# Inlining in the Jive Compiler Backend

Christian Chavez

Thursday $12^{\text{th}}$ 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 played an important role in the way higher-level programming languages are translated into machine languages. The purpose of compilers is two-fold: the translation of human-readable code into machine language, and optimizing the translated code. There exist many optimization techniques compilers use, such as *Common Subexpression Elimination* (CSE) and *Dead Code Elimination* (DCE). Another one is inlining, which replaces the call site of a function with its body, shown in Listing 1.

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

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

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

The benefit of inlining are mainly twofold: The first one is the removal of function call overhead. The second is the potential for unveiling the application of additional optimizations as shown in Listing 1. The drawbacks of inlining are potential code-duplication as exemplified in Listing 2, in specific situations work-duplication[1], and increase of compile time.

```
int foo(int a){
  return e; //Big expression, depending on a
}

int bar(int x, int y){
  return f(x) + f(y);
}
```

Listing 2: Code duplication in bar(), when inlining foo() into bar(). The big expression e in foo(), would be duplicated when inlined into bar(), potentially leading to code-duplication. However, CSE might be able to mitigate code-duplication.

Another factor to consider are recursive functions. Inlining recursive functions uncontrolled leads to non-termination of the compilation. This report will discuss the following techniques used to avoid this in the project:

(1) Only inline the function x times, where x is a maximum recursion depth[10][11]. (2) If there are recursive bindings, make a dependency graph of the recursive calls, and scan the graph for *strongly-connected components* which can then be broken. The key insight being that *the inliner cannot loop if every cycle in the recursive dependency call graph is broken by a call that is never inlined*[7][10].

This report describes the construction of an inliner for the Jive compiler backend. It details the decisions made for the architecture of the inliner. Jive works on an *intermediate representation* (IR) called *Regionalized Value-State Dependence Graph*[2] (RVSDG). The RVSDG is a *demand based* DAG (*directed acyclic graph*) representing the operations performed by any program through nodes, and any dependences between the nodes with edges. Bahmann et. al [1] show how there is no intrinsic structural limitation in the control flow directly extractable from RVSDGs. They also prove in their report termination and correctnes of programs converted from

[1]As detailed by P. Jones and Marlow [10].

[2]Detailed in Section 2.1.

a CFG (*Control Flow Graph*) IR into an RVSDG and back again, without imposing additional overhead on the control flow of the produced object code.

In the below paragraph; describe layout/outline of paper. What does each section in turn discuss?

This 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 is 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) *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.

The RVSDG has two kinds of nodes, and two kinds of edges: simple- and complex- nodes, and data- and state- dependency edges. Simple nodes represent “basic operations”, such as addition and subtraction. Complex nodes contain another RVSDG subgraph, and are also called *regions*.

### 2.1.1 Edges

The RVSDG has two types of edges: data dependence edges, and state dependence edges, representing data and state dependencies respectively. Data dependence edges represent a data dependency one node has to another. State dependence edges are used to preserve the program semantics when it has side-effecting operations. We use dashed lines in this report to denote state dependence edges in figures, as shown in Figure 2.

### 2.1.2 Nodes

The RVSDG has two kinds of nodes: simple nodes are used in an RVSDG to represent simple operations, such as addition and subtraction.

The report puts emphasis on the *apply*- simple node. The *apply*-nodes’ first argument is a link to the node which represents the function the *apply*-noderepresents a call site for. The rest of the input arguments of an *apply*-nodeare the input arguments of the function it’s linked to, with the same arity and order as the node representing the linked function. The results are also of the same order and arity as the outputs of the node representing the linked function the *apply*-noderepresent a call site for.

What is common for all complex nodes, is (1) that they have an extra set of *internal* “outputs” and “inputs”, which are gated from the node’s inputs, and gates to it’s outputs respectively, and (2) they contain an RVSDG subgraph. The complex nodes of an RVSDG relevant for this are as follows:

- **$\gamma$-nodes: N-way statements**

  *$\gamma$-nodes* represent conditional statements. Each $\gamma$-node has a predicate as input. All other edges passing into the $\gamma$-node are edges its subregion’s subgraph(s) depend upon. All subregions must have the same order and arity of inputs and outputs, even if the subgraph in each region does not depend on all of the inputs.

  A $\gamma$-node equivalent to a *switch-case* without fall-through in C/C++. Each case of the switch statement corresponds to a subregion of the $\gamma$-node. Hence, a simple *if-statement* with no else-clause can be represented by a $\gamma$-node with two subregions. The true subregion contains the RVSDG subgraph that represents the body of the if-statement, wheras the false subregion of the $\gamma$-node simply routes all inputs through. As shown in Figure 1.

Fix Figure 1.

```
if((z-2) != 0){
  x = (y*2) / (z+2);
} else{
  //Do nothing
}
```

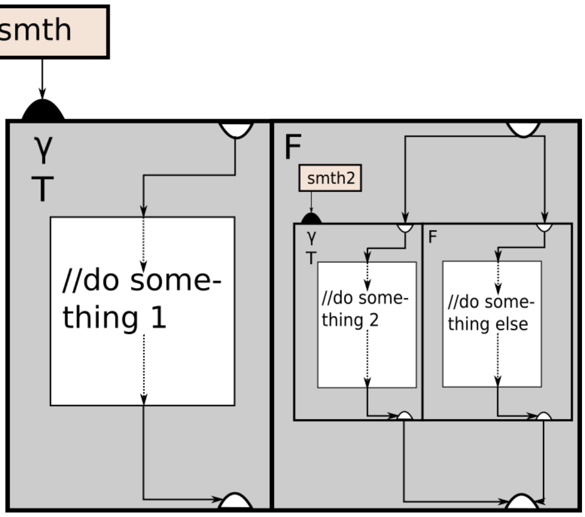


Figure 1: Minimal example of two nested $\gamma$-nodes representing the the same semantics as the C/C++ pseudo code on the left.

- **$\theta$-nodes: Tail-controlled loops**

  *$\theta$-nodes* represent tail controlled loops. As with the $\gamma$-node, its inputs are all the dependencies of its subregion (subgraph). Inside the $\theta$-node there is an extra "internal input", which is the predicate of the tail controlled loop, depending on the loop variant variables. If this predicate evaluates to true, the operations of the subgraph contained in the $\theta$-node are performed again.

  Other loops, such as *for-loops*, can be represented by putting a $\theta$-node inside of the *true* clause of a $\gamma$-node with no subgraph in the subregion of the *false* clause. A $\theta$-node is equivalent to a *do-while* loop in C/C++.

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

```
int fac(unsigned int n){
  unsigned int i = 0;
  unsigned long long result = 1;
  do{
    i += 1;
    result *= i;
    std::cout << "Factorial #" << i << "\tis: " << result << std::endl;
    } while(i < n);
  }
  return result;
}
```

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

Replace Figure 2 with factorial example!

.

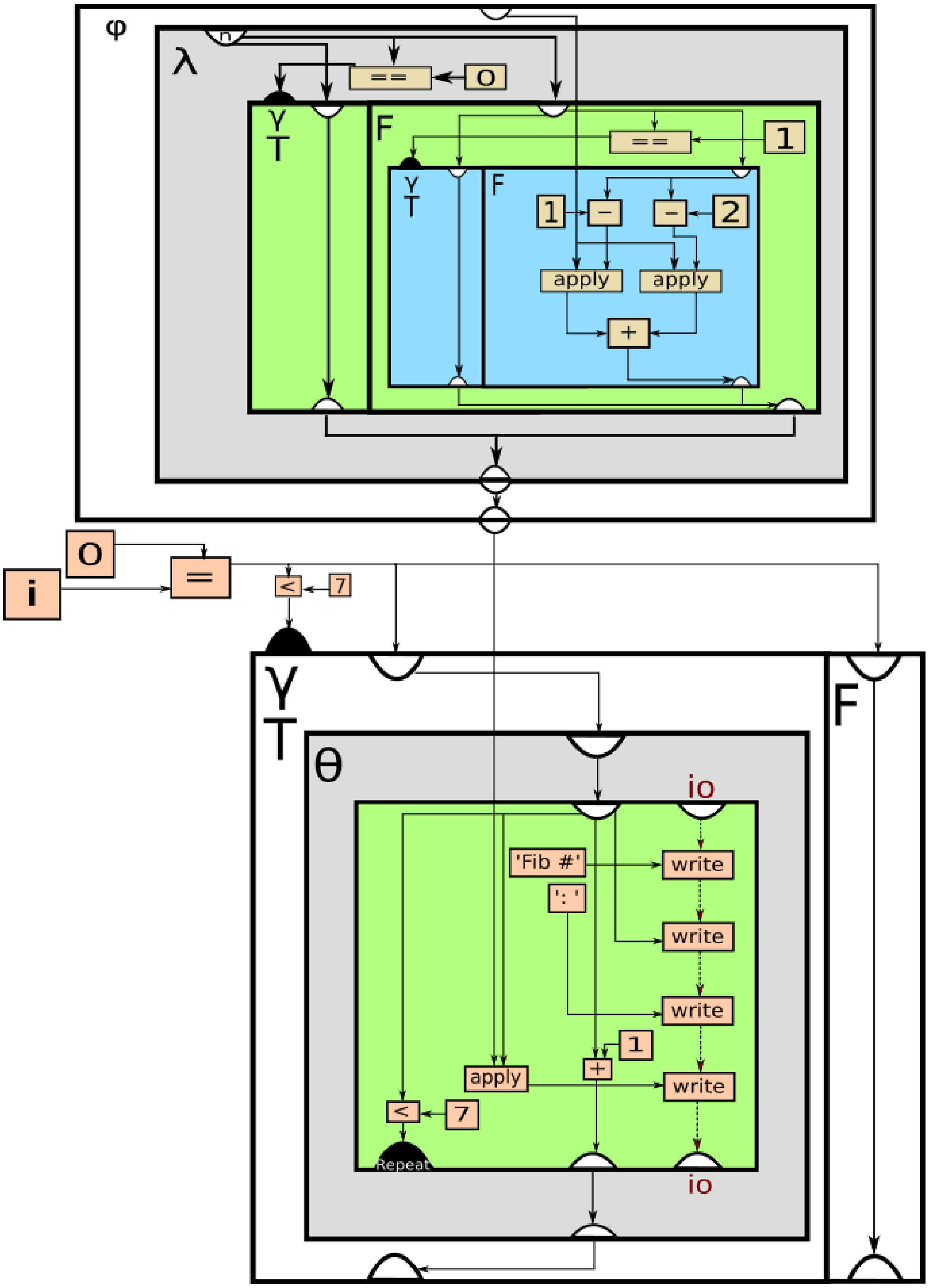


Figure 2: A program consisting of a $\theta$-node looping 7 iterations, calculating and printing the 7 first Fibonacci numbers. The *apply*-node contained in the $\theta$-node links to the same recursive Fibonacci function as in Figure 3.

- **$\lambda$-nodes: Functions**

  *$\lambda$-nodes* represent functions. Their input are all dependencies its subgraph depend upon[3]. A $\lambda$-nodes' outputs need to match the arity of the outputs the function it represents have, unless it is a stateful function, in which case the $\lambda$-node has external outputs for each side-effecting dependency.

  However, $\lambda$-nodes themselves are never linked with anything but the *apply*-nodes representing the call sites of the function the $\lambda$-node represent. It is the *apply*-nodes which receive the input dependence edges and give out the output dependence edges in an RVSDG. As previously mentioned, the arity and order of inputs and outputs for the subgraph(s) inside the linked $\lambda$-node must match the order and arity of the inputs and outputs of the linked *apply*-node.

- **$\phi$-nodes: Recursive environments**

  *$\phi$-nodes*' subgraphs must contain at least one recursive $\lambda$-node. As such, $\phi$-nodes have no external inputs, but they have external outputs which represent links to each $\lambda$-node contained within. The internal "outputs" of a $\phi$-region are links representing the $\lambda$-nodes contained within, thus upholding the DAG properties of an RVSDG containing $\lambda$-nodes representing recursive functions.

  All *apply*-nodes causing recursion inside a $\phi$-node get their first input link from the "internal input" of the $\phi$-node which corresponds to the $\lambda$-node the *apply*-noderepresents a call site for. Hence, the $\lambda$-nodes within do not extend any edges for their "outer outputs" from their RVSDG subgraphs, such as non-recursive $\lambda$-nodes do. Instead, they have one single output, which is linked to the internal "inputs" of the $\phi$-node, which in turn are gated through to each its own external "output" in the $\phi$-node.

  In this way, *apply*-nodes representing calls to the recursive functions contained in the $\phi$-node, which are located elsewhere in the RVSDG, can get their link to the $\lambda$-node representing function the *apply*-nodes represent a call site of. A recursive fibonacci function represented as an RVSDG illustrates the usage of a $\phi$-node in Figure 3).

```c
int rec_fib(unsigned int n){
  if (n < 2){
    return n;
  }
  return rec_fib(n-1) + rec_fib(n-2);
}
```

Listing 4: C/C++ code corresponding to the RVSDG subgraph in Figure 3, which represents a simple recursive fibonacci function.

[3] Of which any parameters the function the $\lambda$-node represents needs is a subset.

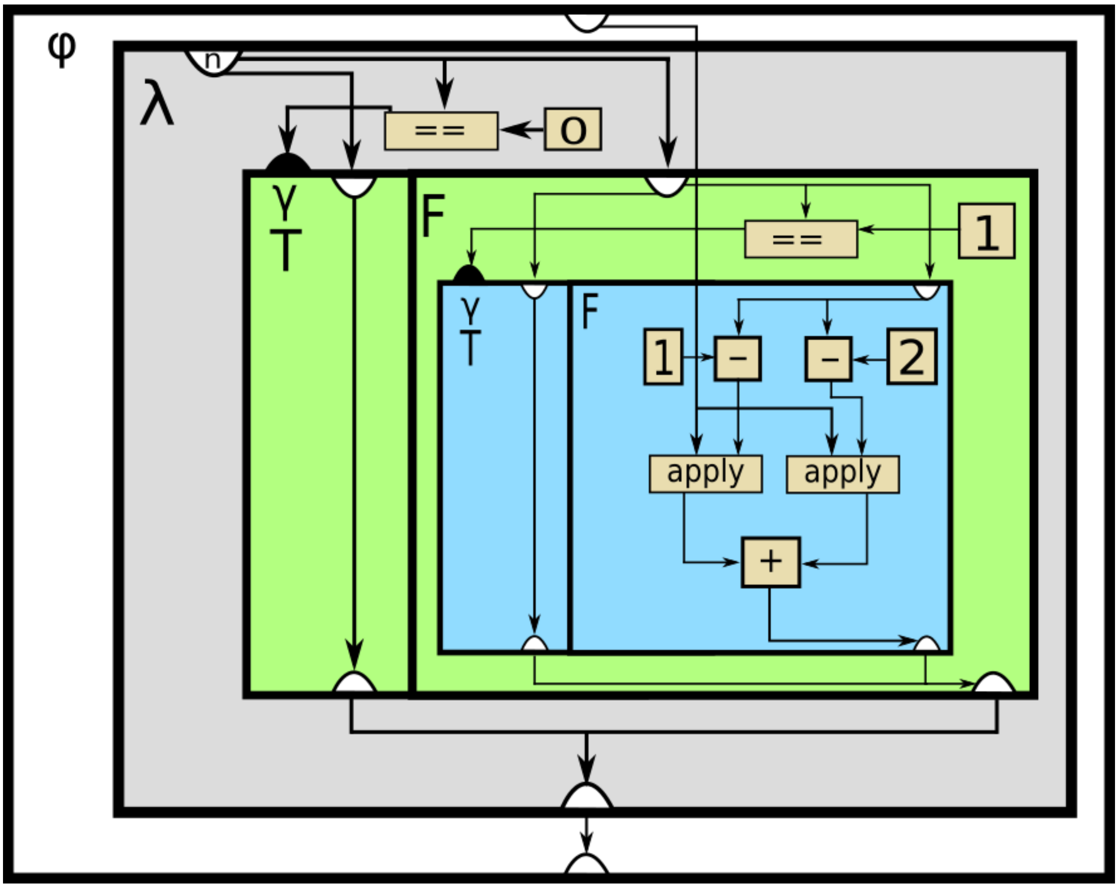


Figure 3: A $\phi$-node containing a $\lambda$-node representing a recursive version of a function producing the $_n$ first numbers in the Fibonacci series.

# 3 The Inliner

A graphical flow chart of the architecture?

In broad strokes, the inliner does two things. It traverses the RVSDG, collecting all the *apply*-nodes (1), before it attempts to inline each one in a certain order (2). This Section will describe how these two actions are performed when working with the RVSDG IR, while the actual heuristics are detailed in Section 4.1 and Section 4.4.

## 3.1 Collecting all function call sites

The traversal of the RVSDG to collect all the apply nodes is simple. The inliner receives the RVSDG graph, iterates through every node of the RVSDG, and makes reference copies of each *apply*-node. Seeing as the RVSDG is a DAG, no care needs to be taken when it comes to recursive functions in the inital collection, as nodes representing recursive environments do not have duplicate *apply*-nodepresent in the graph[4].

## 3.2 The order of inlining call sites

Need a figure showing why the order of inlining matters.
+ reference to further ideas when that's done/started on.

## 3.3 Inlining a call site

When the collected *apply*-nodes from Section 3.1 are ordered in the order we want to visit them in, we run the heuristic described in Section 4.4, or the heuristic described in Section 4.3 dependent upon whether the function the *apply*-noderepresents is recursive or not.

If the *apply*-nodepasses the check, the inlining is performed by making a copy of the subgraph contained in the $\lambda$-node the *apply*-nodeis linked to. The copied subgraph is then put in the RVSDG where the *apply*-nodewas before, with all edges previously connected to the *apply*-nodenow connected to the copied subgraph instead.

However, before moving onto the next *apply*-nodein the list of previously collected *apply*-nodes, the inliner goes through the copied subgraph and attempts to inline all *apply*-nodes contained within first. Hence, the ordering described in Section 3.2 is not X.

find a good word for this, want to say broken.

After the copied subgraph's potential *apply*-nodes are dealt with, the inliner moves on to the next one in the list collected in Section 3.1. When all *apply*-nodes in this list have passed through the heuristic, and at least one *apply*-nodehas been inlined, the RVSDG representing the program is optimized (pruned), and the inliner jumps back to the traversal and starts anew.

[4] As described in Section 2.1.

# 4 Methodology

Need an introduction here, no?

## 4.1 Ordering of the call sites heuristic

## 4.2 Inlining conditions used to decide whether or not to inline

Inliner conditions:

- Statement Count
- Loop nesting depth
- Static call count
- Parameter count
- Constant parameter count
- Calls in procedure
- Dynamic call count

Do we keep this one? How can we even get this one? Waterman used profiling of the program to get this...

## 4.3 Heuristic inlining call sites representing recursive functions

## 4.4 Heuristic inlining call sites representing non-recursive functions

# 5 Results

## 6 Further ideas

# 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 [6] 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 [3] 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 [11] 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%.

Waterman's Ph.D. thesis [12] 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 [5] expand on Waterman's PhD Thesis [12]. 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 [9] 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 no inlining was employed.

P. Jones and Marlow [10] 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 [2] 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 [8] 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 [4] 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] Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer. Perfect reconstructability of control flow from demand dependence graphs. *ACM Trans. Archit. Code Optim.*, 11(4):66:1–66:25, January 2015.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] Bas den Heijer. Optimal loop breaker choice for inlining. Master's thesis, Utrecht University, Netherlands, 2011.

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

[9] 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.

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

[11] 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.

[12] 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 $12^{th}$ 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.