Inlining in the Jive Compiler

Christian Chavez

January 27, 2015

Write "(...) Jive Backend Compiler"?

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

Abstract

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Contents

1 Introduction

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

1.1 Problemsetting

Re-state the assignment: Inlining -¿ Easy, what is it, benefits/drawbacks.
"This paper details the problems and benefits of inlining"
Introduce Jive in a few sentences, say that it's further introduced in section 2.

Inlining is a straight-forward technique used in code compilation, which replaces the call of a function with the body of said function. Its benefits include removal of function call overhead (1) and unveiling of additional potential optimizations in the code (2). The drawbacks are potentially increased code size (3), as well as longer execution times for the compilation of the program (4).

The contribution of this paper is an inliner for the Jive backend compiler[1], detailing the problems and benefits of inlining wrt. inlining in the Jive compiler. Jive is a new backend compiler which works on intermediate representation (IR) code, and performs the typically expected compiler techniques and optimizations on said IR code with the help of a new type of graph; the regionalized value-state dependency graph (RVSDG[2]).

Further details of this assignment can be found in Appendix A.

2 Background

2.1 The Jive Compiler

Fill in later.

2.2 The Regionalized Value-State Dependency Graph

The RVSDG is a directed acyclic graph (DAG), which is what separates the Jive compiler from the rest of the other available backend compilers. Jive converts the control flow graph (CFG) representing the control flow of a program (received as IR code) into an RVSDG graph.

The RVSDG, fulfilling the properties of a DAG, has different types of nodes. These nodes are (among others) the γ-, θ-, λ-, apply-, and ϕ- nodes. Each of these nodes represent a different (and typically common) aspect of a program, such as loops, functions, and if-statements.

Bahmann et. al detail algorithms for how any CFG can be converted into an RVSDG, and any RVSDG back again into a CFG. Hence, due to the RVSDG being a demand dependence graph, with the consequence that any program can be represented as a RVSDG.

Cite HiPEAC paper

[1]Detailed in Section 2.1.
[2]Detailed in Section 2.2.

2.2.1 If-Statements

γ nodes in the RVSDG represent conditional statements. Each γ node has two sets of inputs: the predicate, and the variables the predicate depend upon. The outputs are X, representing the dataflow of the RVSDG.

This term is wrong, check up on this.

same as previous todo.

2.2.2 Loops

θ nodes(/regions) represent loops in the program. They are structured as do-while-loops, with an extra γ node in front of the θ node if it's representing a for-loop. The γ node at the end of the node/region has an edge back to the start of the node θ node, containing the representation of the body of the loop. And also an edge onwards out of the nodes onto the next node in the graph.

2.2.3 Functions

λ nodes(/regions) represent functions. Their input edges represent all data worked upon by the function, as well as the function's input parameter variables. The RVSDG forces all λ nodes to represent strict functions.

2.2.4 Function call sites

Apply nodes are "call nodes", nodes which represent places where a function is called. When scanning the program represented by the RVSDG, it is these nodes that are looked upon as potential places upon which to perform inlining.

2.2.5 Something?

Need to remember proper term for this subsubsection.

ϕ regions are nodes representing parts of the program's control flow where either a functions behave recursively either by calling themselves (mutually recursive), or each two or more calling each other in turn.

A ϕ region needs to have at least one apply node and one γ node. A mutually recursive function will have an output edge from the γ node going back to the start of the apply node supplying the input for said γ node[3].

[3]In functional languages such as Haskell, there can be several mutually recursive functions calling each other, which P. Jones and Marlow [8] call "mutually recursive binding groups". This is represented in the RVSDG as several different apply nodes in a ϕ region where the trailing γ nodes have one or more output edges going to one or more apply nodes. It must have at least one edge going to a different apply node for the RVSDG to represent a mutually recursive binding group.

3 Scheme

4 Methodology

5 Results

6 Discussion

7 Related Work

In this section (...)

To do...

7.1 Inlining

W. Davidson and M. Holler [5] examine the proposition that the increased codesize of inlined code affects the execution time performance on demand-paged virtual memory machines. Using equations developed to the describe an inlined programs' execution time, they test this proposition through the use of a source-to-source subprogram inliner.

Cavazos and F.P. O'Boyle [2] use a genetic algorithm in their auto-tuning heuristics to show how conjunctive normalform (CNF) 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 codesize explosion.

Serrano [9] implements an inliner in the Scheme programming language. He details an algorithm for which functions to inline, as well as an algorithm for how to inline recursive functions and non-recursive functions.

Waterman's Ph.D. thesis [10] examines the use of techniques to adaptively decide which functions to inline. The thesis shows the use of CNF for deciding which functions to inline.

E. Hank, W. Hwu, and R. Rau [7] introduces a new technique called *Region-Based Compilation*. And examines the benefits an aggressive compiler can gain from inlining.

P. Jones and Marlow [8] explore an inlining approach for the Glasgow Haskell Compiler (GHC). The paper introduces a novel approach for deciding what mutually recursive functions can be safely inlined without code explosion or non-terminating programs.

Barton, N. Amaral, and Blainey [1] tests whether there should be put more effort into making better inlining decisions, with the intent on helping the compilers improve on loop optimizations.

Deshpande and A. Edwards [6] detail how inlining should be done in the GHC, how it is useful, and the correctness of their presented algorithm.

W. Hwu and P. Chang [3] explored in 1989 how program profile information could be used to decide whether or not to inline C functions statically. Their motivation was to remove costly function calls in a C program, in addition to statically unveil potential optimizations.

Can I say this? Should it be "function profile (...)" instead?

D. Cooper, J. Harvey, and Waterman [4] build work much similar to Waterman's PhD Thesis [10]. Their paper looks into how parameterization of an adaptive inlining scheme can achieve great results compared to todays typical inlining schemes[4]

7.2 Regionalized Value-State Dependency Graph

Insert reference/summary of HiPEAC paper when published

8 Conclusion

8.1 Further Work

9 References

[1] Christopher Barton, José Nelson Amaral, and Bob Blainey. Should potential loop optimizations influence inlining decisions? In *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '03, pages 30–38. IBM Press, 2003.

[2] 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.

[3] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. *SIGPLAN Not.*, 24(7):246–257, June 1989.

[4] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. An adaptive strategy for inline substitution. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 69–84, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] J.W. Davidson and A.M. Holler. Subprogram inlining: a study of its effects on program execution time. *Software Engineering, IEEE Transactions on*, 18(2):89–102, Feb 1992.

[6] Neil Ashish Deshpande and Stephen A Edwards. Statically unrolling recursion to improve opportunities for parallelism. 2012.

[7] Richard E. Hank, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Region-based compilation: An introduction and motivation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, MICRO 28, pages 158–168, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

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

[4] Comparison made mainly with GCC.

[9] 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.

[10] Todd Waterman. *Adaptive Compilation and Inlining*. PhD thesis, Houston, TX, USA, 2006. AAI3216796.

A Project Description

(On next page)

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.