Inlining in the Jive Compiler Backend

Christian Chavez

Wednesday 4th March, 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

Todo's present in document:

1 Introduction

Since the 1950s, compilers have played an important role in the way higher-level programming languages are translated into machine languages. In broad terms, compilers perform two actions: the translation of human-readable code into machine language, and optimizing the translated programs. There exist many optimization techniques compilers use, one being inlining, which replaces the call site of a function/procedure with its body. This is shown in Listing 1, where another optimization technique: constant folding, is shown to be unveiled by inlining.

```
1 int foo(int x){
2   return x + 3;
3 }
4
5 int bar(int y){
6   return foo(y) + 2;
7 }
```

Listing 1: Function foo() inlined into function bar() would result in the body of bar() being `return x + 3 + 2`, in which case constant folding can be used, replacing the *return* expression of bar() with: `x + 5`.

The benefits of inlining are manifold. One is the removal of function call overhead, and the potential for unveiling the application of additional optimizations being another. The drawbacks are potential code-duplication, exemplified in Listing 2, and in specific situations work-duplication[1]. Another potential drawback is increase of compile time, and an increased program executable size.

```
1 int foo(int a){
2   return e; //Big expression, output of which depends on a's value
3 }
4
5 int bar(int x, int y){
6   return f(x) + f(y);
7 }
```

Listing 2: Code duplication in bar(), when inlining foo() into bar(). The big expression e in foo(), would be duplicated when inlined into bar(). The cost of function call overhead would be replaced with an increased size of the final program. However, in this example, the potential for *Common Subexpression Elimination* (CSE) is likely able to negate some of the program size increase.

Not all functions are straight-forward to inline, such as recursive functions. At compile time, the needed depth of recursion might be unknown, leading to non-termination of the compiler as it tries to inline the recursive function until the necessary depth. Hence, extra care needs to be taken when trying to inline recursive functions.

This report describes the construction of an inliner for the Jive compiler backend. It details the decisions made for the inliner's architecture. Jive works on an *intermediate representation* (IR) called *Regionalized Value-State Dependence Graph*[2] (RVSDG). The RVSDG is a *directed acyclic graph* (DAG) and a *demand-based dependence graph* (DDG), making it an IR well suited for effectively enabling worthwhile compiler optimizations.

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

[1] As detailed by P. Jones and Marlow [8] to be the case for the *Glasgow Haskell Compiler*.
[2] Detailed in Section 2.1.

The report details how the inliner is able to handle recursive functions, and how the inliner permits the configuration of different heuristics to allow rapid exploration of the parameter space. How the RVSDG affects the design of an inliner, and the algorithms used by the heuristics deciding what to inline, are also detailed in this report. Focus is put on whether the RVSDG simplifies or complicates the implementation of the inliner, as well as the impact of the RVSDG on an inliner, and the process of inlining, compared to commonly used IRs.

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

A detailed description of the project assignment can be found in Appendix A.

2 Background

2.1 The Regionalized Value-State Dependence Graph

The *Regionalized Value-State Dependence Graph* (RVSDG) is a *directed acyclic graph* (DAG) and a *demand-based dependence graph* (DDG), consisting of nodes representing computations and edges representing the dependencies between nodes. Each node has inputs and outputs connected through edges. The arity and order of inputs and outputs depend on the operation the node represents, and they also needs to have the same arity and order across all nodes representing the same operation.

In this report, we will discuss two kinds of nodes, and two kinds of edges: the simple- and complex-nodes, and the data- and state- dependency edges. Simple nodes represent the "basic operations", such as addition and subtraction. Complex nodes contain another RVSDG subgraph, also called *regions*. The Complex nodes presented in this section are the γ-, θ-, λ-, and ϕ-nodes.

2.1.1 Edges

The RVSDG has two types of edges: the data dependence edge, and the state dependence edge, which represent the data and state values (respectively) of the edges. The data dependence edge represents a data dependency one node has to another, producing the value for the next computation.

State dependence edges preserve the semantics of the original program, keeping the ordering of the nodes consistent with the program's flow of execution. Stippled lines are used to denote state dependence edges in the figures of this report. An example is shown in Figure 2.

2.1.2 Nodes

Of the two previously mentioned categories of nodes, simple nodes are used in an RVSDG to represent simple operations, such as addition and subtraction.

A special case of the simple nodes listed in this report is the *apply*-node, which always has an edge linking a ϕ-region or λ-node as first input. The arity and order of the rest of the *apply*-node's inputs must match the order and arity of the inputs for the λ-node linked directly or through a ϕ-region. While a ϕ-region or λ-node may each link to several *apply*-nodes representing the same function call, an *apply*-node can (and must) always have a link to one ϕ-region or λ-node as its first input.

The complex nodes of an RVSDG relevant for an inliner are as follows:

- **γ-nodes: N-way statements**

  γ-nodes represent conditional statements. Each γ-node has a predicate as input. All other edges passing into the γ-node are edges its subsection's subgraph(s) depend upon. All subsections must have the same order and arity of inputs and outputs, even if the subgraph in each case does not depend on all of the inputs, or modify all of the outputs.

  A γ-node most closely represents a *switch-case* without fall- through in each case. Each case of the switch statement corresponds then with a subsection of the γ-node.

  Hence, a simple *if-statement* with no else-clause can be represented by a γ-node with two subsections. The true subsection containing the RVSDG subgraph representing the body

of the if-statement. The false subsection of the γ-node simply routing through all inputs straight out again unmodified.

How nested γ-nodes can represent the semantics of *if, else if, else* is shown in Figure 1.

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

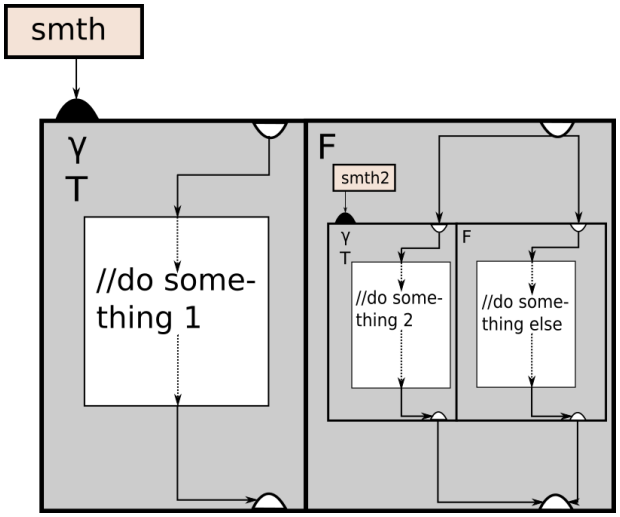

Figure 1: Minimal example of two nested γ-nodes representing the the same semantics as the C/C++ pseudo code on the left.

- **θ-nodes: Tail-controlled loops**

  θ-nodes represent tail controlled loops. Like with the γ-node, its input is the predicate. All other edges are the dependencies needed by its subgraph(s) representing the statements in the body of the loop.

  In C/C++ θ-nodes are equivalent to *do-while loops* containing the subgraph representing of the body of the loop inside the node. Other loops, such as *for-loops*, can be represented by putting a θ-node inside of the *true* clause of a γ-node with no subgraph in the subsection of the *false* clause. The γ- and θ-nodes both need to have the exact same predicate, and same dependencies on the predicate, if the combined subgraph is to represent a for-loop.

  See Figure 2 for an example of a θ-node with corresponding C/C++ code in Listing 3. The stippled directed edges in Figure 2 denote state dependencies between nodes.

```
int recursive_fibonacci(int n){
  if (n == 1 || n == 0){
    return n;
  }
  return recursive_fibonacci(n-1) + recursive_fibonacci(n-2);
}

for(int i = 0; i < 7; i++){
  std::cout << "Fib #" << i << ": " << recursive_fibonacci(i) << std::endl;
}
```

Listing 3: C/C++ code corresponding to the RVSDG subgraph in Figure 2.

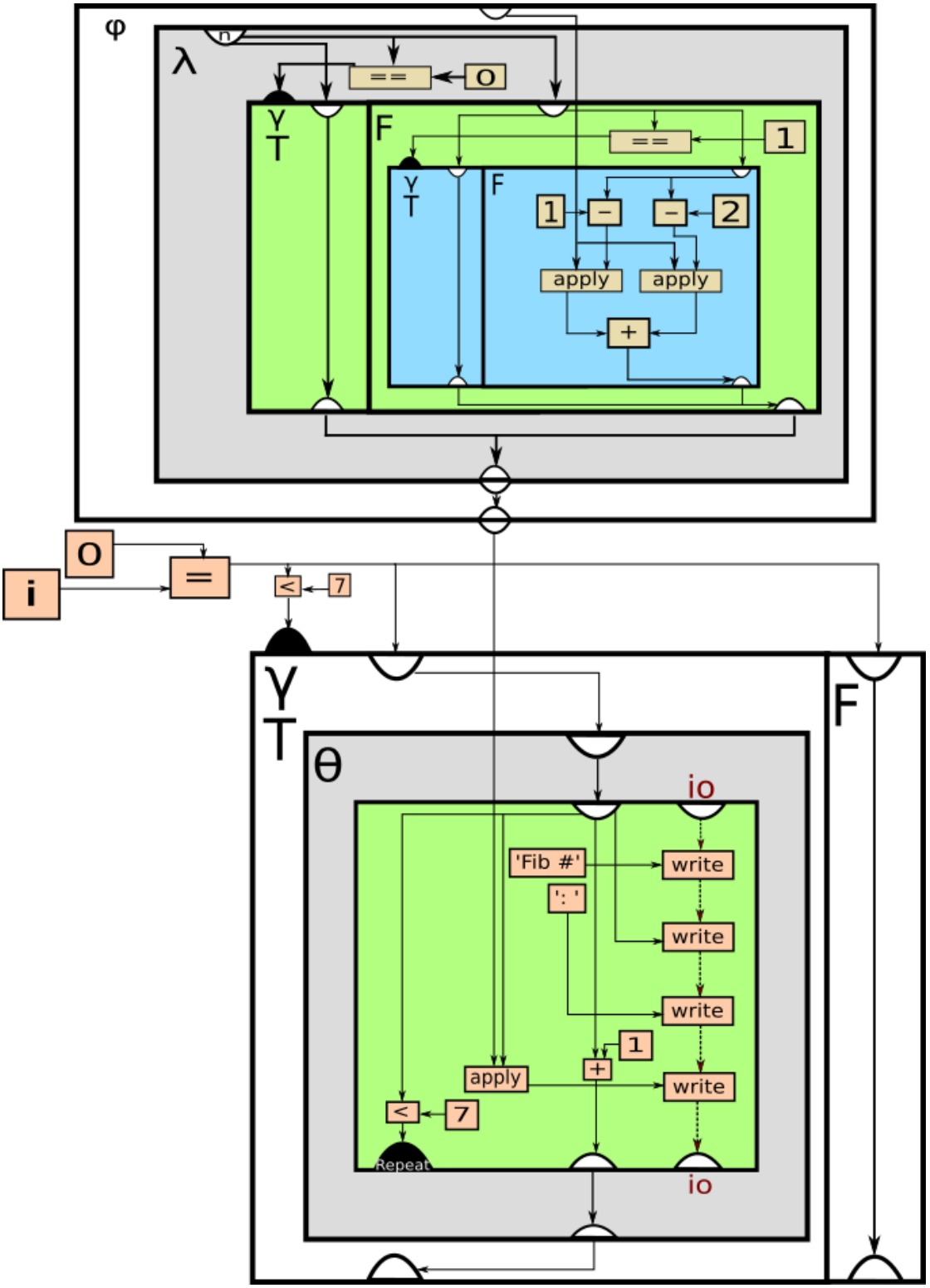

Figure 2: A program consisting of a θ-node looping 7 iterations, calculating and printing the 7 first Fibonacci numbers. The *apply*-node contained in the θ-node links to the same recursive Fibonacci function as in Figure 3.

- **λ-nodes: Functions**

  λ-nodes represent functions, and these are paired with at least one *apply*-node. There is only one λ-node per function in the program the RVSDG represents. As with the other complex nodes, the order and arity of inputs needed for the subgraph of the λ-node need to match the order and arity of the inputs and edges in its corresponding *apply*-nodes. However, there are never any RVSDG nodes connected to the inputs of a λ-node. They serve rather as placeholders, telling the *apply*-nodes the arity and order of the inputs of the λ-node linked to each *apply*-node.

  As previously mentioned, *apply*-nodes represent the call sites of the function represented by the λ-node. All *apply*-nodes have an edge linking it to its corresponding λ-node as its first input. Hence, the only dependence edges going *from* a λ-node are the edges linking it to its *apply*-nodes.

  However, if the λ-node represents a recursive function, it will reside inside of a ϕ-region. It's still linked with all *apply*-nodes representing calls to the function, but instead of being directly linked, like λ-nodes representing non-recursive functions, the *apply*-nodes are linked to an output of the ϕ-region corresponding to the correct λ-node contained in the ϕ-region. Figure 3 illustrates how a recursive λ-node is contained by a ϕ-region in a RVSDG.

- **ϕ-regions: Recursive environments**

  ϕ-regions contain λ-nodes representing recursive functions. ϕ-regions, like λ-nodes, have no inputs for themselves. They may have inputs representing dependencies of their internal RVSDG subgraphs. The functions represented by λ-nodes contained in a ϕ-node behave recursively either through calling themselves, or two or more calling each other (mutually recursive).

  Internally, and externally, they have two sets of "outputs". The external ones can be considered "actual" outputs, enable links to any external *apply*-nodes representing calls to λ-nodes residing inside of the ϕ-region. The internal "outputs" are used to link the internal *apply*-nodes, which give the λ-nodes contained by the ϕ-region their recursive behaviour without breaking the DAG properties of the RVSDG.

  Hence, the λ-nodes inside of a ϕ-region are not directly linked to any *apply*-nodes, like λ-nodes representing non-recursive functions. Instead, the link used by non-recursive λ-nodes goes to the ϕ-region itself, enabling the aforementioned linking between a ϕ-region and internal/external *apply*-nodes. Figure 3 illustrates how a ϕ-node containing the representation of a recursive fibonacci function would look like in an RVSDG, and Figure 2 illustrates how *apply*-nodes can be linked with a ϕ-region.

```
1 int recursive_fibonacci(int n){
2   if (n == 1 || n == 0){
3     return n;
4   }
5   return recursive_fibonacci(n-1) + recursive_fibonacci(n-2);
6 }
```

Listing 4: C/C++ code corresponding to the RVSDG subgraph in Figure 2, which represents a simple recursive fibonacci function.

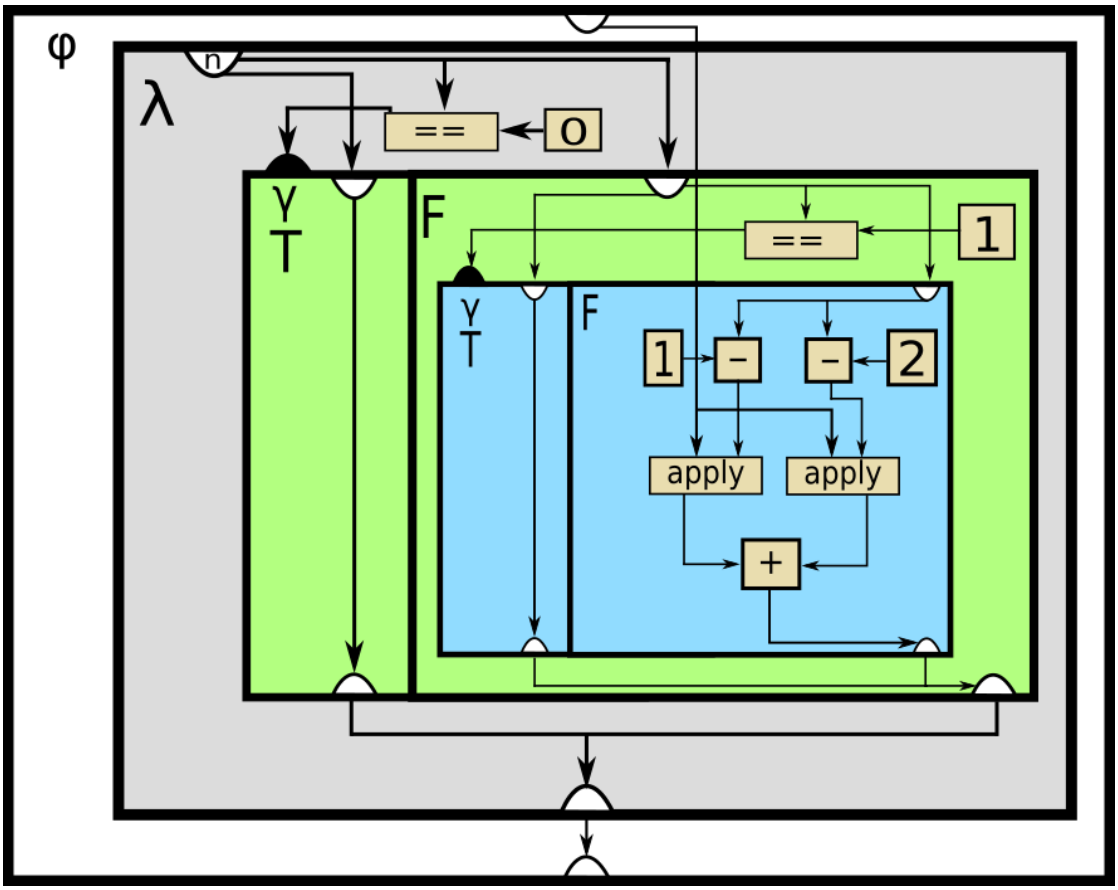

Figure 3: A ϕ-node containing a λ-node representing a recursive version of a function producing the n first numbers in the Fibonacci series.

3 Inlining Heuristics using an RVSDG

Scheme section

The project described in this report implements the heuristic(s) deciding which functions to inline, in the Jive compiler using an RVSDG IR. This section discusses first the utilization of the Heuristics used, then continues with how their implementation in this project.

3.1 Application of the Heuristics

The heuristics used in the project described by this report can be divided into to main subcategories: the heuristics used to decide the order of which *apply*-nodes to look at first, and the heuristics used to decide whether or not to inline a single *apply*-node.

When traversing the RVSDG representing the program being compiled, the order of which *apply*-nodes we find first makes no difference to the approach used in this project. However, the ordering used to sort the *apply*-nodes *found* makes a difference, due to the ordering deciding which *apply*-node might be inlined first.

The heuristic used to order the *apply*-node in this project uses the relative position of each *apply*-node. The further away from the root-node (which represents the final operation of the program), the higher the value representing this relative position.

Draw figure showing why this makes a difference.

Figure **??** illustrates why the order of functions inlined make a difference. Had we not traversed the RVSDG collecting all the *apply*-nodes before inlining them, the the impact of the inlining order would not be as simple to measure.

Using this approach, we can test the consequences of different heuristical orderings of *apply*-nodes in the RVSDG worked on by the compiler.

What else can I say about this?

3.2 Implementation of the Heuristics

The codebase of the project this report describes, is implemented in C/C++. As such, *Standard Template Library* (STL) additions such as vectors, templates, and containers, in addition to the *Object Oriented* (OO) features in C++, are used in the implementation of the inliner.

3.2.1 Traversing the graph for *apply*-nodes

First, we traverse the graph given as a parameter to the function we have analogous to main(), namely apply_inliner_heuristic ().

3.2.2 Ordering the *apply*-nodes of the graph

3.2.3 Whether to inline each *apply*-nodes

4 Methodology

Need an introduction here, no?

4.1 Heuristics

The heuristics used in the project described by this report can be divided into to main subcategories: the heuristics of which *apply*-nodes to look at first, and the heuristics of whether or not to inline a single *apply*-node.

4.1.1 Ordering of the *apply*-node

4.1.2 The decision of inlining each *apply*-node

5 Results

6 Discussion

7 Related Work

As mentioned in Section 1, compilers have existed, and 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.

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 of the heuristics hillclimbing algorithm. 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 using a hillclimber algorithm, 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%. However, their results also show that inlining is sufficiently able to unveil further compiler optimizations. Thus 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 results where more than 80% of the execution time was spent inside functions with less than 250 operations, when run without inlining.

P. Jones and Marlow [8] describe the inliner 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 the risk of non-termination. Jones and Marlow report on average of 30% run time decrease.

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 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

List of Figures

Listings

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

An Inliner for the Jive compiler

Nico Reissmann

Friday 12th December, 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.