IMOP: a Self-Stabilizing Source-to-Source Compiler Framework for OpenMP C

Aman Nougrahiya  
Department of CSE, IIT Madras  
amannoug@cse.iitm.ac.in

V. Krishna Nandivada  
Department of CSE, IIT Madras  
nvk@iitm.ac.in

Type: tutorial  
Duration: half day  
Expected number of participants: around 20-25

1 Introduction

Since the public release of its first specification in 1997 [2], OpenMP has stood the test of time in being an industry standard for writing portable shared-memory parallel programs in C/C++ and Fortran. Naturally, there has been a lot of interest in designing new compiler analyses and optimizations for OpenMP. OpenMP API has been implemented as a core part of almost every mainstream compiler, such as GCC [8], LLVM [5], Cetus [3], and ROSE [7]. However, we are not aware of any compiler framework which was designed from the ground up taking OpenMP semantics into account – in all such compilers, OpenMP support was added as an extension to the base infrastructure. As a result, not all components of such frameworks are generally applicable (or conforming) to the OpenMP parallel semantics. Furthermore, the default program representations used in serial frameworks may not be the most intuitive and efficient representations to extend these compilers with new optimizations and analyses, especially in the context of parallel semantics.

In this document, we present a tutorial proposal about a new open-source source-to-source compiler framework called IMOP (IIT Madras OpenMP compiler), which addresses the above-discussed limitations. In IMOP, each component has been designed and implemented by taking OpenMP syntax and semantics into account. IMOP comprises of more than 154kLOC in Java, and works on OpenMP C programs as its input. It is a source-to-source transformation framework, which works at higher levels of program representation, where the OpenMP constructs and directives are easy to reason about and work with. IMOP also provides the compiler writers access to lower-level abstractions that can be used to hide the issues that may occur due to syntactic sugar inherent in the higher levels of program representations.

One of the salient features of IMOP is its self-stabilizing nature. In any compiler, an optimization pass may comprise alternating phases of analyses and transformations. Upon any transformation of the program, various analysis data (such as points-to graphs), and program abstractions (such as concurrency graphs) may become inconsistent with the modified state of the program. The existing compiler frameworks do not perform automated update (or even automated invalidation) of such analysis data and program abstractions, after any program transformations. As a result it is left to the transformation writer to identify (i) which data-structure to update, (ii) when to update the data structures, and (iii) how to perform the actual update. This makes stabilization a very challenging proposition, especially in the presence of a multitude of analyses and transformation passes. In contrast, IMOP is self-stabilizing in nature – it automatically ensures that all analysis data and program abstractions are kept consistent with the modified state of the program, upon any program transformation. Interestingly, such guarantees are provided not only for the existing transformation passes in IMOP, but also for future transformations that may be added to the framework. Similarly, the consistency guarantees are provided not just for the existing analysis data, but also for a major class of future analyses. This saves a lot of programming efforts while adding new optimization passes to the framework, as compiler writers need not worry about explicitly modelling the effects of new/existing transformations on new/existing analyses, in majority of the cases.

IMOP has been successfully used in various published works [4, 9–11] by other research groups; it is also being used in other projects that are under active development.

1.1 Contents of the tutorial

In Figure 1, we provide an overview of our proposed plan for the tutorial. The tutorial will be thoroughly example-driven, along with numerous hands-on exercises, as listed in the plan. We have divided our tutorial into two parts – basic and beyond.

The first (basic) part will start with a discussion on the ideal properties of a compiler framework, and demonstrate the drawbacks of existing frameworks. Then we will introduce IMOP, and briefly discuss its guiding principles and salient features. We will present an overview of various standard program abstractions in IMOP, focussing on how they differ from their counterparts in other frameworks. Next, we will look into various ways of traversing and querying these program abstractions, and demonstrate how to use the existing compiler analyses of IMOP. We will also discuss various transformation passes in IMOP, starting with how we generate new snippets of code in IMOP without having to explicitly create the AST ourselves (unlike in the case of...
Part A. Basic Components of IMOP (3 hrs)

A.1 Introduction.
   (i) Properties of an ideal compiler framework. (ii) Goals and guiding principles of IMOP. (iii) Salient features of IMOP. (iv) Intended audience of IMOP. (v) Block diagram of IMOP.

A.2 Program abstractions in IMOP.

A.3 Analysis passes.
   (i) Visitors for AST traversals. (ii) Visitors for CFG traversals. (iii) Traversals over a CG. (iv) Structural queries on program AST, CFG and CG. (v) Standard iterative data-flow analyses (IDFAs). (vi) Hands-on exercise.

A.4 Transformation passes.

Part B. Beyond Basics (1 hr)

B.1 Program abstractions in IMOP.
   (i) Phase-flow graphs (concurrency graphs). (ii) Inter-task data flow graphs.

B.2 Analysis passes.

B.3 Instrumentation.
   (i) Instrumentation capabilities. (ii) Hands-on exercise.

B.4 Advanced topics.
   (i) The meaning and necessity of self-stabilization. (ii) Utilizing SAT-solving capabilities of IMOP (Z3 integration).

B.5 Miscellaneous.
   (i) Case studies. (ii) Current status. (iii) The road map. (iv) Installation and licensing notes. (v) FAQs.

Figure 1. Course outline of the tutorial

many other frameworks). This will be followed by a brief description of the notion of elementary transformations, as well as higher-level CFG transformations.

In the second part of this tutorial, we will shift our attention towards more involved topics like data-flow analysis, program transformation, and OpenMP-specific components of the framework, such as concurrency graphs, and inter-thread data-flow passes. Among analysis passes, we will first describe the generic data-flow analysis pass, along with guidelines on instantiating the generic pass to create specific iterative data-flow passes, demonstrated with the help of an example instantiation. From transformation passes, we will exemplify the instrumentation capabilities of IMOP, which can be used for a variety of purposes, ranging from profiling to debugging. As part of the discussion on advanced topics, we will explain the self-stabilizing property of IMOP, where it automatically ensures that all stored data from various analyses, and all representations of the program, are made consistent with the modified state of the program after every elementary transformation. We will also briefly describe the integration of Microsoft’s Z3 SAT solver [1] in IMOP, using which one can write powerful compiler analyses (like dependence analysis), without concerning oneself about the low-level details related to the generation of SAT constraints.

2 General details about the tutorial

Objectives of the tutorial

This tutorial is aimed towards compiler researchers, educators, and industrial practitioners alike. By the end of this tutorial, the participants will
- learn how to analyze, optimize, and profile their (OpenMP) C programs, using the existing passes of IMOP,
- be able to write custom optimization passes in IMOP for (OpenMP) C programs,
- understand how to efficiently implement any standard or custom iterative data-flow analysis by extending the generic data-flow passes of IMOP,
- understand the benefits of working in a self-stabilizing framework, and achieve the same in IMOP.

Advertisement plan

We plan to use OpenMP and HPC related mailing lists (in US/Europe/Asia), social networking sites (Twitter, LinkedIn, Facebook), OpenMP Forum, and Latest News section of the homepage of OpenMP’s website for advertising.
Information on past tutorials

In February 2020, we publicly released the IMOP compiler framework during its half-day tutorial at CGO 2020, held at San Diego, CA, USA [6]. This very first tutorial on IMOP (5 participants), contained numerous hands-on exercises, which were in the form of writing passes for querying, analyzing, debugging, and optimizing sample C (OpenMP) programs. The participants were able to finish majority of the exercises encountered during the tutorial. The overall feedback (taken informally during a discussion session at the end) was positive and very encouraging, especially in relation to the ease-of-use of IMOP, as compared to the other standard compiler frameworks. Due to the prevailing Covid-19 pandemic situation, we could not conduct further tutorial sessions after CGO 2020. An online half-day tutorial on IMOP has been selected again for CGO 2021.

In a graduate-level course, Program Analysis (EVEN 2019), offered by Dr. Rupesh Nasre at the Department of CSE, IIT Madras, a short tutorial on IMOP was given to a class of about 30-35 students. IMOP was used by about half of the students as an alternative to the LLVM compiler framework [5], for implementing the final assignment of the course. Towards the end of the course, in a feedback form, the students gave an average rating of more than 4 (out of 5) each, while comparing IMOP to LLVM, in terms of programmability, code readability, and ease of debugging. The overall average rating for IMOP was 4.29/5.

A public talk on IMOP has been given in ACM ISOFT Software Engineering Research in India (SERI 2019), at IISc Bangalore, which was well-received by the audience of about 35-40 researchers and industry practitioners.

Prerequisite knowledge

Following are the compulsory and preferable prerequisites for the audience of this tutorial.

**Must.** A basic undergraduate-level knowledge of compilers and fundamental programming skills in Java/C#/(C++)

**Preferable.** A basic understanding of OpenMP and parallel programming.

Biography of the organizers

**Aman Nougrahiya** is a senior doctoral research scholar at the Department of Computer Science and Engineering, IIT Madras. Under the guidance of V. Krishna Nandivada, he has single-handedly designed and implemented the IMOP compiler framework. Over the period of 4 years during his doctoral research, he has written more than 130 kLOC in Java, for developing IMOP. His other ongoing PhD project concerns with optimization of synchronization operations in parallel programs. He has served as an Artifact Evaluation PC member, for PPoPP 2018 and 2019. He has also served as a student volunteer in ICSE 2014. In the fall of 2014, he was a research intern at Microsoft Research Lab, India, at the Programming Languages and Tools (PLATO) group, where he worked in the area of region-based memory management. His research interests are Multicore Systems, Memory Models, Compilers, Programming Languages, Program Optimizations, and Software Engineering.

**V. Krishna Nandivada** is currently a Professor in the Department of CSE at IIT Madras. He is a senior member of ACM and IEEE. Before joining IIT Madras, he spent nearly 5.5 years at IBM India Research Lab (Programming Technologies and Software Engineering group). Prior to joining IBM Research, he was associated with Hewlett Packard (2000-2001). He holds a BE degree from NIT Rourkela, ME degree from IISc Bangalore, and PhD degree from UCLA. His research interests are Compilers, Program Analysis, Programming Languages, and Multicore systems.

References


