Inlining in the Jive Compiler Backend

Christian Chavez

Wednesday 25th 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 been translating most, if not all, higher-level programming languages into machine languages. The purpose of a compiler is two-fold: translate human-readable code into machine language (1), and optimize the translated code (2). There exist many code optimization techniques, such as *Common Subexpression Elimination* (CSE) and *Dead Code Elimination* (DCE).

Another code optimization is *inlining*, which replaces the call site of a function with its body. Listing 1 shows how when function `foo()` is inlined into `bar()`, the body of `bar()` will become `return y + 3 + 2;`. After the inlining is performed, the potential for optimization using *Constant Folding* is unveiled, and can be applied to the body of `bar()`.

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

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

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

The benefits of inlining are mainly two-fold: The first one is the removal of function call overhead. Function call overhead is the overhead cost in memory space needed on the stack. Before CPU cache memory and registers became as spacious as they are now, this could be a very worthwhile optimization in itself. The second is the potential for unveiling the application of additional optimizations as shown in Listing 1.

The drawbacks of inlining are code-duplication, and an increased compile time. In specific situations, work-duplication can also occur[1]. Listing 2 exemplifies how inlining can lead to code-duplication, when `foo()` is inlined into `bar()`. This happens because the big expression `e` in `foo()` is twice copied into `bar()`.

However, code-duplication might be negatable if the potential optimization of CSE is unveiled.

```
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()`.

If inlining is performed without caution, another drawback occurs when the compiler attempts to inline recursive functions. Unless recursive functions are handled specially, non-termination of the compilation will occur. The literature proposes two main approaches for handling recursive functions:

1. Avoid non-termination of the compilation by only inlining recursive functions to a certain depth [10][11].

[1] As detailed by P. Jones and Marlow in Section 2.2.2 [10].

2. If the recursive environment has more than one recursive binding[2], scan the *dependency graph* of the recursive binding group for *Strongly Connected Components* (SCCs). If an SCC is found, choose one of its recursive bindings which will not be inlined. The rest of the recursive bindings in the SCC can then safely be inlined [7][10].

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:

This report describes the construction of an inliner for the Jive compiler backend, detailing its design and. Jive uses an *intermediate representation* (IR) called *Regionalized Value-State Dependence Graph*[3] (RVSDG). The RVSDG [1] is a *demand-based-* and a *Directed Acyclic -Graph* (DAG) representing the operations performed by any program through nodes, and any dependences between the operations through edges.

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

This report explains 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]Recursive bindings defined as Section 3.2 of P. Jones and Marlow [10] defines them.
[3]Detailed in Section 2.1.

2 Background

2.1 The Regionalized Value-State Dependence Graph

The *Regionalized Value-State Dependence Graph* [1] (RVSDG) is a *Directed Acyclic Demand-Based Dependence Graph* (DADBDG), 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, as exemplified with the two subsections of node representing the if-statement in Figure 1.

```
if( (z-2) != 0 ){
  x = (y*2) / (z+2);
}
```

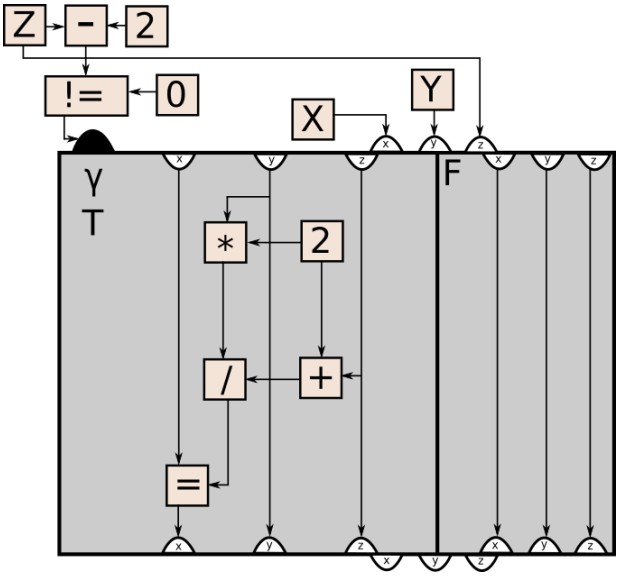

Figure 1: Example of an RVSDG subgraph equivalent to a simple C/C++ if-statement.

2.1.1 Edges

The RVSDG has two types of edges: data dependence edges, and state dependence edges, representing data and state dependencies one operation (node) has to another, respectively. Data dependence edges are used to denote data dependencies of an operation. An example of this being a variable used in an addition. State dependence edges are used to preserve the semantics of the program when the program has side-effecting operations, meaning that there are no data dependencies between operations (nodes) that need to be performed in a set order.

Figure 1 illustrates how data dependence edges work and look like, and Figure 2 shows how state dependence edges are used and look like, in RVSDGs. We use dashed lines in this report to denote state dependence edges in figures such as shown in Figure 2.

2.1.2 Nodes

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

order of inputs and outputs of any RVSDG node need to match across all the nodes representing the same operation.

However, simple nodes representing associative operations, such as multiplication or equality-check, can switch the order of their inputs and still yield the correct output. The same is not valid for non-associative operations, such as division, substraction, and assignment operations.

The report puts special focus on the simple node called the *apply*-node. An *apply*-node represents the call site within an RVSDG of a function of the program the RVSDG represents. The first argument of an *apply*-node is a link to the complex node which represents the function the *apply*-node represents a call site to. The rest of the input arguments of an *apply*-node need to match the order and arity of the input arguments of the function-node it's linked to. Likewise, the results also need to match the same order and arity as the outputs of the function-node.

Complex nodes contain an RVSDG subgraph, which is why they are also referred to as *regions*. As simple nodes, complex nodes also need to have the same order and arity of their inputs and outputs across all nodes representing an equivalent operation.

Differing from the simple nodes with their contained subgraph, complex nodes "gate" the inputs they get from the rest of the RVSDG through to internal outputs. Consequently, they also have internal inputs which gate through to their external outputs, connecting the dependencies to the rest of the RVSDG. The complex nodes of an RVSDG relevant for this report are as follows:

- **γ-nodes: N-way statements**

  γ-nodes represent conditional statements. Each γ-node has a predicate as first input. All other edges passing as inputs to the γ-node are edges its subregions depend upon. All subregions must have the same order and arity of internal inputs and outputs, even if the subgraph in each region does not depend on all of the internal outputs.

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

- **θ-nodes: Tail-controlled loops**

  θ-nodes represent tail controlled loops. As with the γ-node, its inputs (and outputs) are all the dependencies needed the RVSDG subgraph in its subregion.

  Inside the θ-node there is an extra first internal input, which is the predicate of the tail controlled loop. If this predicate evaluates to true, the rest of the internal inputs of the θ-node are mapped to their corresponding internal outputs.

  This enables the iterative behaviour of an RVSDG θ-node. The new values from the internal inputs of the previous iteration are mapped to the internal outputs of the θ-node. Then the operations represented by its contained RVSDG subgraph are executed again with the new values.

  A θ-node is equivalent to a *do-while* loop in C/C++, as shown in Figure 2. See Listing 3 for the C/C++ code corresponding to the RVSDG depicted in Figure 2.

```
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!

Other loops than tail-controlled loops can be represented by combining complex nodes. A *for-loop* can be represented by putting a θ-node inside of the *true* clause of a γ-node conaining no subgraph in the subregion representing the *false* clause.

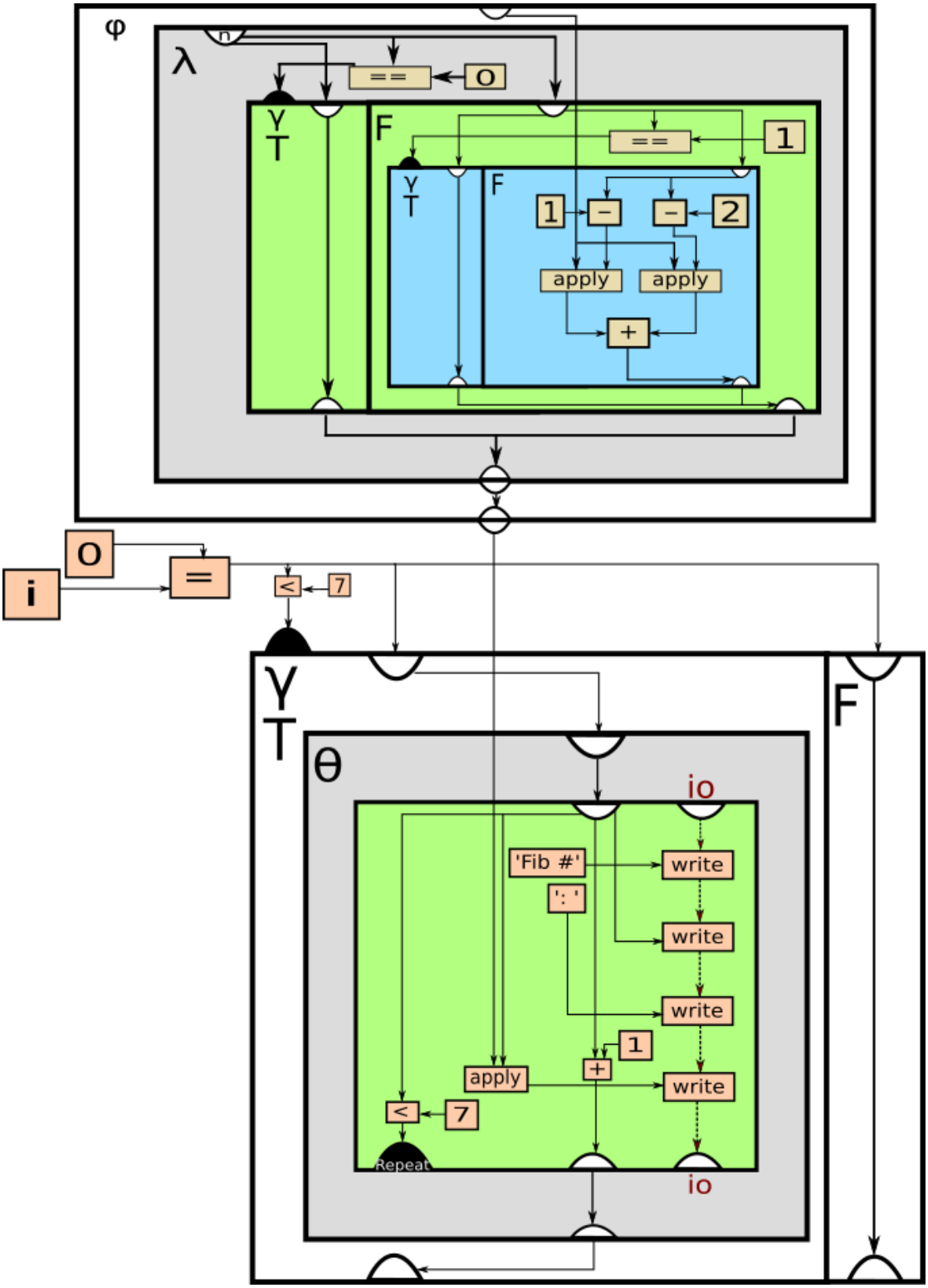

Figure 2: An RVSDG representing the C/C++ program code in Listing 3.

- **λ-nodes: Functions**

  A λ-*node* contains an RVSDG subgraph representing operations performed by a specific function. λ-*nodes* only have internal outputs and outputs, representing the arguments and results needed and given to its contained RVSDG subgraph, respectively.

  The invocation of a function represented by a λ-*node* is accomplished through linking the λ-*node* to the *apply*-node at the call site of the function in the RVSDG.

  The external inputs of the *apply*-node are mapped to the internal outputs of the λ-*node*. This enables the evaluation of the body of the λ-*node*. The arity and order of its internal inputs and outputs must match the arity and order of the external inputs and outputs of all correspondingly linked *apply*-nodes. Thus the results of the λ-*node* are mapped to the results of the *apply*-node. Figure 3 shows an RVSDG representation of a recursive fibonacci function.

- **ϕ-regions: Recursive environments**

  ϕ-*regions* contain at least one recursive λ-*node*. Like the λ-*node*, it only has internal inputs and outputs. The internal outputs of the ϕ-*region* connect to the links used by the *apply*-nodes within it to link to their respective λ-*nodes*, if these are also present within the same ϕ-*region*. The internal inputs of a ϕ-*region* link to the λ-*nodes* contained within. Thus enabling *apply*-nodes outside of the recursive environment to link with the λ-*nodes* contained within.

  An RVSDG representing a recursive fibonacci function illustrates the usage of a ϕ-*region* in Figure 3.

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

Listing 4: C/C++ code corresponding to the RVSDG subgraph in Figure 3.

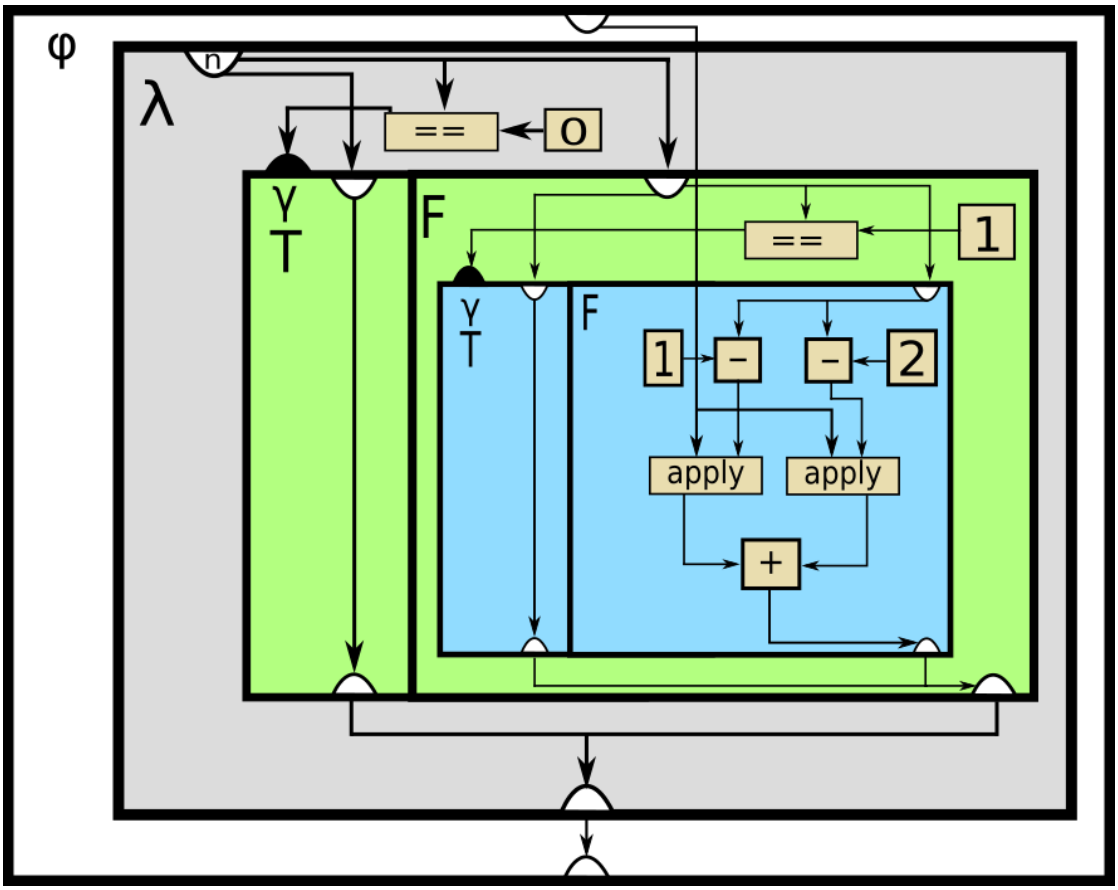

Figure 3: A ϕ-node containing a λ-node representing a recursive version of a function producing the n first numbers in the Fibonacci series.

3 The Inliner

A graphical flow chart of the architecture?

The inliner this report describes performs the following when given an RVSDG as input:

1. Scan through the RVSDG, finding all the *apply*-nodes.
2. Order the *apply*-nodes according to the algorithm described in Section 3.1.
3. Use the ordering to evaluate each *apply*-node in turn and decide whether or not to inline all the non-recursive *apply*-nodes according to the algorithm described in Section 3.2*.
4. When all non-recursive *apply*-nodes have been evaluated, evaluate at the recursive ones according to the same ordering as the non-recursive ones. Then decide whether or not to inline each when evaluating them according to the algorithm described in Section 3.3*.

*Whenever an *apply*-node has been inlined, the RVSDG is "pruned" by Jive's optimizer. Hence, care needs to be taken to ensure that a previously collected *apply*-node still is present when we evaluate it. Previously inlined *apply*-nodes may have caused Jive's pruning to remove this *apply*-node.

3.1 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.

The ordering of the *apply*-nodes we look at when deciding whether or not to inline them, matters because inlining opportunities might be missed with one ordering, and unveiled with another.

Our inliner orders them according to a property each node has in Jive, distance_from_root. This property gives us an indication of where in the *call graph* the call site is located, allowing us to iterate through the call sites with a *Depth First Search* (DFS) top-down through the call graph.

Ensure the correctness of this statement. Also, need to add a statement saying something about bottom up.

3.2 Inlining a call site

Describe the algorithm and inliner conditions we land on after testing.

When a call site is inlined, the inliner scans through the inlined RVSDG subgraph copied from the function. Then it evaluates all *apply*-nodes found in the subgraph according to the same ordering as used on the one inlined. Finally, it returns to the next one in the ordered list of *apply*-nodes from which it got the original one which was inlined.

The ordering is ensured not to be broken if the original ordering is a top-down or bottom-up DFS of the RVSDG's call graph.

3.3 Inlining a call site for a recursive function

Describe GHC paper's way to inline.

After all non-recursive *apply*-nodes are inlined, the inliner has a list of all the recursive ones sorted in the same order as the non-recursive ones were evaluated.

The inliner then looks at each recursive *apply*-node, and scans the recursive binding group for *Strongly Connected Components* (SCCs). If an SCC is found, one *apply*-node is chosen as breakpoint, and all the rest of the recursive bindings in the SCC are inlined instead.

Write how when implemented.

If no SCC is found, the recursive *apply*-nodeis

Decide on this. Is it unrolled? Or ignored? If unrolled, how many times?

4 Methodology

Need an introduction here, no?

4.1 Inlining conditions used when inlining

Inliner conditions:

- Statement count: This function property is the number of C/C++ statements contained within a function.A function's statement count is an inliner condition we want to utilize because it is gives us an idea of the size of the code- duplication if we inline the function.
- Loop nesting depth: This property tells us how potentially useful it is to inline this specific call site. The assumption is that most of a program's execution time is spent within loops, so there is potentially more to gain if optimizations are unveiled by inlining call sites inside nested loops.
- Static call count: This property tells us how many call sites there are for this function in the program. If this count is low, it may be worth inlining all the call sites and eliminating the original function.
- Parameter count: The greater the amount of parameters a function has, the greater the invocation cost of said function. This is especially true when type conversion is required. In some cases, the computational cost of an inlined with low statement count may be smaller than the cost of invoking it if it has many parameters[12].
- Constant parameter count: This property tells us how many of the call site's parameters are constant at the call site. Function invocations with constant parameters can often benefit more from unveiled optimizations after inlining.
- Calls in procedure: This function property tells us how many call sites are located inside the function the call site invocates. Hence, it enables finding leaf functions. Waterman [12] introduced this parameter for two distinct reasons: leaf functions are often small and easily inlined, and a high percentage of total execution time is spent in leaf 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, 2012.

[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 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.