Inlining in the Jive Compiler Backend

Christian Chavez

Friday 10th April, 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 higher-level programming languages into machine languages. The purpose of a compiler is two-fold: translate human-readable code into machine language, and optimize the translated code. There exist many code optimization techniques compilers use, 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 a situation where if function foo() is inlined into bar(), the body of bar() becomes return y + 3 + 2;. After the inlining is performed, *Constant Folding* is unveiled, which can replace 3+2 with 5.

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

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

Listing 1: *Constant Folding* unveiled when inlining foo() into 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 needed on the stack. 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, if0 foo() is inlined into bar(). This happens because the big expression e in foo() is copied twice into bar().

However, code-duplication might be mitigated if CSE is applicable.

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

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

Listing 2: Code duplication in bar(), when inlining foo() into bar().

If inlining is performed blindly on all function call sites, non-termination of the compilation can occur. This can happen when the compiler attempts to inline recursive functions. Unless recursive functions are handled specially, non-termination of the compilation may 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 [11][12].
2. If the recursive environment has more than one recursive binding[2], decide on bindings (functions) in the *recursive binding environment* to be loop breakers, and do not inline

[1]As detailed by P. Jones and Marlow in Section 2.2.2[11].

[2]Recursive bindings defined as in Section 3.2 of P. Jones and Marlow[11].

these. The remaining recursive functions in the recursive binding environment can then safely be inlined wrt. termination of the compilation[8][11].

This report describes the construction of an inliner for the Jive compiler backend, detailing the design and architecture. Jive uses an *intermediate representation* (IR) called *Regionalized Value State Dependence Graph*[3] (RVSDG). The RVSDG[1] is a *demand-based* and *directed acyclic graph* (DAG) where nodes represent computations, and edges represent the dependencies between these computations.

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, we focus on how different heuristics have different consequences, such as code-duplication, and others. A detailed description of the project assignment can be found in Appendix A.

[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*, 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.

Figure 1 exemplifies how the C/C++ code on the left side can be represented as a *directed acyclic demand-based dependence graph*, depicted in the figure on the right side.

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

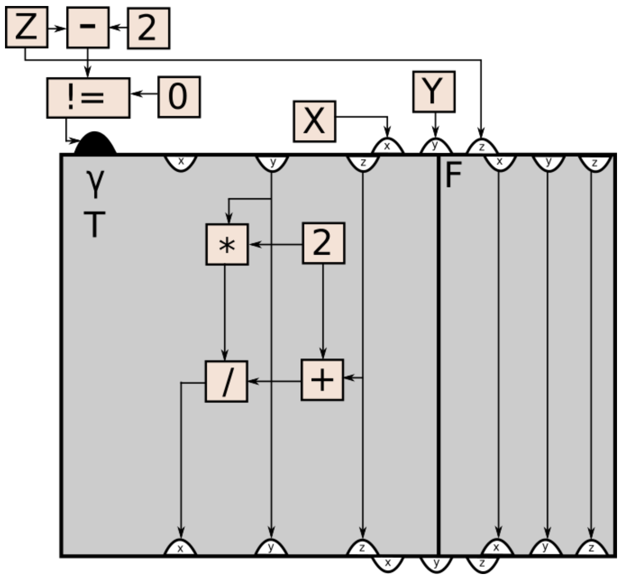

Figure 1: Example of an RVSDG subgraph equivalent to the C/C++ if-statement on the left.

2.1.1 Edges

The RVSDG has two types of edges: data dependence edges and state dependence edges. They represent data and state dependencies operations have to another, respectively. An example of data dependence edges are the operands used in an addition.

State dependence edges are used to preserve the semantics of the program when the program has side-effecting operations. Consequently, there are no data dependencies between operations, giving an order to the execution of the connected operations.

2.1.2 Nodes

The RVSDG has two kinds of nodes: simple nodes and complex nodes. Simple nodes are used to represent primitive operations, such as addition and substraction. The arity and order of inputs and outputs of any RVSDG node need to match the operation.

This report puts special focus on the *apply*-node. An *apply*-node represents the call site of a function. The first input argument of an *apply*-node is a link to the function the *apply*-

node invokes. The rest of the inputs 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*. Differing from the simple nodes with their contained subgraph, complex nodes normally "gate" the external inputs they get from the rest of the RVSDG through to internal outputs. Consequently, they also have internal inputs which can gate through to their external outputs, connecting the dependencies to the rest of the RVSDG.

Figure 2 illustrates how complex nodes can contain both simple- and complex- nodes, and how complex nodes have internal and external outputs. In Figure 2, the node representing the if-statement of the code in Listing 3 shows how the external inputs are often mapped directly to internal outputs, and vica versa with the internal inputs and external outputs.

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 γ-nodes, its inputs (and outputs) are all the dependencies needed for 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. However, the first time the operations of the node are executed, the external inputs are mapped to the internal outputs. Thus, the operations represented by its contained RVSDG subgraph can be executed as a tail-controlled loop.

  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.

```
unsigned long long fac(unsigned int n){
  unsigned int i = 0;
  unsigned long long result = 1;
  do{
    i += 1;
    result *= i;
    } while(i < n);
  }
  return result;
}
```

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

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. This will equate to a *for-loop* if the γ-node and θ-node have the same predicate expression.

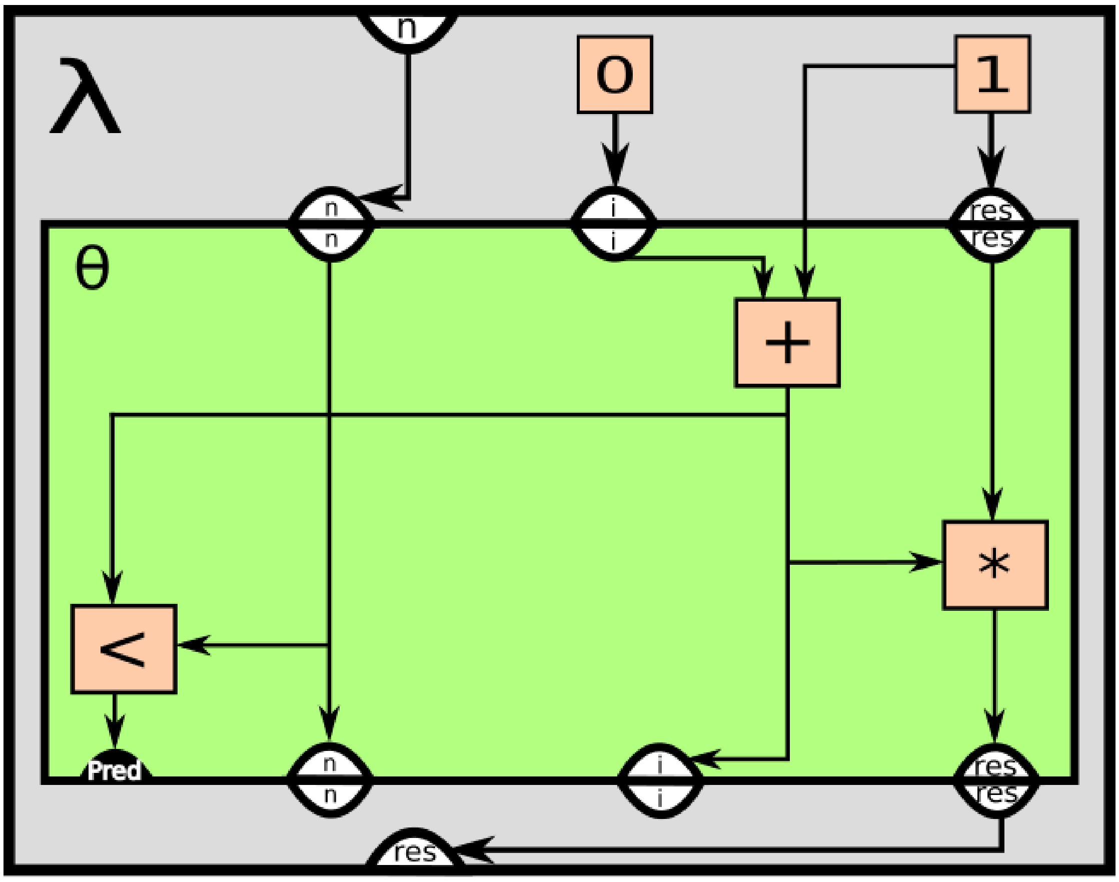

Figure 2: An RVSDG representing the C/C++ program code in Listing 3.

- **λ-nodes: Functions**

  λ-*nodes* represent functions. They contain an RVSDG subgraph representing the body of a function. λ-nodes only have internal inputs, representing the dependencies given from its contained RVSDG. Respectively, its internal outputs give the dependencies needed by the contained RVSDG. However, its external output is what give the *apply*-nodes their first link, enabling them to invoke the function represented by the λ-*node*.

  When the invokation of a λ-*node* is linked with an *apply*-node, the external inputs of the *apply*-node are mapped to the internal outputs of the λ-*node*, and likewise with the external outputs and internal inputs, respectively. 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 computations represented in the body the λ-*node* are mapped to the results of the *apply*-node.

Figure 2 shows an RVSDG representation of an iterative factorial function, with its corresponding C/C++ equicalent code in Listing 3. Figure 3 and Listings 4 also illustrate the workings of λ-nodes through the representation of an RVSDG and the equivalent C/C++ code for a recursive fibonacci function.

- **ϕ-regions: Recursive environments**

  ϕ-*regions* represent recursive environments. They contain at least one recursive λ-*node*. Like the λ-*node*, they have no external inputs. However, the internal outputs of the ϕ-*region* represent the links utilized by the *apply*-nodes contained within to connect with the respective λ-nodes also contained within the same ϕ-*region*.

  The internal inputs of a ϕ-*region* receive the function invocation links from the λ-nodes contained within. The internal inputs map to the external outputs, thus enabling *apply*-nodes outside of the recursive environment to link with the λ-nodes contained within.

  An RVSDG representing the recursive fibonacci function written in C/C++ in Listing 4, illustrates the usage of a ϕ-*region* in Figure 3.

```
unsigned 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 in Figure 3.

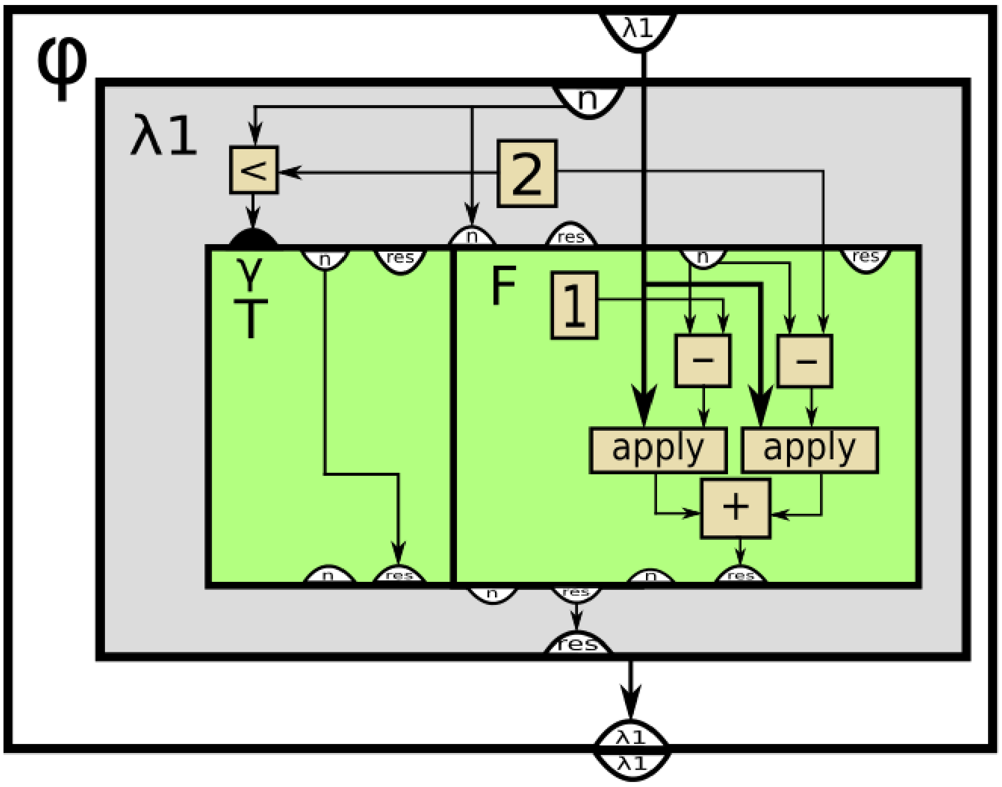

Figure 3: An RVSDG representing the equivalent of the C/C++ code in Listing 4. The RVSDG depicts a ϕ-node containing a λ-node representing a recursive version of a function producing the nth number in the Fibonacci series.

3 The Inliner

Introduce section with how the RVSDG helps with the different steps in the below enumerated list.

The inliner of this project performs the following when given an RVSDG as input:

1. Scan through the RVSDG, finding all ϕ-regions.
   (a) If no ϕ-regions are found, jump to Step 2.
   (b) If ϕ-regions are found, use the approach described in Section 3.3 to fill a list of *loop breakers* with references to the recursive λ-nodes contained within the ϕ-regions, which are *not* to be inlined.
2. Scan through the RVSDG, finding all the *apply*-nodes. Store a reference to each in a new list.
3. Order the list of *apply*-nodes, as discussed in Section 3.1.
4. Look at each *apply*-node in turn in the list from Step 3 and decide whether or not to inline it according to the heuristic implementation discussed in 3.2.
   (a) If the function the *apply*-node invokes is referenced in the *loop breakers* list, or if it does not meet the criteria of the inlining heuristics discussed in Section 3.2, junp to Step 4 and evaluate the next *apply*-node in turn.
   (b) If inlined, make a new list containing any newly copied (inlined) *apply*-nodes.
      i. If this new list is not empty, execute Steps 3$\rightarrow$4b with this new list instead of the list from Step 2. After list is completed, continue with previous list.

3.1 The order of call sites inlined

Need reference to further ideas related to ordering of inlining.

While implementing the inliner of this project, it came to our attention that the order of which the functions are inlined, can have an effect on the compiled program.

The inlining conditions we use as criteria for whether or not to inline, only look at the properties of the function a call site invokes. Hence, when a successive series of functions call one another, we only consider at one at a time. Thus, 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.

Figure 4 illustrates the different outcomes dependent upon the order we visit each call-site (*apply*-node). If our criteria for inlining is that the inlined function does not exceed the inlining condition: *Statement Count* > 4, we can inline $\lambda_1 \Rightarrow \lambda_2 \Rightarrow \lambda_3$. However, if we inline $\lambda_3 \Rightarrow \lambda_2$, then the combined function λ_{2+3} will have a SC exceeding the given limit.

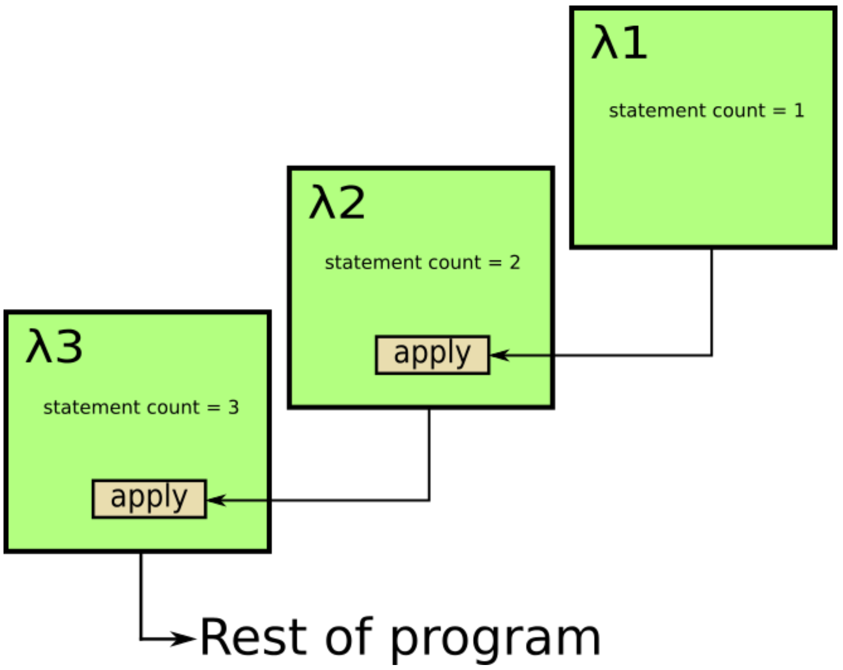

Figure 4: A minimal example of an RVSDG subgraph, showing why the ordering of inlining makes a difference for the resulting program executable.

3.2 Inlining a call site

Utilizing the *inlining conditions* described in Section 4.1, heuristics evaluating the function invoked by each *apply*-nodecan be written in *Conjunctive Normal Form* (CNF). This enables an efficient way to search the parameter space for optimal parameters for the inlining heuristics.

The inliner evaluates each *apply*-node with the given heuristic, and decides whether or not to inline the call site this *apply*-node represents, depending on the properties of the function it invokes.

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

Is the below paragraph too implementation specific?

When a function is inlined, additional *apply*-nodes might be copied into the RVSDG. These are handled as discussed in Step 4(b)i. However, when inlining a call-site, the optimization of an RVSDG performed after each inlining might also remove previously found *apply*-nodes. Hence, care must be taken with any list of references to found *apply*-nodes not yet evaluated by the heuristic.

3.3 Deciding which recursive functions to inline

The inliner evaluates all functions, recursive or not, with the same heuristic, described in Section 3.2. However, the inliner of this project only evaluates *some* of the *apply*-nodes invoking recursive functions, to ensure termination of the compiler.

Describe how we decide which recursive functions we evaluate with the inlining heuristic.

The inliner of this project performs the following when given an RVSDG as input:

1. If ϕ-regions are found, use Tarjan's *Strongly Connected Component* (SCC) algorithm[5] to find any SCCs within each ϕ-region.

    (a) If there exist any λ-nodes which are not part of any SCC in the ϕ-region, add a reference to each and all of these to the list of *loop breakers*, ref. Step 1b from Section 3.

    (b) If any λ-nodes are found to be part of an SCC in the ϕ-region, select one these per SCC to be a *loop breaker*, and add a reference to each of the loop breakers to the list of *loop breakers*.

Explain how when implemented

Hence, the inliner has a list of recursive functions which it knows *not* to inline, to ensure termination of the compilation. All other remaining recursive functions may then be safely inlined with the same criteria as any non-recursive functions.

4 Methodology

Need an introduction here, no?

4.1 Inlining conditions used when inlining

To effectively test for an apt heuristic when deciding whether or not to inline a call site[4], our approach is based on previous work[9][13]. This approach utilizes something we call *Inliner Conditions* (ICs) which evaluate the function invoked by the call site.

Using ICs in this way allows us to write and re-write inlining heuristics effectively, since we can write them using CNF in the following fashion: SC < X || SCC < Y || (SCC < Z && LND > W)

The ICs utilized in this project are the following:

- Statement count (SC): This function property is equates to 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 (LND): 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 (SCC): 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. If the count is 1, then the call site can always be inlined, seeing as there is no risk of code-duplication.
- Parameter count (PC): 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[13].
- Constant parameter count (CPC): 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 (CP): 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[13] 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.

[4]As discussed in Section 3.2.

5 Results

6 Further ideas

6.1 Dynamic profiling and adaptive compilation

Basically re-iterate Waterman's PhD idea.

6.2 Choosing loop breakers more carefully

Bas' MSc. idea

6.3 The ordering of which call sites to order first

Lots to say on ordering of call sites to be inlined

6.3.1 Something

todo[inline]If time permits, I want to briefly discuss the idea of evaluating all the apply nodes (in order), before inlining any. So as to be able to see the total cost of inlining a successive chain of function calls in one direction or the other.

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 [7] 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 [12] 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 [13] 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 [6] expand on Waterman's PhD Thesis [13]. 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 [10] 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 [11] 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 [9] 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] David Cheriton and Robert Endre Tarjan. Finding minimum spanning trees. *SIAM Journal on Computing*, 5(4):724–742, 1976.

[6] 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.

[7] 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.

[8] Bas den Heijer. Optimal loop breaker choice for inlining. Master's thesis, Utrecht University, Netherlands, 2012.

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

[10] 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.

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

[12] 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.

[13] 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.