**Gino** consists of:
- **Parallelism enablers**
- **Parallelization techniques**
- **Parallelization planner**
- **Parallelizer**
- **Task engine**
- **Synchronizing data structures**
- **Profiler**
- **Utils**
- **PDG generator** and SCC analysis
- **Abs**
- **Memory alias analysis**
- **Alias analysis frameworks (SCAF, SVF, LLVM)**

**NOELLE** provides:
- **Memory alias analysis**
- **PDG generator and SCC analysis**
- **Profiler**
- **Utils**
- **Task engine**
- **Synchronizing data structures**

**VIRGIL** provides:
- **Memory alias analysis**
- **PDG generator and SCC analysis**
- **Profiler**
- **Utils**
- **Task engine**
- **Synchronizing data structures**
Gino: the parallelizing compiler upon NOELLE

Identify potential parallelism

Paralllelism enablers

Memory alias analysis

PDG generator and SCC analysis

Profilers

Alias analysis frameworks (SCAF, SVF, LLVM)
Compilation pipeline

1. Let’s assume `test.cpp` is the whole program
   
   ```
   clang -O1 -Xclang -disable-llvm-passes -c -emit-llvm test.cpp -o test.bc
   noelle-simplification test.bc -o test.bc
   ```

2. Now we need to profile the code to identify hot code
   
   ```
   noelle-prof-coverage test.bc baseline_with_runtime_prof -lm -lstdc++ -lpthread
   $ ./baseline_with_runtime_prof 10 20 30
   432500
   
   default.profraw
   ```

   ```
   $ noelle-meta-prof-embed default.profraw test.bc -o test_with_profile.bc
   opt -pgo-test-profile-file=/tmp/tmp.X3krDBb9S4 -block-freq -pgo-instr-use test.bc -o test_with_profile.bc
   ```

3. Now we need to make the IR more amenable for parallelization
   
   ```
   gino-pre test_with_profile.bc -noelle-verbose=2 -noelle-min-hot=1
   ```
Gino: the parallelizing compiler upon NOELLE

Identify potential parallelism

Gino

Parallelism enablers

NOELLE

Abs

PDG generator and SCC analysis

Profilers

Memory alias analysis

Alias analysis frameworks (SCAF, SVF, LLVM)
Gino: the parallelizing compiler upon NOELLE

Source code → Front-end

Front-end

IR → Identify potential parallelism

Gino

IR → Mapping parallelism onto the target architecture

Parallelizer

Optimizing parallel code → Back-end

Parallel IR

NOELLE

Memory alias analysis

Abs

PDG generator and SCC analysis

Parallelism enablers

Gino

Profilers

Parallelization planner

Parallelization technique

Parallelizer

Utils

Synchronizing data structures

Task engine

VIRGIL

Parallelism enablers

Alias analysis frameworks (SCAF, SVF, LLVM)
Gino: the parallelizing compiler upon NOELLE

Understanding Parallelism

Mapping parallelism onto the target architecture

Parallelization planner

Parallelizer

Parallelization technique

PDG generator

Profilers

Utils

Synchronizing data structures

Task engine

Abs

Memory alias analysis

PDG generator and SCC analysis

Alias analysis frameworks (SCAF, SVF, LLVM)

Synchronizing data structures

Task engine

VIRGIL
Compilation pipeline

4. We need to profile the code

```bash
noelle-prof-coverage test_with_profile.bc baseline_with_runtime_prof -lm -lstdc++ -lpthread
```

```bash
$ ./baseline_with_runtime_prof 10 20 30
432500
```

```bash
default.profraw
```

```bash
noelle-meta-prof-embed default.profraw test_with_profile.bc -o test_with_new_profile.bc
```

5. Now we need to compute the PDG and embed it into the IR

```bash
noelle-meta-pdg-embed test_with_new_profile.bc -o code_to_parallelize.bc
```

6. Now we can parallelize the IR

```bash
gino code_to_parallelize.bc -o parallelized_code.bc
```

7. Now we can generate the parallelized binary

```bash
clang++ parallelized_code.bc -pthread -O3 -lm -lstdc++ -lpthread -o parallel_binary
```
Gino: the parallelizing compiler upon NOELLE
Gino: the parallelizing compiler upon NOELLE

Source code → Front-end → IR
- Identify potential parallelism
- Memory alias analysis
- PDG generator and SCC analysis

Middle-end
- Mapping parallelism onto the target architecture
- Parallelization planner
- Profilers
- NOELLE
- Abs
- Parallelism enablers
- Virgil

Parallel IR → Back-end → Parallelization technique → Parallelizer → Optimizing parallel code
- Synchronizing data structures
- Task engine
- Profilers
- PDG generator and SCC analysis
- Memory alias analysis
- Alias analysis frameworks (SCAF, SVF, LLVM)
- Synchronizing data structures
Outline

• Introducing Gino
• Gino’s compilation pipeline
• Debugging Gino
Developing and testing

• Let’s say you are working to improve Gino or NOELLE (e.g., induction variable detection algorithm)

• You need to test the correctness and impact of your work. ...
  • Gino can help you to do it
Testing

• NOELLE includes tests for its code transformations (e.g., code parallelization, loop-invariant code motion, etc...)

Performance (of the generated binary) tests
Regression tests
Testing

• NOELLE includes tests for its code transformations (e.g., code parallelization, loop-invariant code motion, etc...)

doc
enable
install
LICENSE
Makefile
README.md
src
tests

If you don’t have condor installed in your platform

If you have condor installed in your platform

It runs the transformations only using their default configurations (e.g., unroll-factor set to be the default one)

It generates condor files to run in parallel all transformations with many different configurations (generating more than 30,000 tests that all run in parallel)
Testing with condor

cd tests ; make condor

doc
enable
install
LICENSE
Makefile
README.md
src
tests

condor
Makefile
performance
regression
scripts

... regression_65
... regression_66
... regression_67
... regression_68
... regression_69
... regression_70
... regression_71
... regression_72
... regression_73
... regression_74
... regression_75
... regression_76
... regression_77

copy of the original regression dir
one directory per configuration for
the code transformations

All these tests
(~30,000 at the moment)
run in parallel!
Testing with condor

cd tests ; make condor

cd tests ; make condor_check

```bash
$ make condor_check
./scripts/condor_check.sh :
# REGRESSION TESTS:
Checking the regression test results
There are 21204 jobs that are still running
No new tests failed so far
There are new tests that now pass for all configurations. They are the next ones:

- Chunking
- DSWPIterations_RemovableIntraIterMemEdge
- Exit call2
- Exit call3
- IndependentIterations11
- IndependentIterations5
- LIOM
- LIOM_2
- Multiloops
- Multiloops_list
- ReductionIterationsAnd
- ReductionIterationsOr

# UNIT TESTS:
They are still running

# PERFORMANCE TESTS:
They are still running
```
Testing with condor

cd tests ; make condor

condor
Makefile
performance
regression
scripts

Tests that completed successfully get automatically deleted
Directory of a test that failed is kept (so you can debug it; check compiler_output.txt) and a script to reproduce the fail is automatically generated

To reproduce the fail:
• Go to the directory of the test (e.g., cd regression_4/Simple)
• Run ./run_me.sh
Re-run the tests using condor

cd tests;

1. Make sure no tests are still running
   condor_q `whoami`

2. Clean the tests directory
   make clean

3. Run the tests
   make condor
Running a single test without condor

cd tests ; make download

1. Go to the test directory  
   (e.g., cd regression/Simple)
2. Clean the directory  
   make clean
3. Enable NOELLE and Gino binaries  
   in your environment  
   source ../../../enable  
   source WHERE_NOELLE_IS/enable
4. Run the test  
   make test_correctness
5. Check the output  
   (look at the makefile to understand the scripts)
Notes about improving Gino or NOELLE
Typical flow

1. The parallelizer in the master branch works, but you want to improve the speedup obtained by it for a given benchmark
   • Let’s assume you are using NOELLEGym

2. You extend/modify a code analysis/transformation in the parallelizing pipeline described in these slides
   • To do so, you modify something in NOELLEGym/NOELLE/src, and then you recompile and install NOELLE

3. You re-run the parallelizer and the new parallel binary generated doesn’t work (e.g., seg fault)

   How should you debug it?
An approach to debug a loop-based parallelizing compiler

Assumption: the bug fit the common case, which is about parallelizing a given loop (independent on what other loops are parallelized)

1. **Shrinking:**
   Identify a single loop that its parallelization (when using the new changes) leads to the bug

2. **Comparing:**
   Use master to parallelize that single loop.
   Check the differences (compiler output and then the IR) of the parallelization between master and the changes.

3. **Correctness checking:**
   Deep analysis on the difference in parallelization that is incorrect (by manually checking why that parallelization aspect that differ is incorrect)
An approach to debug a loop-based parallelizing compiler

1. Shrinking

    gino code_to_parallelize.bc -o parallelized_code.bc
An approach to debug a loop-based parallelizing compiler

1. Shrinking

Loops of the program that satisfies the options given as input to NOELLE

Loops selected by the planner

Loops parallelized
An approach to debug a loop-based parallelizing compiler

1. Shrinking

$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll
An approach to debug a loop-based parallelizing compiler

1. **Shrinking**

Loops selected by the planner

Loops parallelized

$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll
An approach to debug a loop-based parallelizing compiler

1. Shrinking

$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll
An approach to debug a loop-based parallelizing compiler

1. Shrinking

Remove looporder for a few at a times (e.g., binary search)

Then, compile and run a given version of `code_to_parallelize.ll` that has a subset (or one) loop with the looporder metadata.

$ llvm-dis code_to_parallelize.bc
$ vim code_to_parallelize.ll
An approach to debug a loop-based parallelizing compiler

1. Shrinking

Remove looporder for a few at a times

Then, compile and run a given version of code_to_parallelize.ll that has a subset (or one) loop with the looporder metadata
An approach to debug a loop-based parallelizing compiler

1. Shrinking

Remove looporder for a few at a times

Then, compile and run a given version of code_to_parallelize.ll that has a subset (or one) loop with the looporder metadata
An approach to debug a loop-based parallelizing compiler

1. Shrinking

Remove looporder for a few at a times

Then, compile and run a given version of code_to_parallelize.ll that has a subset (or one) loop with the looporder metadata

```
clang++ parallelized_code.bc -pthreads -O3 -lm -lstdc++ -lpthread -o parallel_binary
```
An approach to debug a loop-based parallelizing compiler

1. **Shrinking**
   As soon as you found the bad loop, go to step 2
An approach to debug a loop-based parallelizing compiler

1. **Shrinking:**
   Identify a single loop that its parallelization (when using the new changes) leads to the bug

2. **Comparing:**
   Use master to parallelize that single loop.
   Check the differences (compiler output and then the IR) of the parallelization between master and the changes.
An approach to debug a loop-based parallelizing compiler

Assumption: the bug fit the common case, which is about parallelizing a given loop (independent on what other loops are parallelized)

1. **Shrinking:**
   Identify a single loop that its parallelization (when using the new changes) leads to the bug

2. **Comparing:**
   Use master to parallelize that single loop.
   Check the differences (compiler output and then the IR) of the parallelization between master and the changes.

3. **Correctness checking:**
   Deep analysis on the difference in parallelization that is incorrect (by manually checking why that parallelization aspect that differ is incorrect)
Always have faith in your ability

Success will come your way eventually

Best of luck!