Lecturers:

- David Chisnall (Univ. of Cambridge)
- Hal Finkel (Argonne National Lab.)
- Tobias Grosser (ETH Zurich)

Lab assistants:

- Michael Kruse (ENS Paris)
- Gábor Horvath (Ericsson)
- Pierrick Brunet (Quarkslab)
- Serge Guelton (Quarkslab)
- André Guinet (Quarkslab)
- Juan Martinez (Quarkslab)

Organisation:

- Sid Touati (Univ. of Côte d'Azur)

1. Introduction to LLVM

Lecturer: David Chisnall

Biography: David Chisnall is a Senior Research Associate at the University of Cambridge. He teaches the MPhil Modern Compiler Design course and works on hardware-software co-design with an emphasis on security and reliability. His main research focus is safe interoperability between programming languages. As part of this work, he maintains a research fork of Clang and LLVM that add support for hardware-enforced spacial memory safety for C-like languages. He has been a contributor to the LLVM project since 2008 and is the maintainer of the most widely deployed open source Objective-C implementation. His other open source activities include being a member of the FreeBSD Core Team for 4 years and the author of the C++ runtime shipped with FreeBSD (and recent Playstation iterations). David Chisnall received a BSc in Computer Science from the University of Wales, Swansea in 2003 and a PhD in Computer Science from Swansea University in 2008.

Course overview:

This course will cover the design decisions involved in designing a modern compiler intermediate representation, with a specific focus on the design decisions made by
LLVM IR and the affects that these have on the design of a compiler. We will explore the structure of the LLVM optimization pipeline, the relationship between analysis and transformation to produce optimization.

The course will investigate the tradeoffs between ahead-of-time (AoT) and just-in-time (JIT) compilation, in particular with regard to feedback-driven optimization. We will use a simple language incorporating an interpreter and LLVM-based compiler as a case study, with exercises to extend this language and explore the different execution modes.

The course has the following aims for students:

- To understand modern compiler intermediate representations, including SSA form
- To understand the structure of a modern compiler pipeline
- To gain practical experience generating and transforming LLVM IR

**Bibliography (background reading):**

[http://www.cl.cam.ac.uk/~dc552/L25.html](http://www.cl.cam.ac.uk/~dc552/L25.html)

**Lab overview:**

- Two sets of exercises:
  
  - The first one will work with an LLVM IR pass, using the SimplePass skeleton (https://github.com/CompilerTeaching/SimplePass) to extend in two directions:

    1. To investigate the number of instructions that occur in each basic block. I'd expect the students to test the heuristic that there's a branch roughly every 7 instructions in typical code by modifying the pass to perform this analysis and then compiling some of either their own code or code from projects that they use with it. This is the first exercise that I set in the MPhil course. It will give them some familiarity with LLVM IR and help build their intuition about the structure of real code once it’s been compiled.

    2. To modify the pass to produce a transformation that will memoise a function call. This provides an example of a transformation that is useful only if you know something about the library that you’re using. The result should cache the result of a call to an expensive function, so that the second call to the function is omitted. This gives them experience performing a transformation and gives them a bit more practice with the IR.
The second set of exercises will build on my MysoreScript toy language (https://github.com/CompilerTeaching/MysoreScript):  

1. Add C-like for loop and do-while loop to the language. This will involve extending the grammar, adding new AST nodes, extending the interpreter, and finally adding compiler support. This gives them some experience with all of the steps in the front end for a simple language.

2. Add hot-loop transfer. If a loop has executed more than 10 times in the interpreter, compile it. This can be done fairly simply in MysoreScript by constructing a closure that contains the loop, compiling it, and then calling the compiled closure. This gives them some insight into the performance tradeoffs between interpretation and compilation for a simple language, which should apply to most DSLs.

2. Code Transformation and Analysis Using Clang and LLVM

Lecturer: Hal Finkel

Biography: Hal Finkel graduated from Yale University in 2011 with a Ph.D. in theoretical physics focusing on numerical simulation of early-universe cosmology. He’s now the Lead for Compiler Technology and Programming Languages at the ALCF where he helps develop the Hardware/Hybrid Accelerated Cosmology Code (HACC), a two-time IEEE/ACM Gordon Bell Prize finalist. Hal has designed and implemented a tree-based force evaluation scheme and the I/O subsystem and contributed to many other HACC components. He also has contributed to the LLVM compiler infrastructure project for many years and is currently the code owner of the PowerPC backend and the pointer-aliasing-analysis subsystem, among others. He is the lead developer on the bgclang project, which provides LLVM/Clang on the IBM Blue Gene/Q supercomputer, and represents Argonne on the C++ Standards Committee.

Course overview: This series of lectures will cover code transformation and analysis using components of the LLVM compiler infrastructure [1]. LLVM’s C/C++ frontend, Clang [2], supports not only compiling source code for execution (i.e. transforming it into LLVM’s intermediate representation (IR)), but also features a powerful source-level static analysis framework [3]. This can be coupled with Clang’s rewriting and tooling functionalities [4] to create sophisticated source-to-source transformation tools.

For some use cases, runtime checking must supplement static reasoning. In some cases, for example, Clang’s undefined-behavior sanitizer [5], these checks much be inserted very early in the code-generation process. In other cases, such as the address [6] and thread sanitizers [7], the checks can be inserted after the code undergoes optimizing transformations. Runtime checks are associated with corresponding runtime-library functionality in LLVM’s compiler-rt project [8].
At the conclusion of the lecture series, students will understand Clang's static-analysis, rewriting, and tooling infrastructures well enough to create novel analyses for the stand-alone analyzer, analysis-based warnings for regular compilation, and source-to-source rewriting tools. Students will understand how Clang's undefined-behavior sanitizer works and how Clang's code-generation can be extended to create runtime checks. Finally, students will understand how the address and thread sanitizers work, both the IR-level transformations and the runtime-library components. Students will be able to create their own tools based on this model.

Bibliography:


Lecturer: Tobias Grosser

Biography: Tobias Grosser is a senior researcher in the Scalable Parallel Computing Laboratory (SPCL) of Torsten Hoefler at the Computer Science Department of ETH Zürich. Supported by a Google PhD Fellowship he received his doctoral degree from Universite Pierre et Marie Curie under the supervision of Albert Cohen. Tobias’ research is taking place at the border of low-level compilers and high-level program transformations with the goal of enabling complex - but highly-beneficial – program transformations in a production compiler environment. He develops with the Polly loop optimizer a loop transformation framework which today is a community project supported through the Polly Labs research laboratory. Tobias also developed advanced tiling schemes for the efficient execution of iterated stencils. Today Tobias leads the heterogeneous compute efforts in the Swiss University funded ComPASC project and is about to start a three year NSF Ambizione project on advancing automatic compilation and heterogenization techniques at ETH Zurich.

Course overview:

The generation of parallel code is important for the fast execution of classical high-performance applications such as weather prediction, but also modern applications such as image processing, machine learning, and biology simulations. The LLVM compiler infrastructure enables the automatic introduction and generation of parallel code through SIMDization, automatic parallelization, as well as automatic GPU code, generation. In this course, we learn about the fundamental building blocks that enable the generation and optimization of parallel program code. Starting off from learning about the SIMD instruction set extensions of LLVM we learn how to write our own SIMD accelerated vector code generator that can generate fast vector
code for all LLVM supported architectures. We then look into different approaches to model parallelism at the source language level, at the compiler IR level, and -- using Polly -- how to model parallelism with an abstract geometric representation based on integer polyhedra. Using these representations we discuss how parallelism information can be derived, how transformations to expose parallelism can be applied, and finally how fast parallel code can be generated. In the last part of this course, we discuss GPU code generation and learn how LLVM can be used to generate GPU accelerated code for AMD and NVIDIA systems, discuss the available CUDA and OpenCL extensions, and learn how Polly ACC can fully automatically perform GPU offloading. We conclude with an overview how these techniques allow for the automatic optimization of high-level languages such as Julia.

Bibliography:

OpenMP: http://www.openmp.org/specifications/
OpenCL: https://www.khronos.org/opencl/
CUDA: http://docs.nvidia.com/cuda/#axzz4UgqQh7F5
LLVM: http://www.llvm.org
Polly: http://polly.llvm.org
POCL: http://portablecl.org/docs/html/
Polyhedral Background: http://www.polyhedral.info

Publications:

• Whole-Function Vectorization.
  Ralf Karrenberg, Sebastian Hack.

• Auto-vectorization of interleaved data for SIMD.
  Dorit Nuzman, Isa Rosen, Ayal Zaks.

• poocl: A Performance-Portable OpenCL Implementation.
  Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, Heikki Berg

• gpucc: An Open-Source GPGPU Compiler.

• Presburger Formula and Polyhedral Compilation.
  Sven Verdoolaege.
  https://lirias.kuleuven.be/bitstream/123456789/523109/3/polycomp-tutorial-v0.02.pdf

• Polly - Performing Polyhedral Optimizations on a Low-level Intermediate Representation.
  Grosser, Tobias and Armin, Groesslinger and Lengauer, Christian.

• Polly-ACC Transparent Compilation to Heterogeneous Hardware.
  Grosser, Tobias and Hoefler, Torsten.
Detailed schedule of the summer school:

<table>
<thead>
<tr>
<th>Time</th>
<th>June 12</th>
<th>June 13</th>
<th>June 14</th>
<th>June 15</th>
<th>June 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>8h30-12h</td>
<td>Introduction to LLVM</td>
<td>Introduction to LLVM</td>
<td>Code Transformation and Analysis Using Clang and LLVM</td>
<td>Code Transformation and Analysis Using Clang and LLVM</td>
<td>Generation of Optimization of Parallel Code in LLVM</td>
</tr>
<tr>
<td></td>
<td>David Chisnall</td>
<td>David Chisnall</td>
<td>Hal Finkel</td>
<td>Hal Finkel</td>
<td>Tobias Grosser</td>
</tr>
<tr>
<td>12h-13h30</td>
<td>Lunch</td>
<td>Lunch</td>
<td>Lunch</td>
<td>Lunch</td>
<td>Lunch</td>
</tr>
<tr>
<td>13h30-17h30</td>
<td>Lab</td>
<td>Lab</td>
<td>Lab</td>
<td>Lab</td>
<td>Lab</td>
</tr>
<tr>
<td></td>
<td>David Chisnall</td>
<td>David Chisnall</td>
<td>Hal Finkel</td>
<td>Michael Kruse/M. Horvath/J. Martinez/S. Guelton</td>
<td>Tobias Grosser</td>
</tr>
<tr>
<td></td>
<td>Gábor Horvath</td>
<td>Gábor Horvath</td>
<td>Gábor Horvath</td>
<td>Juan Martinez/P. Brunet</td>
<td>Gábor Horvath</td>
</tr>
<tr>
<td></td>
<td>André Guinet</td>
<td>André Guinet</td>
<td>André Guinet</td>
<td>Sergio Guelton</td>
<td>André Guinet</td>
</tr>
<tr>
<td></td>
<td>Pierrick Brunet</td>
<td>Pierrick Brunet</td>
<td>Pierrick Brunet</td>
<td>Pierrick Brunet</td>
<td>Pierrick Brunet</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Social event</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>