Inlining in the Jive Compiler Backend

Christian Chavez

February 11, 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

List of Figures

Listings

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 actions: the translation from human-readable code to machine language, and optimizing the translated programs. There exist many optimization techniques compilers use. One such optimization is inlining, a straight-forward optimization which replaces the call of a function with its body. See example below in Listing 1.

```c
int foo(int x){
  return x + 3;
}

int bar(int y){
  return foo(y) + 2;
}

// bar() with foo() inlined:
int bar(int y){
  y + 3 + 2;
}
```

Listing 1: Function *foo()* inlined into function *bar()*.

```c
int foo(int x){
  //Computationally expensive function,
  //such as Fibonacci for very high numbers
}

int bar(int y){
  return foo(y) + foo(y+4000);
}
```

Listing 2: Work duplication in *bar()*, when inlining *foo()* into *bar()* due to the twice over calculation of *foo(y)*.

However, the decision of which functions to inline has long been treated as "black magic", due to the non-existence of a perfect inlining heuristic. This report attempts to answer exactly that question for the new compiler backend Jive, introduced shortly.

The benefits of inlining are removal of function call overhead, and a potential for unveiling additional optimizations. The drawbacks are potential code- and work- duplication, shown in Listings 3 and 2, respectively. Additionally, inlining can also negatively affect the compile time, and program executable size[1].

```c
typedef struct{
  int x;
  int y;
} coords_t;

coords_t* foo(int z){
  coords_t* a = (coords_t*) malloc(sizeof(coords_t));
  a->x = 1; a->y = 2;
  a->x *= z; a->y *= z;
  return a;
}

coords_t* bar(){
  coords_t* a = foo();
  coords_t* b = foo();

  b->x += a->x; b.y += a->y;
  return b;
}

```

[1] Unless optimizations to counteract this are unveiled.

```c
// bar() with foo() inlined, with code duplication
coords_t* bar(int z){

  // Lines which are duplicated unnecessarily
  coords_t* a = (coords_t*) malloc(sizeof(coords_t));
  a->x = 1; a->y = 2;
  a->x *= z; a->y *= z;

  // Lines which are duplicating unnecessarily
  coords_t* b = (coords_t*) malloc(sizeof(coords_t));
  b->x = 1; b->y = 2;
  b->x *= z; b->y *= z;
  b->x += a->x; b.y += a->y;

  return b;
}
```

Listing 3: Code duplication in *bar()*, when inlining *foo()* into *bar()*. Total code size (counted by number of statements) for the functions *foo()* and *bar()* are 11 when not inlined, and 13 when inlined.

Not all functions are straight-forward to inline, such as recursive functions. Only an unknown subset of all recursive functions can be inlined, and if one is inlined incorrectly, it can lead to non-termination of the compiler. Recursive functions, the heuristics parameter space, and how the inliner enables rapid exploration of the parameter space, are other topics this report will discuss in turn.

This report describes the inliner for the new compiler backend Jive. It will detail the decisions made for its architecture and heuristics. Jive takes program code in Intermediate Representation (IR) as input and works on a new IR representation, the Regionalized Value-State Dependence Graph (RVSDG[2]).

Todo: In the below text (write it into the text) describe layout/outline of paper. What does each section in turn discuss?

How the RVSDG affects the design of an inliner, and the process of inlining, will also be looked into in this report. Focus will be put on whether the RVSDG simplifies or complicates the implementation of the inliner, and the process of inlining compared to commonly used IRs.

Finally, the implemented inlineris evaluated before we conclude. In the evaluation, focus will be put on how different heuristics have different consequences, in terms of code- and work-duplication.

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

[2] Detailed in Section 2.1.

2 Background

2.1 The Regionalized Value-State Dependence Graph

The RVSDG is a Directed Acyclic Graph (DAG).

A RVSDG has different kinds of nodes, and two types of edges. Nodes can be generalized into two categories, simple and complex nodes. Simple nodes are the nodes representing the "basic operations" a program performs, such as the addition and subtraction of integers. Complex nodes are nodes which contain an RVSDG subgraph. The complex nodes presented below are the γ-, θ-, λ-, apply-, and ϕ-nodes. The edges will be discussed first.

2.1.1 Edges

One type of edge used in an RVSDG is the data dependence edge. This edge represents a data dependency one node has before it can be computed/executed.

The other type of edge is the state dependence edge. This edge is meant to keep the ordering of the nodes consistent with the original flow of execution, when there is no ordering by data dependencies between them. Stippled lines are commonly used to denote state dependence edges.

Make tail-controlled *theta*-node example with some printfs for showing state dependency edges?

See Figure **??** for an example of why state dependency edges are necessary for the RVSDG.

2.1.2 Simple nodes

Simple nodes are defined by their inability to contain an RVSDG subgraph.

Simple nodes are used in an RVSDG to represent simple operations, such as addition, subtraction, and similarly simple operations often referred to as *primitive operations* in programming languages.

2.1.3 Complex nodes

- **N-way statements**

  γ-nodes represent conditional statements. Each γ-node has two sets of inputs: the predicate, and all other edges its subgraph depend upon.

  An if-statement is represented as a γ-node containing the subgraph representing the body of the if-statement. If the predicate evaluates to true, the subgraph will be executed.

  If-else statements are also represented as γ-nodes, but they are divided[3] into two subsections. One subsection of the node contains the subgraph of what will happen if the predicate evaluates to true, and the other will contain the subgraph representing the body of the else- statement.

  The equivalent of how an RVSDG representing nested if-statements looks like, is shown with C/C++ in Listing 4, and its representative RVSDG example in Figure 1.

[3] Typically vertically.

```
if(smth){
  //do something 1
} else if(smth2){
  //do something 2
} else{
  //do something else
}
```

Listing 4: C/C++ code statements corresponding to Figure 1 to the right.

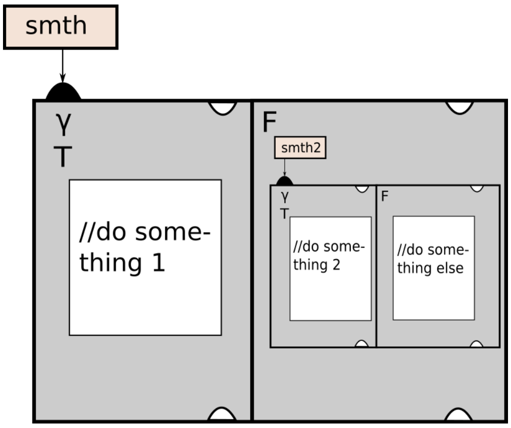

Figure 1: Minimal example of two nested γ-nodes representing the the same control flow as the code in Listing 4 to the left.

- **Tail-controlled loops**

  θ-nodes represent tail loops in the program. They are equivalent to do-while loops containing the representation of the body of the loop. Like with the γ-node, its inputs are all the dependencies needed by its subgraph that represents the body of the loop.

  Other loops, like for example for-loops, can be represented by putting a θ-node inside of the *true* clause of a γ-node with an empty subsection for the *false* clause. If this textual example is to represent a for-loop, both the θ- and the γ-nodes need to each have the same predicate.

- **Functions**

  λ-nodes represent functions, and these are paired with *apply*-nodes. Their *apply*-nodes represent the call sites of the function in the graph (read: program). There should only exist one λ-node per function in the program the RVSDG represents. As each *apply*-node represents a call site, all *apply*-nodes have an edge linking it to its corresponding λ-node.

  See the λ- and its *apply*-node inside of the ϕ-node in F̃igure 2 for an example of a representation of a recursive function in RVSDG.

- **Mutually recursive functions**

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

  To uphold the DAG properties of an RVSDG, there is only one edge going from the inner border of the ϕ-node to the outer border of its contained λ-node. Equivalently, an edge going out from the outer border of the λ-node, to the inner edge of the ϕ-node. This is to denote that once the *apply*-nodes have called the recursive function(s), the function(s) will complete their execution inside of the ϕ-node before exiting the ϕ-node. There is hence no cycle, and thus the DAG properties of an RVSDG are upheld.

Figure 2 illustrates how a ϕ-node containing the representation of a recursive fibonacci function would look like in an RVSDG.

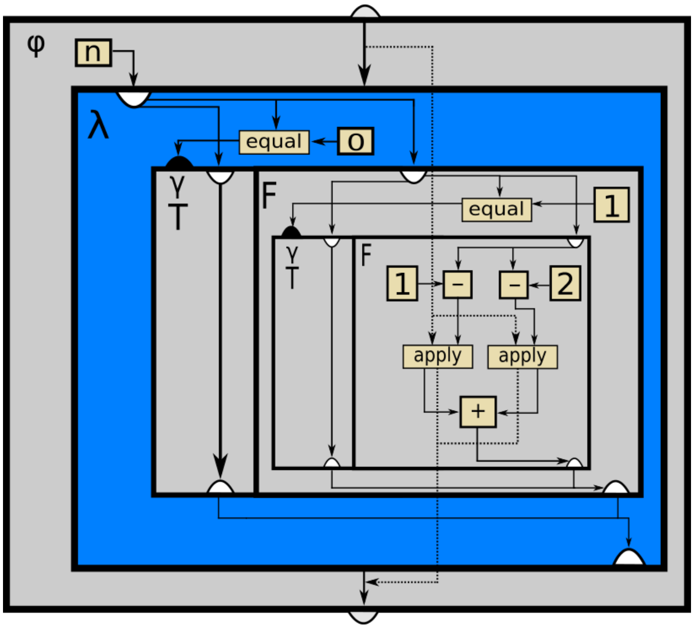

Figure 2: A ϕ-node containing a λ-node representing a recursive version of a function producing the n first numbers in the Fibonacci series.

3 Scheme

4 Methodology

5 Results

6 Discussion

7 Related Work

As mentioned in Section 1, compilers have existed, and most likely optimized code, since the last half of the 20th century. Inlining has long been an important optimization for most compilers.

W. Davidson and M. Holler [5] examine the hypothesis that the increased code size of inlined code affects execution time on demand-paged virtual memory machines. Using equations developed to describe the execution time of an inlined program, 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, for 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, as well as an algorithm for how to inline recursive functions. The paper reports an average run time decrease of 15%.

Insert HiPEAC paper between these two?

Waterman's Ph.D. thesis [10] examines the use of adaptive compilation techniques in combination with an inlining heuristic. His thesis shows how CNF can be used for deciding which functions to inline. It also details how there can be no single given correct set of parameters for all programs, given the search space the heuristics hillclimbing algorithm uses[4]. The thesis reports consistently better or equal run time compared to the GCC inliner and ATLAS.

D. Cooper et. al [4] expand on Waterman's PhD Thesis [10]. Their paper details how the proper use of the parameterization search space[5] in an adaptive inlining scheme can achieve improved results compared to GCCs inliner. Their results range from 4% to 31% run time decrease compared to GCCs inliner.

E. Hank et. al [7] introduce a new technique called *Region-Based Compilation*. They examine the benefits an aggressive compiler gains from inlining on Very Long Instruction Word (VLIW) architectures. The paper reports that aggresive inlining can become costly, with an average code size expansion of 400%. Their results show that aggressive inlining is sufficiently able to unveil further compiler optimizations, leading to an average of 50% of program execution time spent in functions with more than 1000 operations. This is an improvement, compared to their test result of more than 80%, of execution time spent inside functions with less than 250 operations when run without inlining.

P. Jones and Marlow [8] describe the inlining approach for the Glasgow Haskell Compiler (GHC). Their paper introduces a novel approach for deciding which mutually recursive functions can safely be inlined without code size explosion or non-terminating compilation. Jones and Marlow report on average of 30% run time increase.

The report of Barton et. al [1] tests whether the potential for loop fusion should be taken into consideration in the inliner. They disprove this using the IBM®XL Compile Suite, measuring how many additional loops they were able

[4] To decide which functions to inline.

[5] Using an hillclimber algorithm.

to fuse in the SPECint2000 and SPECfp2000 benchmark suites. The results reported 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 an inlining algorithm meant to improve inlining in the GHC. The algorithm improved the parallelism of recursive functions by “widening” them, into the equivalent of multiple recursive calls through unrolling recursion. No results were reported.

W. Hwu and P. Chang [3] explore how program profile information could be used to decide whether or not to statically inline C functions. Their motivation was to remove costly function calls in a C program, in addition to 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 0% to 99% reduction of dynamic function calls in their test benchmarks.

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.