Inlining in the Jive Compiler Backend

Christian Chavez

Wednesday 29th 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. Compilers use many code optimization techniques, such as *Common Subexpression Elimination* (CSE) and *Dead Code Elimination* (DCE).

Find reference, if time allows.

Another code optimization is *inlining*, which replaces the call site of a function with its body. Listing 1 shows the definitions of functions foo() and bar(). If foo() is inlined into bar, the body of bar() becomes return y + 3 + 2;, allowing *Constant Folding* (CF) to replace 3+2 with 5.

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

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

Listing 1: C/C++ code showing the definitions of foo() and bar() when exemplifying CF.

The benefits of inlining are mainly two-fold: The first one is the removal of function call overhead. Function call overhead is the cost in memory needed on the stack, as well as the CPU cycles needed for setting up the call and performing the call. The second is the potential for unveiling the application of additional optimizations as demonstrated in Listing 1.

The drawbacks of inlining are code-duplication, and an increased compile time. In specific situations, work-duplication can also occur [10]. Listing 2 exemplifies a situation where inlining can lead to code-duplication, if foo() is inlined into bar(). If foo() is called more than once in the program like bar() does, code-duplication will occur. This is due to the big expression e in foo() being copied into bar() as soon as one of the two invocations of foo() is inlined. Hence, if both are inlined, code-duplication still occurs. However, code-duplication might be mitigated if CSE is applicable, and if foo() is static with all its invocations inlined, then the definition of foo() may be removed through the use of DCE.

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

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

Listing 2: C/C++ code showing the definitions of foo() and bar(), when exemplifying code-duplication.

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. Hence, recursive functions need to be handled carefully. The literature proposes two main approaches for inlining recursive functions:

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

2. In a mutually recursive environment, decide on a function to be a *loop breaker*, and never inline it. Any other remaining recursive functions in the mutual recursive environment can

then safely be inlined wrt. termination of the compilation [7][10].

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*[1] (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.

[1]Detailed in Section 2.

2 Background - 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 can be represented as an RVSDG in the figure on the right. The nodes in Figure 1 represent operations in a program, while the edges between the nodes show the dependencies nodes have to each other, thus giving the order of execution.

In all RVSDG examples depicted in this report, the order of inputs in a node goes clockwise. The first input of a node is the one closest to the bottom left corner of the node.

```
x = (y*2) / (z+2);
```

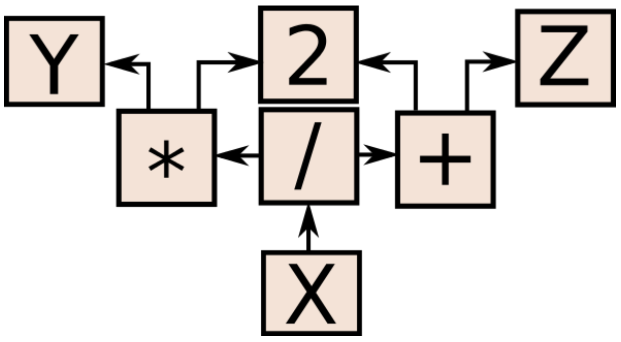

Figure 1: Example of an RVSDG subgraph equivalent to the C/C++ on the left.

2.1 Edges

The RVSDG has two types of edges: data dependence edges and state dependence edges. They represent data and state dependencies operations have to each other, respectively. An example of data dependence edges are the operands used in an addition, such as in Figure 1.

State dependence edges are used to preserve the semantics of the program when the program has side-effecting operations. If there are no data dependencies between operations, state dependence edges can give the needed order of execution for the connected operations.

2.2 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. Figure 1 is an example of an RVSDG containing only simple nodes.

2.2.1 Simple Nodes

One simple node of special interest for this report is the *apply*-node. An *apply*-node represents the call site of a function. The first input argument of an *apply*-node is the function the *apply*-node invokes. The remaining inputs are the arguments to this function. Likewise, its results are the results of the invocation of its function. Order and arity of inputs and outputs need to match the arguments and results of the funtion, respectively.

2.2.2 Complex Nodes

Complex nodes contain one or more RVSDG subgraphs, which is why they are also referred to as *regions*. Differing from the simple nodes with their contained subgraph, complex nodes may besides the normal inputs and outputs also have internal inputs and outputs. Figure 2 shows which inputs/outputs are the external ones, and which are the internal ones. Figure 2 also illustrates how the values of the external inputs are mapped to the internal outputs of each subregion, and vica versa with each subregion's internal inputs being mapped to the external outputs.

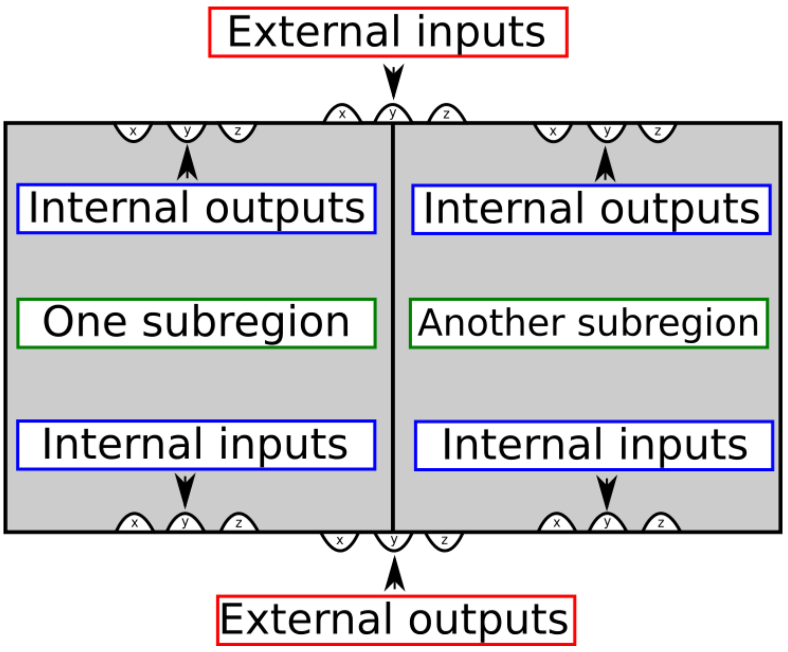

Figure 2: A minimal example of a complex node, showing which inputs/outputs are external/internal, and how they can have multiple subregions.

As Figure 2 also shows, some complex nodes may also have multiple *subregions*. If a complex node has more than one subregion, the arity and order of all the internal inputs/outputs must match between all subregions, as well as match the arity and order of the external inputs/outputs of the complex node.

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. Each subregion represents one case. 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 3 for an example of a γ-node.

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

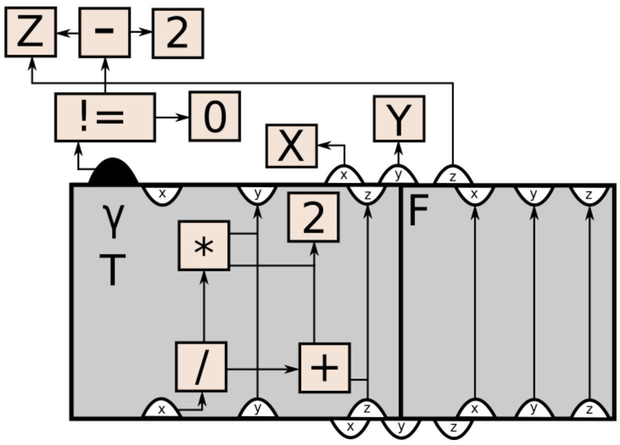

Figure 3: Example of an RVSDG subgraph on the right, depicting a γ-node equivalent to the C/C++ if-statement on the left.

- **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.

  The first time the operations of the node are executed, the external inputs are mapped to the internal outputs, as Figure 2 exemplifies. This enables the complex node's contained RVSDG subgraph to execute with the values given as external inputs to the θ-node.

  However, 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. Thus, the operations represented by its contained RVSDG subgraph can be executed as a tail-controlled loop.

  Finally, when the predicate evaluates to false, the internal inputs are mapped to the external outputs of the θ-node instead of the internal outputs.

  A θ-node is equivalent to a *do-while* loop in C/C++, as shown in Figure 4.

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

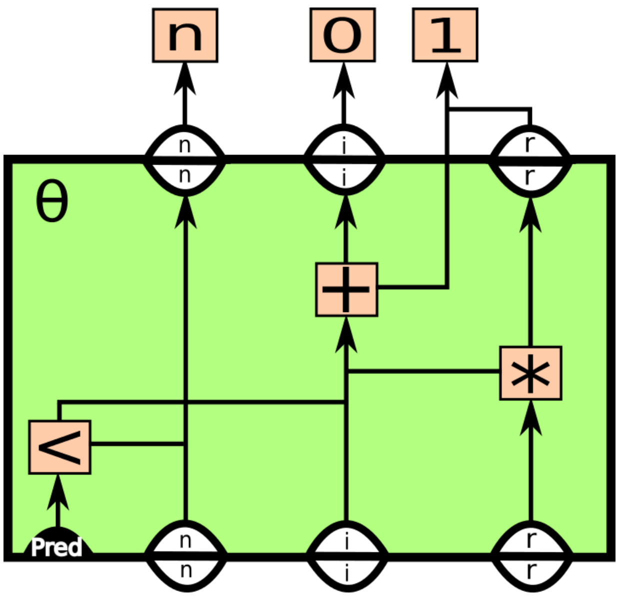

Figure 4: An RVSDG subgraph on the right, depicting a θ-node equivalent to the C/C++ do-while loop on the left.

A *for-loop* can be represented by putting a θ-node inside of the *true* clause of a γ-node conaining no nodes in the subregion representing the *false* clause, as long as the γ- and θ-nodes share the same predicate.

- **λ-nodes: Functions**

  λ-*nodes* represent functions. A λ-node contains an RVSDG subgraph representing the body of a function. The internal inputs of a λ-node represents the results of the function. Respectively, its internal outputs represent the arguments of the function. While λ-nodes don't have any external inputs, their external output are what give the *apply*-nodes their first input, enabling them to invoke the function represented by the λ-node.

  The arity and order of a λ-node's internal inputs and outputs must match the arity and order of the external inputs and outputs of all connected *apply*-nodes.

  Hence, when an *apply*-node connected with a λ-node is executed, the external inputs of the *apply*-node are mapped to the internal outputs of the λ-node, before the λ-node is invoked. Likewise the external outputs of the *apply*-node are mapped internal inputs of the λ-node when the operations of the λ-node have been executed. See Figure 5 for an example of a λ-node.

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

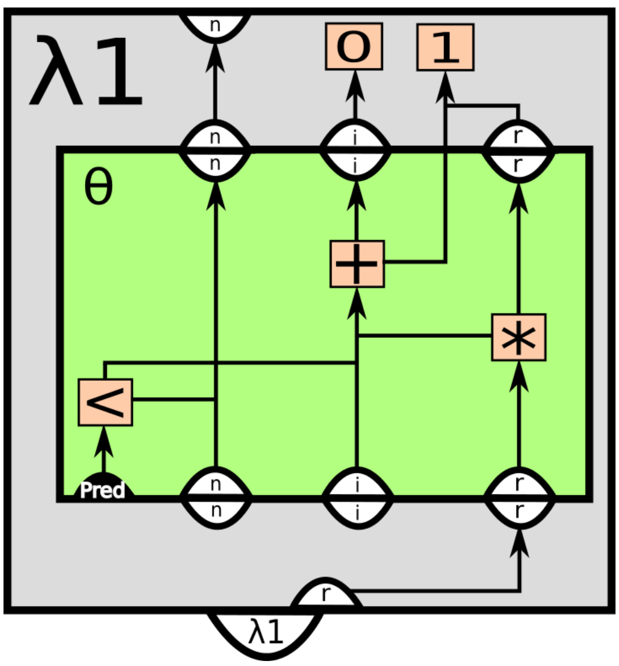

Figure 5: Example of an RVSDG subgraph on the right, depicting a λ-node equivalent to the C/C++ function on the left.

- **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 connect with the λ-nodes, as depicted in Figure 6.

```
unsigned long long
fac(unsigned int n){
  unsigned long long r;
  if(n > 1){
    r = n*fac(n-1);
  } else{
    r = 1;
  }
  return r;
}
```

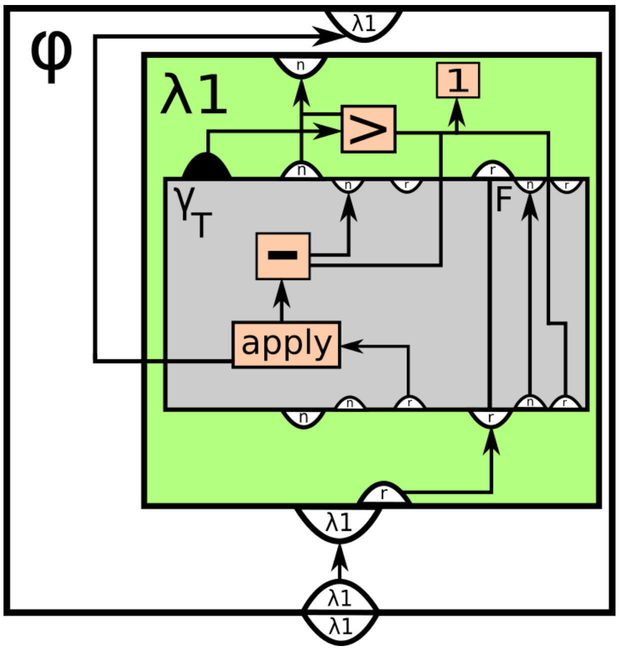

Figure 6: Example of an RVSDG subgraph on the right, depicting a ϕ-region equivalent to the recursive C/C++function on the left.

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. For all recursive environments (ϕ-regions):

   (a) Use the approach described in Section 3.3 to fill a list of *loop breakers*. These λ-nodes are *not* to be inlined.

2. Scan through the RVSDG, finding all *apply*-nodes. Exclude all function calls to loop breakers, calls invoking functions that are not-statically known, or external functions.

3. Make a list of the *apply*-nodes found in Step 2, and order the list of according to the heuristics discussed in Section 3.2. The order of *apply*-nodes inlined can affect the amount of *apply*-nodes inlined, even when each *apply*-node is evaluated with the same heuristic.

4. Look at each *apply*-node in turn from the list made in Step 3 and decide whether or not to inline it according to the heuristic discussed in 3.1:

   (a) If the *apply*-node is inlined, add any newly copied (inlined) *apply*-nodes, following the same criteria as used in Step 2, to the list of *apply*-nodes. Continue with Step 3.

   (b) If the *apply*-node is not inlined, continue with Step 4, evaluating the next *apply*-node.

5. When the inliner reaches the end of the list, no more *apply*-nodes have been inlined, and the inliner is finished.

3.1 Inlining a call site

To effectively test for an apt heuristic when deciding whether or not to inline a call site, our approach is based on previous work [8][12]. This approach utilizes something we call *Inliner Conditions* (ICs) which evaluate the function invoked by the call site. An IC evaluates a specific property of the function invoked by *apply*-node.

The inliner used several different ICs, *Statement Count* (SC) and *Static Call Count* (SCC) being among these. 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)

Utilizing the inlining conditions described in Section 4.1, heuristics evaluating the function invoked by each *apply*-node can 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 for evaluating a call site on after testing.

3.2 The order of call sites inlined

Need reference to further ideas related to ordering of inlining.

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 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 7 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: SC > 4, we can inline $\lambda_1 \Rightarrow \lambda_2 \Rightarrow \lambda_3$. However, if we inline the functions in the opposite order, $\lambda_3 \Rightarrow \lambda_2$, then the combined function λ_{2+3} will have a SC exceeding the given limit.

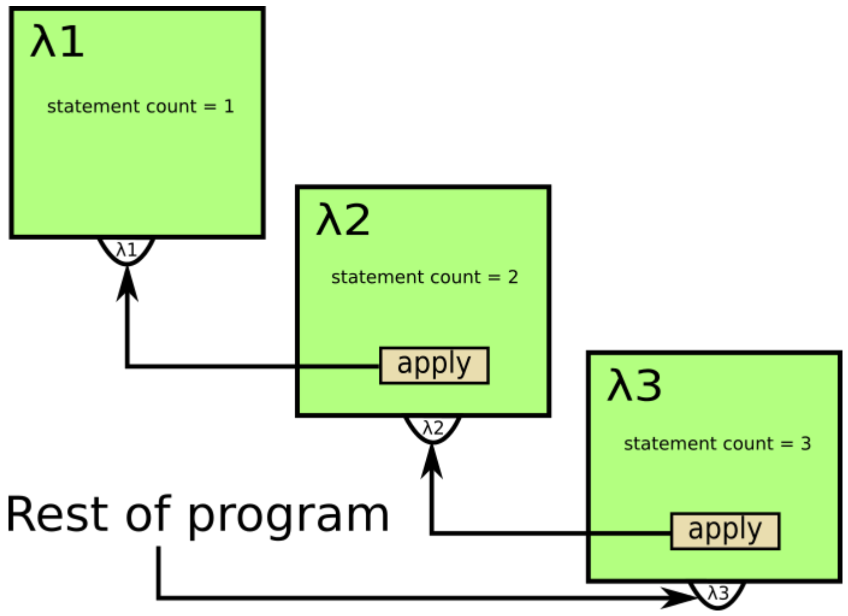

Figure 7: A minimal example of an RVSDG subgraph, depicting a function call order in a program.

3.3 Deciding which recursive functions to inline

The inliner evaluates all functions, recursive or not, with the same heuristic, described in Section 3.1. 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.

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 The Inlining Conditions (ICs)

The ICs utilized in this project are the following:

- **Statement count (SC)**

  This function property 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 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 [12].

- **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.s

- **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 [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

Make a subsection or paragraph showing the results of *apply*-nodes which link to static calls, vs all calls.

6 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 normal form (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.

7 Conclusion

8 Further ideas

8.1 Dynamic profiling and adaptive compilation

Basically re-iterate Waterman's PhD idea.

8.2 Choosing loop breakers more carefully

Bas' MSc. idea

8.3 The ordering of which call sites to order first

Lots to say on ordering of call sites to be inlined

8.3.1 Something

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.

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.