When should one inline recursive functions?

Christian Chavez

Nico Reissmann, Magnus Jahre, and Christian Chavez are with the Norwegian University of Science and Technology (NTNU).

***Index Terms*—Function inlining, Jive, Compiler, 2015, NTNU**

***Abstract*—Lorem ipsum...**

I. Introduction

Describe layout of paper. What does each section in turn discuss?

A. Problemsetting

In these days, optimizing data movemement with regards to the memory hierarchy in computers, is much more relevant than optimizations removing or simplifying operations/computations. The removal of the CPU cycles spent waiting for the necessary data to traverse to the top of the memory hierarchy is a much more rewarding optimization, than optimizing away the instruction counts using the on-chip ALUs in a processor.

This paper explores when inlining should be performed in the Jive compiler, using the new novel approach of the Regionalized Value-State Dependency Graph (**RVSDG**)[1] to allow the compiler perform just such optimizations on inlined code.

The RVSDG (Section III-A) substitutes the known Control Flow Graph (CFG), moving the focus of the program flow graph away from explicitly describing operations and implicitly showing datamovements performed in the program, and instead doing the opposite: explicitly describing the data movement, and implicitly describing the operations performed by the program.

II. Related Work

In this section (...)

To do...

A. Regionalized Value-State Dependency Graph

Insert reference/summary of HiPEAC paper when published

B. Inlining

Cavazos and F.P. O'Boyle [1] explores the heuristic approach on when to inline functions, and in its paper shows how conjunctive normalform can easily be used to decide if and when to inline a specific call site. They report between 17% and 37% execution time improvements without code explosion in their testing of their genetic algorithm auto-tuning.

Serrano [3] explains when to inline functions in the Scheme programming language, by classifying them as either recursive or non- recursive functions, and thereafter mainly inlining non-recursive functions if they do not bloat the code size above a certain treshold.

Serrano [3] also takes overhead into account when deciding on when to inline, and hence also looks at inlining certain recursive functions which fulfill some requirements, at least to a certain depth.

Find a good reference?

C. Recursive Inlining

P. Jones and Marlow [2] explore how the decision of when to inline mutually recursive binding groups in Haskell, through the use of a novel approach using strictness-analysis and forming a graph of the strongly connected components (**SCC**) in said mutually recursive binding group.

They also show how this approach in the Haskell core language does not explode code size when inlining some of the recursive functions. Through the use of their SCC they show how instead of not inlining any recursive function (like Serrano [3]), the SCC graph and some other flags can show when it is not only safe, but desirable to inline a recursive function.

III. Background

A. The Regionalized Value-State Depenency Graph

- γ blocks/regions in the RVSDG are conditionals. The inputs of each γ block is the variables(/data) upon which the conditional depends, as well as the operation performed on these. A CNF may then be represented as several nested γ blocks, all the while retaining the properties of a demand-dependence graph.
- θ blocks/regions are the loop constructs.

  Need better explanation.
- λ blocks/regions are the functions. They retain the demand-dependency graph properties by implementing
- ϕ blocks/regions are the ones describing mutually recursive environments. Inside a ϕ region, the RVSDG will have at least one λ region, and perhaps more. If there is more than one λ present, then the "mutually recursive binding group" situation which P. Jones and Marlow [2] describe when discussing how to inline recursive functions, is also present in the program flow represented by this RVSDG.

Ask Nico for help?

[1] See Appendix I for a the problemsetting in its original PDF.

References

[1] John Cavazos and Michael F. P. O'Boyle. Automatic tuning of inlining heuristics. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 14–, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program.*, 12(5):393–434, July 2002.

[3] Manuel Serrano. Inline expansion: When and how? In *Proceedings of the9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education*, PLILP '97, pages 143–157, London, UK, UK, 1997. Springer-Verlag.

APPENDIX I
REPORT DESCRIPTION

Project Description: An Inliner for the Jive compiler

Nico Reissmann

December 12, 2014

Compilers have become an essential part of every modern computer system since their rise along with the emergence of machine-independent languages at the end of the 1950s. From the start, they not only had to translate between a high-level language and a specific architecture, but had to incorporate optimizations in order to improve code quality and be a par with human-produced assembly code. One such optimization performed by virtually every modern compiler is *inlining*. In principle, inlining is very simple: just replace a call to a function by an instance of its body. However, in practice careless inlining can easily result in extensive *work* and *code duplication*. An inliner must therefore decide carefully when and where to inline a function in order to achieve good performance without unnecessary code bloat.

The overall goal of this project is to implement and evaluate an inliner for the Jive compiler back-end. The project is split in a practical and an optional theoretical part. The practical part includes the following:

- Implementation of an inliner for the Jive compiler back-end. The inliner must be able to handle recursive functions and allow for the configuration of different heuristics to permit rapid exploration of the parameter space.
- An evaluation of the implemented inliner. A particular emphasis is given to different heuristics and their consequences for the resulting code in terms of work and code duplication.

The Jive compiler back-end uses a novel intermediate representation (IR) called the Regionalized Value State Dependence Graph (RVSDG). If time permits, the theoretical part of the project is going to clarify the consequences of using the RVSDG along with an inliner. It tries to answer the following research questions:

- What impact does the RVSDG have on the design of an inliner and the process of inlining?
- Does the RVSDG simplify/complicate the implementation of an inliner and the process of inlining compared to other commonly used IRs?

The outcome of this project is threefold:

1. A working implementation of an inliner in the Jive compiler back-end fulfilling the aforementioned criteria.
2. An evaluation of the implemented inliner.
3. A project report following the structure of a research paper.