Inlining in the Jive Compiler Backend

Christian Chavez

February 3, 2015

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 **3**
1.1 Report Outline 3

2 Background **3**
2.1 The Regionalized Value-State Dependency Graph 3
2.1.1 Data-dependency edges 3
2.1.2 State-dependency edges 3
2.1.3 N-way statements 4
2.1.4 Tail-controlled loops 4
2.1.5 Functions 4
2.1.6 Mutually recursive functions 4
2.1.7 Fibonacci example 4

3 Scheme **5**

4 Methodology **5**

5 Results **5**

6 Discussion **5**

7 Related Work **5**

8 Conclusion **6**
8.1 Further Work 6

9 References **6**

Appendices **8**

A Project Description **8**

1 Introduction

Since the 1950s, compilers have played an important role in the way programming code is translated into machine languages. In broad terms, compilers perform two things; The translation of human-readable code to machine language, and optimizing the translated programs. One such optimization is inlining.

Need lslisting inlining example here of inlining, code duplication, and work duplication.

Inlining is a straight-forward optimization, which replaces the call of a function with its body. Its benefits include removal of function call overhead and unveiling of additional optimizations. The drawbacks are potential code duplication, work duplication, as well as longer execution times for the compilation of the program. Not all functions are straight-forward to inline, recursive functions being part of this subset, and subsequently an area of focus in this report.

We will also discuss how the inlining heuristics in the Jive backend compiler are used, and how the heuristic can be controlled/modified. Further details of the assignment of this paper can be found in Appendix A.

1.1 Report Outline

Todo: Describe layout/outline of paper. What does each section in turn discuss?

This paper details the inliner for the new compiler backend, Jive, the decisions made for its architecture, and evaluates its performance. Jive takes code in intermediate representation (IR) as input and works on a new IR representation, the regionalized value-state dependency graph (RVSDG[1]).

Todo: Fill in for the empty sections as they come.

2 Background

2.1 The Regionalized Value-State Dependency Graph

The RVSDG is a directed acyclic graph (DAG). The RVSDGhas different types of nodes, and two types of edges. The nodes are the γ-, θ-, λ-, apply-, and ϕ-nodes. The edges will be discussed first.

2.1.1 Data-dependency edges

There are two types of edges in the RVSDG.

The first type is the data-dependency edge. This edge describes a data-dependency one node has from its parent node. In other words, an edge saying which nodes need to finish first before the necessary data is ready for the current node.

2.1.2 State-dependency edges

The other type of edge is the state-dependency edge. This edge is meant to keep the ordering of the nodes consistent with the execution flow of the program, when there are no data-dependencies between them. Stipled lines are used to

Remove this last sentence?

[1] Detailed in Section 2.1.

denote state-dependency edges.

See figure X for an example of why state-dependency edges are necessary for the RVSDG.

Make Fibonacci figure example with RVSDG nodes with a printf in main()

2.1.3 N-way statements

γ-nodes in the RVSDG represent conditional statements. Each γ-node has two sets of inputs: the predicate, and the data dependencies the predicate depend upon. The outputs are any data dependencies used further in the graph (program).

If-else statements are represented as a γ-node, which is split vertically in two. Each subsection has the subgraph and operations to be performed if the predicate is evaluated to true and false, respectively.

2.1.4 Tail-controlled loops

θ-nodes represent loops in the program. They are do-while loops containing the representation of the body of the loop, as well as an edge onwards out of the nodes onto the next node in the graph. θ-nodes

A for-loop would be presented with an if-else γ-node, with a θ-node in the body of the γ-node representing the "true" subgraph

2.1.5 Functions

λ-nodes represent functions, and *apply*-nodes are "call-nodes" which represent where a function is called in the RVSDG.

2.1.6 Mutually recursive functions

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 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.

2.1.7 Fibonacci example

Abovementioned Fibonacci example with examples of graphnodes, maybe code too?

3 Scheme

4 Methodology

5 Results

6 Discussion

7 Related Work

Insert HiPEAC paper

As mentioned in Section 1, inlining has been done since the last half of the 20th century. Following comes the related research used as basis for the work done in this report.

W. Davidson and M. Holler [5] examine the hypothesis that the increased code size 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 hypothesis 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 code size explosion, in java when testing on an Intel PC.

Serrano [9] implements an inliner in the Scheme programming language. The paper details an heuristic for which functions to inline in scheme, as well as an algorithm for how to inline recursive functions and non-recursive functions. Serrano reports an average run time preformance increase of 15% with the inlined Scheme programs.

Waterman's Ph.D. thesis [10] examines the use of adaptive compilation techniques in combination with an inlining heuristic. The thesis shows how CNF can be used for deciding which functions to inline, and examines how there is no single given correct set of parameters for the search space of his hillclimbing algorithm that his heuristic uses. Waterman reports consistently better or equal run time performance when compared to the GCC inliner and ATLAS.

D. Cooper, J. Harvey, and Waterman [4] continue on Waterman's PhD Thesis [10] research. Their paper details how proper use of the parameterization search space of an adaptive inlining scheme using the hillclimber algorithm can achieve great results compared to GCCs inlining scheme. Their results range from 4% to 31% run time performance increase when compared with GCCs inliner.

E. Hank et. al [7] introduce a new technique called *Region-Based Compilation*. They examine the benefits an aggressive compiler can gain from inlining in conjunction with very long instruction words (VLIW) architecture. They report that aggresive inlining can be expensive with an average code expansion of 400%. Yet, they also show that aggressive inlining is able to also unveil further compiler optimizations, leading to averages of 50% of program execution time

spent in functions with more than 1000 operations, instead of more than 80% of execution time spent in functions with less than 250 operations when run without inlining.

P. Jones and Marlow [8] explore an inlining approach for the Glasgow Haskell Compiler (GHC) detail the inner workings of the GHC. Their paper also introduces a novel approach for deciding which mutually recursive functions can be safely inlined without code explosion or non-terminating programs. Jones and Marlow report an average of 30% run time performance increase when using the default settings on the GHC inliner.

Barton et. al [1] tests whether the potential for loop fusion should be taken into consideration in the inlining decisions. They examine this through the use of the IBM®XL Compile Suite and measuring how many additional loops they were able to fuse in the SPECint2000 and SPECfp2000 benchmark suites. Their results indicate that the compiler already catches most of the potential loop fusion optimizations, and the results cannot justify an inter-procedural loop fusion implementation.

Deshpande and A. Edwards [6] detail how inlining should be done in the GHC, with a goal to improve parallelism of recursive functions by "widening" them into the equivalent of multiple recursive calls through unrolling recursion. They leave it as an exercise to the reader to find their performance results.

W. Hwu and P. Chang [3] explore 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. Through the use of the IMPACT-I C compiler, they profile dynamic program information, resulting in a call graph with weighted edges. They report an average elimination of dynamic function calls across their 14 test benchmarks at 16.17%.

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.

[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.