Inlining in the Jive Compiler Backend

Christian Chavez

Thursday 21st May, 2015

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) [1, Ch. 8.5] and *Dead Code Elimination* (DCE) [1, Ch. 8.5].

Another code optimization is *inlining* [1, Ch. 12.1], which replaces the call site of a function with its body. Listing 1 shows the definition of functions `A()` and `B()`. If `A()` is inlined into `B()`, the body of `B()` becomes `return y + 3 + 2`, permitting *Constant Folding* (CF) [1, Ch. 8.5] to replace `3+2` with `5`.

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

int B(int y){
  return A(y) + 2;
}
```

Listing 1: C/C++ code showing the definitions of `A()` and `B()` when exemplifying inlining and 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 and performing the call. The second is the potential for unveiling the application of additional optimizations such as CF demonstrated in Listing 1.

The drawbacks of inlining are code duplication, and an increased compile time. In specific situations, work duplication can also occur [11]. Listing 2 exemplifies a situation where inlining can lead to code duplication if either of `C()`s invocations in `D()` are inlined. This is due to the big expression `e` being copied into `D()` as soon as one or more of `C()` invocations are inlined. However, duplication might be mitigated if CSE is applicable. Also, if both of `C()`s invocations are inlined, then the definition of `C()` may be removed through the application of DCE since `C()` is `static`.

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

int D(int x, int y){
  return C(x) + C(y);
}
```

Listing 2: C/C++ code showing the definitions of C() and D(), when exemplifying inlining and 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 mainly two approaches for inlining recursive functions:

1. Avoid non-termination of the compilation by only inlining recursive functions to a certain depth [11, 12], and therefore breaking the recursive cycle.

2. In a mutually recursive environment, decide on one or more functions to be *loop breakers*, and mark them to never be inlined. Loop breakers are chosen so that the recursive call cycle will be broken in the mutually recursive environment. Having chosen correct loop breakers permits inlining of the remaining recursive functions in the mutually recursive environment, *without* risking non-termination of the compiler [8, 11].

This report describes the construction of an inliner for the Jive compiler backend, detailing its design and architecture. Jive uses an *intermediate representation* (IR) called the *Regionalized Value State Dependence Graph* (RVSDG) [2].

The RVSDG described in Section 2, is a *demand-based* and *directed acyclic graph* (DAG) where nodes represent computations, and edges represent the dependencies between these computations.

Section 3 details the scheme of our project, and explains how the inliner is able to handle recursive functions, as well as 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 section.

In Section 4, we describe how the implemented inliner is evaluated, as well as the inlining heuristics we chose for the SPEC2006 Benchmark Suites used for testing. Section 5 then reports on the characteristics of the SPEC2006 Benchmark Suites used for the testing, as well as the results of the inliner. Also in Section 5, we focus

on how different heuristics have different consequences, such as code duplication, and others.

In Section 6 we summarize the existing related literature found in our background study for this project. Following, in Section 7, we conclude, before we finally discuss ideas for potential further research in Section 8. A detailed description of the project assignment can be found in Appendix A.

Figure out what to write about the below, and where in the report to put it...

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.

2 The Regionalized Value State Dependence Graph

The *Regionalized Value State Dependence Graph* [2] (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. 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);
```

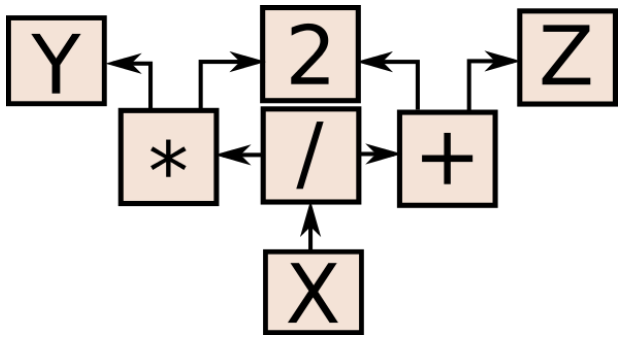

Figure 1: Example of an RVSDG subgraph equivalent to the C/C++ on the left.

2.1 Edges

The RVSDG has data dependence edges and state dependence edges, representing 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. Figure 2 illustrates the use of state dependence edges, depicted as dotted edges.

```
*x += 7;
*y += 7;
```

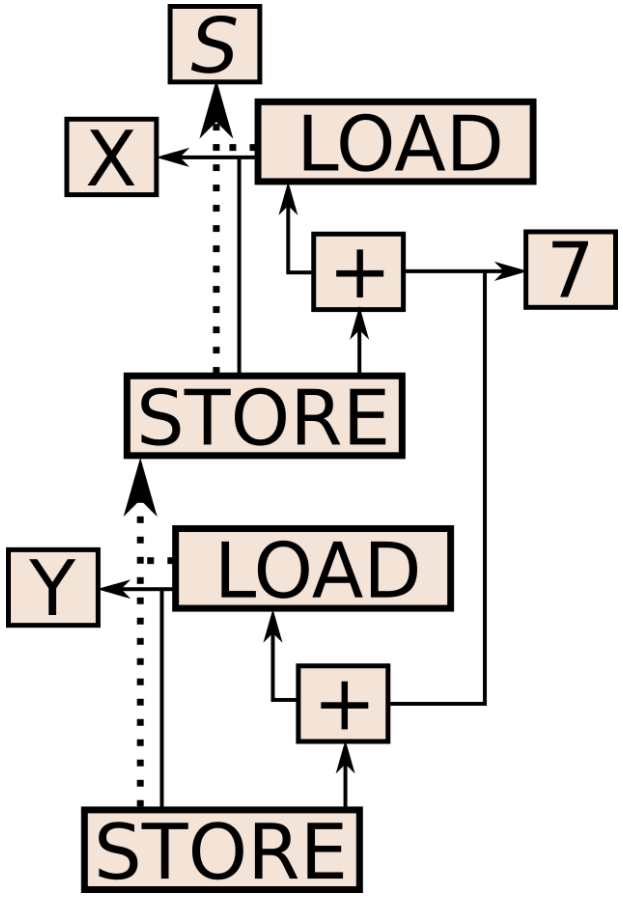

Figure 2: Example of an RVSDG subgraph equivalent to the C/C++ on the left.

The *S*-node in Figure 2 supplies the needed state edge for x's *load*- and *store*-nodes.

2.2 Nodes

The RVSDG has two kinds of nodes: simple nodes and complex nodes. The arity and order of inputs and outputs of any RVSDG node need to match the operation.

2.2.1 Simple Nodes

Simple nodes are used to represent primitive operations, such as addition and substraction. Figure 1 is an example of an RVSDG containing only 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 3 shows which inputs/outputs are the external ones, and which are the internal ones. Figure 3 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.

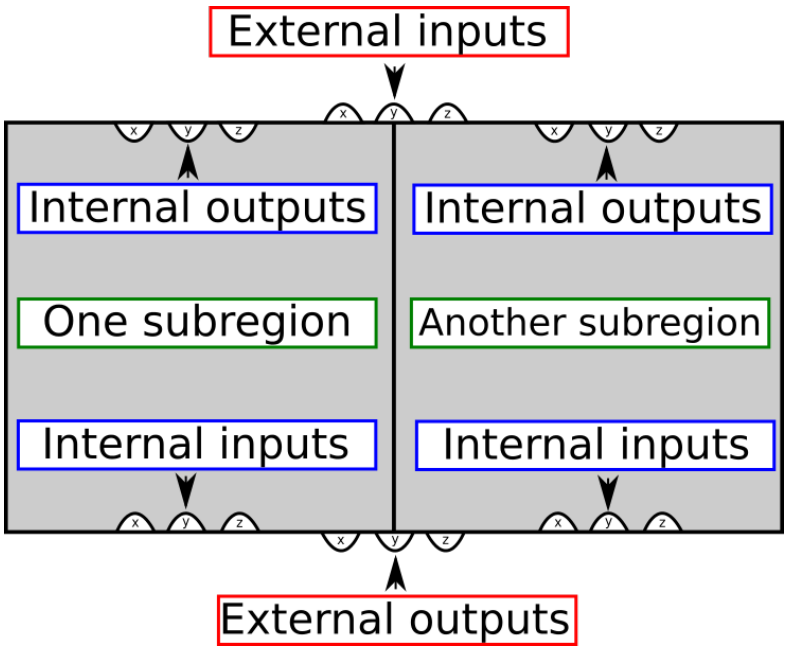

Figure 3: A minimal example of a complex node, showing which inputs/outputs are external/internal, and how they can have multiple subregions.

As Figure 3 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 4 for an example of a γ-node.

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

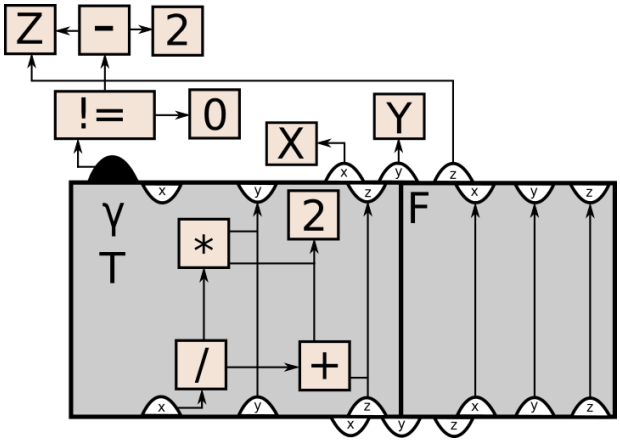

Figure 4: Example of an RVSDG subgraph on the right, depicting a γ-node equivalent to the C/C++ if-statement on the left.

- **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 body of the loop is executed, the external inputs are mapped to the internal outputs, as Figure 5 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 are 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 5.

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

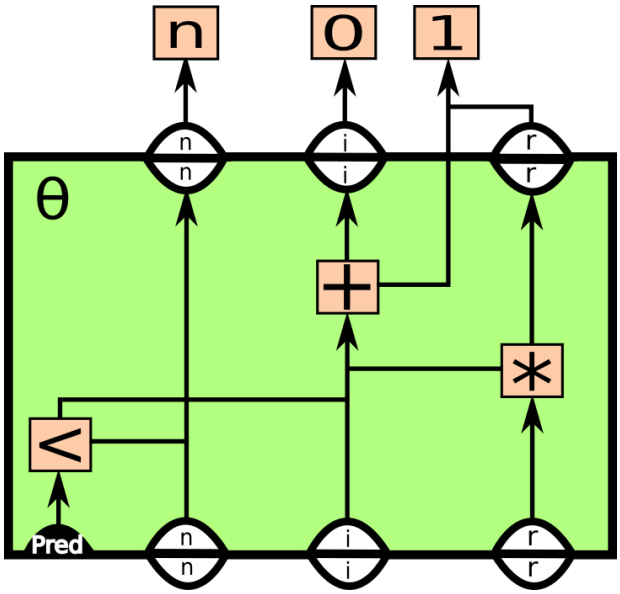

Figure 5: An RVSDG subgraph on the right, depicting a θ-node equivalent to the C/C++ do-while loop on the left.

A *head-controlled* loop can be represented by putting a θ-node inside the subregion representing the *true* clause of a γ-node. Additionally, both the γ-node's first input and the first internal input of the θ-node need to share the same predicate. Finally, the γ-node's subregion representing the *false* clause cannot contain nodes.

- **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 internal inputs of the λ-node are mapped to the external outputs of the *apply*-node when the operations of the λ-node have been executed. See Figure 6 for an example of a λ-node.

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

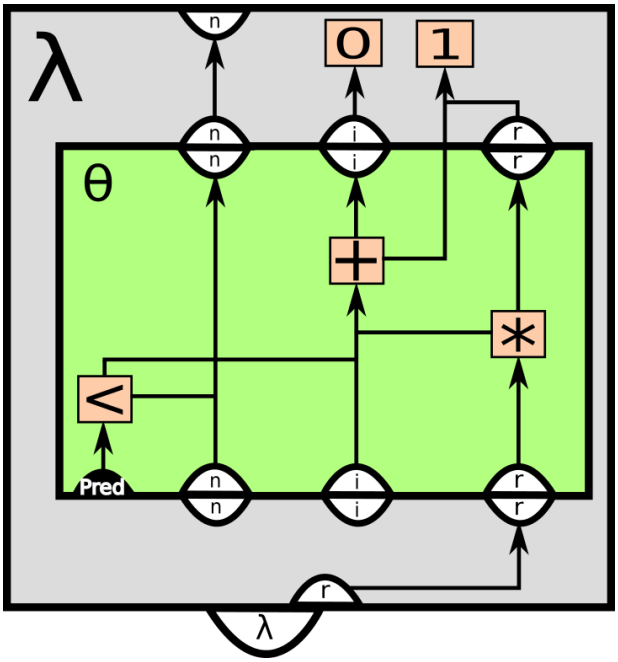

Figure 6: Example of an RVSDG subgraph on the right, depicting a λ-node equivalent to the C/C++ function on the left.

- **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.

  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 7.

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

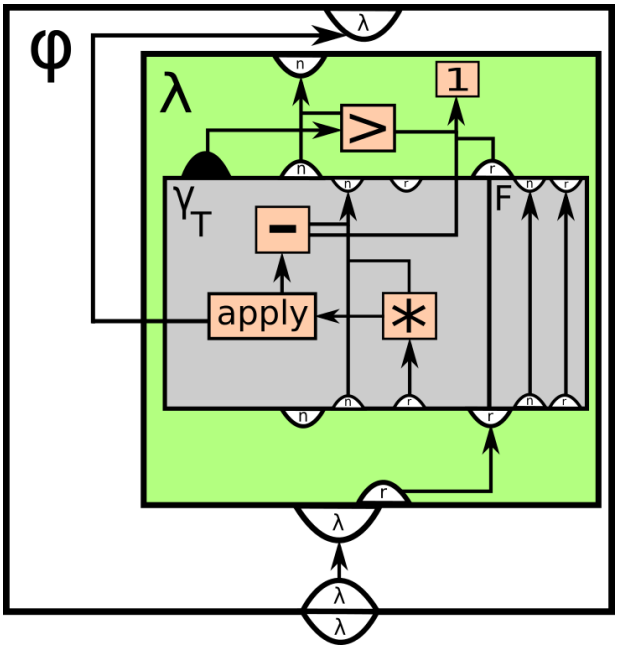

Figure 7: Example of an RVSDG subgraph on the right, depicting a ϕ-region equivalent to the recursive C/C++function on the left.

3 The Inliner

Our inliner, written in C++, utilizes Jive integrations, and is compiled with Jive. This section details the actions performed by the inliner, and the reasoning behind these. First we list the action the inliner performs, before we go more into depth on the reasoning behind them.

The following heuristic is executed by the inliner of this project when given an RVSDG as input:

1. For all recursive environments (ϕ-regions):
   (a) Use the approach described in Section 3.1 to fill a list *loop breakers*. These functions (λ-nodes) are *not* to be inlined.
2. Scan through the RVSDG, finding all call sites (*apply*-nodes). Exclude all function calls to loop breakers, calls invoking functions which are not statically known, or external functions.
3. Make a list of the *apply*-nodes found in Step 2, and order the list according to the heuristics discussed in Section 3.2. The order of *apply*-nodes inlined can affect the total amount of *apply*-nodes inlined, even when all of the *apply*-nodes are 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 Section 3.3:
   (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 Deciding which recursive functions to inline

The inliner visits all *apply*-nodes invoking statically known functions in the RVSDG given as input, whether they invoke a recursive function or not. Before the inliner

collects any *apply*-nodes, it first finds all the ϕ-regions in the RVSDG. In each ϕ-region, the call graph made by the λ-node's *apply*-nodes inside always make a Strongly Connected Component (SCC).

Hence, for each ϕ-region, the inliner chooses one λ-node, and performs the following steps:

1. Mark the λ-node as "visited".
2. Collect all the *apply*-nodes contained within this λ-node.
3. If one of the collected *apply*-node invokes another λ-node from within the same ϕ-region, check if the λ-node is marked as "visited":
   (a) If the invoked λ-node is marked as "visited", add it to the list of *loop breakers*.
   (b) If the invoked λ-node is *not* marked as "visited", recursively perform the steps of this list on that λ-node.

In this manner it is guaranteed, by virtue of the SCC the call graph inside a ϕ-region makes, that all λ-nodes are visited. It is also guaranteed that for every cycle in the SCC, there is at minimum *one* λ-node marked as a loop breaker. As such, the inliner knows which recursive functions it must never attempt to inline, so as to ensure termination of the compilation. The remaining λ-nodes residing in ϕ-regions not marked as loop breakers may then be inlined according to the same criteria as any non-recursive function.

While there are better ways [8] to choose loop breakers[1], we unfortunately did not have the time to implement one for this project.

Put a reference to Section 8.2 here?

3.2 The order of call sites inlined

The order in which the *apply*-nodes inlined are inlined, can make a difference for not only *how many* *apply*-nodes are inlined, but also *which* that get inlined. To illustrate this, let us give the reader the following example.

As mentioned, the heuristic deciding whether or not to inline a specific *apply*-node is the same for all the *apply*-nodes collected from the RVSDG given as input to the inliner. Hence, if the heuristic deciding whether an *apply*-node is inlined or not depends upon whether the *internal node count* of the function it invokes

[1] So as to minimize the amount of loop breakers per cycle in the ϕ-region.

is less than four, the example illustrated by Figure 8 will either inline *both* of the depicted *apply*-nodes, or just one of them.

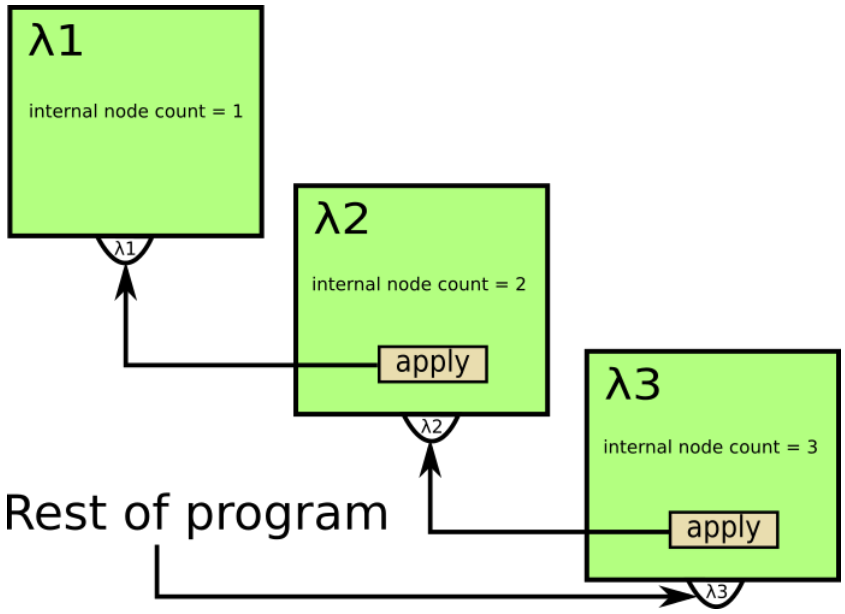

Figure 8: A minimal example of an RVSDG subgraph, depicting a function call order in a program.

If the inliner evaluates the *apply*-nodes in Figure 8 in a top-down order, λ_1 can be inlined into λ_2. The newly created λ_{1-2} with an internal node count of 3 can be inlined into λ_3, resulting in all of them being inlined into a new single function; λ_{1-2-3}.

However, if the inliner evaluates the *apply*-nodes in Figure 8 in a bottom-up order instead, λ_3 can be inlined into λ_2. But the newly created function λ_{2-3}'s amount of nodes contained within exceeds 3, and can hence not be inlined into λ_1.

This constructed example assumes the node count contained within each function created by inlining to be the sum of the itnernal node counts of the original function and the one inlined. In other words, no optimizations or other compiler techniques are assumed performed on the RVSDG during the traversal of the apply-nodes, or before/after each inlining.

As our constructed example shows, the order in which the *apply*-nodes being inlined are visited matter. Because of this, our inliner is able to traverse the *apply*-nodes of the RVSDG given as input in either a top-down order, or a bottom-up order. Section 8.3 discusses ideas for potential work for future research regarding the impact of the order of inlined call sites.

3.3 Inlining a call site

While there exist several [11, 6] ways in literature in which inlining is performed, we decided upon our chosen approach [13] because it permits effective testing for an apt heuristic when deciding on whether or not to inline a call site.

As mentioned, all *apply*-nodes eligible for inlining get inlined based on the same heuristic per run of our inliner. The heuristic is based on *Inliner Conditions* (ICs), such as the following:

- *Node Count* (NC)

  The amount of nodes (operations) contained within the function invoked by the *apply*-node.

- *Loop Nesting Depth* (LND)

  The amount of nested loops the *apply*-node resides within.

- *Static Call Count* (SCC)

  The total amount of *apply*-nodes invoking the same function in the RVSDG given as input.

These ICs and others described in Section 4.1 allow us to write and re-write the inlining heuristic effectively, by letting us write them using *Conjunctive Normal Form* (CNF). Thus, the CNFs written for our inliner heuristic are written in the following fashion: (NC $<$ X || SCC $<$ Y || (SCC $<$ Z && LND $>$ W)), where X, Y, Z, and W are placeholder variables.

While the work of Waterman [13], which our approach is based upon, utilizes a hillclimber algorithm to find the most optimal values for the placeholder variables for each individual program, we did not have sufficient time to implement a hillclimber algorithm in this project.

Still, while the placeholder variables used in our CNFs need to be hardcoded for each run of our inliner, this approach still permits us to search the parameter space for decent parameters for the inlining heuristic.

4 Methodology

Using the *Inliner Conditions* (ICs) described in Section 4.1, the inliner of this project has been tested using C-language programming files from the SPEC2006 Benchmark Suite. This section first details the ICs, before explaining how the files tested from the SPEC2006 Benchmark Suite were chosen. Finally, we list the CNFs for each group of files from the SPEC2006 Benchmark Suite the inliner ran said files with.

4.1 The Inlining Conditions (ICs)

The ICs utilized in this project are as follow:

- **Node Count (NC)**

  This function property equates to the number of C/C++ statements contained within a function. A function's node count is an IC we want to utilize because it gives us an idea of the size of the duplication if we inline the function.

- **Loop Nesting Depth (LND)**

  This call site 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 invoke this function in the program. If this count is low, it may be worth inlining all the call sites and eliminating the original function. For example, if the count is 1, then the call site can always be inlined, seeing as there is no risk of 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 node 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 Node (CIN)**

  This function property tells us how many call sites are located inside the function the call site being evaluated invokes. Hence, it enables the finding of 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.

4.2 The SPEC2006 Benchmark Suite files

The SPEC2006 Benchmark Suite files were chosen with the following criteria:

1. The Benchmark Suite program code files were written in 32-bit C.
2. Clang-3.4 (on Ubuntu 14.04) was able to convert the inputted C files to LLVM IR assembly code with the `-S` and `-emit-llvm` flags.
3. Jive was able to interpret all of the assembly commands in the outputted .ll files and construct RVSDGs from them.
4. The files could to be tested within a time limit of 200 seconds, executed single-process, on a Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz, with 3072 KB cache size, Ubuntu 14.04 64-bit Linux distro.

Confirm with Nico that there were no more requirements.

With these requirements, files from the following benchmarks were used for testing:

- 400.perlbench
- 401.bzip2
- 403.gcc
- 429.mcf
- 433.milc
- 435.gromacs
- 445.gobmk
- 456.hmmer
- 458.sjeng
- 462.libquantum
- 464.h264ref
- 470.lbm
- 482.sphinx3

See Appendix B for the complete list of .c files used for testing from each of the above benchmarks from SPEC2006.

4.3 CNF clauses common for all SPEC2006 Benchmark Suites used

The heuristics used to decide whether or not to inline a call site are written with boolean logic in Conjunctive Normal Form (CNF) using the ICs from Section 4.1.

However, a CNF expression whose clauses are ideal for one program will not be ideal for all programs. For some programs, it might even lead to a worse executable than if no inlining had been performed at all. Hence, for each SPEC2006 benchmark suite's .c files which passed the requirements listed in Section 4.2, we profiled the .c files and found the following CNFs for each set to be the most decent ones we could find manually. See Section 5.1 for the data found when profiling the SPEC2006 Benchmark Suite files.

While a certain CNF may be ideal for the optimization of one program, the same does not hold for all. Even so, there are some CNF clauses which are worth keeping in all the CNFs used in our testing. These are as follow:

- $SCC == 1$

  This clause simply states that if there's only *one* call site invoking this function, then it can be inlined without worry of code duplication or work duplication.

- Think of more...?

4.4 SPEC2006 Benchmark Suite's individual CNFs

In addition to the CNF clauses from Section 4.3 and through manual testing of hardcoded values, the following CNFs were the most optimal ones we found for each individual SPEC2006 Benchmark Suite:

Confirm with Nico that this is okay.

- 400.perlbench
- 401.bzip2
- 403.gcc
- 429.mcf
- 433.milc
- 435.gromacs
- 445.gobmk
- 456.hmmer
- 458.sjeng
- 462.libquantum
- 464.h264ref
- 470.lbm
- 482.sphinx3

Ideally, we should not have looked for the most optimal CNFs ourselves manually. While it was our hope to instead implemented a hillclimber [6] as mentioned in Section 3.3, the time allotted for this project was not enough.

5 Results

Ask Nico whether result needs an introduction at all...

5.1 Profiling results

Remember to mention/show the results of the amount of *apply*-nodes which link to static calls, vs all calls, and etc. Data which does not directly relate to the ICs.

- 400.perlbench
- 401.bzip2
- 403.gcc
- 429.mcf
- 433.milc
- 435.gromacs
- 445.gobmk
- 456.hmmer
- 458.sjeng
- 462.libquantum
- 464.h264ref
- 470.lbm
- 482.sphinx3

5.2 Inlining results

Replace with table and grapfs, but keep for LaTeX ease of use until then.

- 400.perlbench
- 401.bzip2
- 403.gcc
- 429.mcf
- 433.milc
- 435.gromacs
- 445.gobmk
- 456.hmmer
- 458.sjeng
- 462.libquantum
- 464.h264ref
- 470.lbm
- 482.sphinx3

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 [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 [4] 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 [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. [3] 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 [5] 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

7.1 Acknowledgments

Nico Reissmann, Magnus Jahre, and Christian Chavez are with the Norwegian University of Science and Technology (NTNU).

8 Further Work

In this section we've collected our ideas for potential future work discovered while working on the inliner for this project. First, in Section 8.1 and Section 8.2, we will discuss potential future work already described in literature [6, 13, 8], before finishing with more original ideas for future work in Section 8.3.

Add final section reference if written.

8.1 Dynamic Profiling, Parametrization, and Adaptive Compilation

As mentioned in Section 3.3 and Section 4.4, Waterman [13] implemented a hill-climber algorithm to dynamically and adaptively re-compile a program so as to find the optimal inlining heuristic for each compilation.

This is something we wanted to implement for the inliner of this project, but did not have the resources for. It is our belief that the strengths of Jive's RVSDG approach to compilation/inlining can be further exploited if the choosing of inlining heuristic was done dynamically and adaptively like Waterman proposes.

The work of Cooper, et al. [6] shows how the parametrization of the inliner Waterman [13] proposed can easily be expanded upon and added to, making it a more powerful tool. This is also something we believe that Jive could benefit from wrt. to inlining, but did not have the resources to implement.

8.2 Choosing loop breakers more carefully

When deciding upon recursive functions to be marked as *loop breakers*, discussed in Section 3.1, our approach does not choose these carefully. Ideally, an approach such as the one Bas [8] proposes, would be more ideal because it makes an effort in choosing the minimum amount of loop breakers needed.

Again, due to time and resources, we did not get the chance to implement Bas's suggested approach based on the Directed Blackout Feedback Vertex Set problem. However, it's utility in inlining functions belonging to recursive environments is obvious, and its implementation in Jive's inliner would have been advantageous.

8.3 The ordering of which call sites to order first

To our knowledge, no other literature exist on the subject of the consequences of the ordering of inlined functions. As illustrated with the example from Section 3.2,

the order in which inlines are performed affects not only the total count of inlines, but also *which* call sites are inlined.

Since the RVSDG gives us a directed call graph for each program compiled, other approaches can be tested. One such approach could be to see whether there are properties ignored/gained which are inherent to the ordering of the call sites in a top-down ordering compared to a bottom-up ordering.

Another approach could be to find the aggregate cost of inlining sequential call sites: profiling them first, before inlining them. Would this approach hide worthwile potential optimizations found if the call sites in the chain were inlined one by one? Or would more worthwile optimizations be found, if this approach gives an increased chance of applying optimization techniques like *Common Subexpression Elimination*?

List of Figures

Listings

9 References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] 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.

[3] 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.

[4] 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.

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

[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.

B SPEC2006 Benchmark Suite files used

- 400.perlbench
  - deb.ll
  - locale.ll
  - perlapi.ll
  - specrand.ll
  - stdio.ll
- 401.bzip
  - crctable.ll
  - randtable.ll
- 403.gcc
  - alloca.ll
  - asprintf.ll
  - c-convert.ll
  - c-errors.ll
  - cppdefault.ll
  - cpperror.ll
  - cpphash.ll
  - dwarfout.ll
  - fibheap.ll
  - genrtl.ll
  - getpwd.ll
  - hex.ll
  - hooks.ll
  - insn-peep.ll
  - intl.ll
  - langhooks.ll
  - lbasename.ll
  - line-map.ll
  - main.ll
  - mbchar.ll
  - mkdeps.ll
  - obstack.ll
  - partition.ll
  - print-rtl.ll
  - reorg.ll
  - rtl-error.ll
  - safe-ctype.ll
  - sdbout.ll
  - sibcall.ll
  - varray.ll
  - vasprintf.ll
  - version.ll
  - vmsdbgout.ll
  - xcoffout.ll
  - xmalloc.ll
  - xstrerror.ll
- 429.mcf
  - mcfutil.ll
  - output.ll
  - pbeampp.ll
  - pbla.ll
  - pflowup.ll
  - psimplex.ll
  - pstart.ll
  - treeup.ll
- 433.milc
  - addmat.ll
  - addvec.ll
  - byterevn.ll
  - clear_mat.ll
  - clearvec.ll
  - d_plaq4.ll
  - gauge_info.ll
  - gaussrand.ll
  - grsource_imp.ll
  - io_nonansi.ll
  - layout_hyper.ll
  - l_su2_hit_n.ll
  - make_ahmat.ll
  - make_lattice.ll
  - m_amat_hwvec.ll
  - m_amatvec.ll
  - m_amv_4dir.ll
  - m_amv_4vec.ll
  - mat_invert.ll
  - m_mat_an.ll
  - m_mat_hwvec.ll
  - m_mat_na.ll
  - m_mat_nn.ll
  - m_matvec.ll
  - m_mv_s_4dir.ll
  - msq_su3vec.ll
  - rand_ahmat.ll

  - ranmom.ll
  - ranstuff.ll
  - realtr.ll
  - rephase.ll
  - reunitarize2.ll
  - r_su2_hit_a.ll
  - s_m_a_mat.ll
  - s_m_a_vec.ll
  - s_m_mat.ll
  - s_m_s_mat.ll
  - s_m_vec.ll
  - su3_adjoint.ll
  - su3mat_copy.ll
  - su3_proj.ll
  - su3_rdot.ll
  - sub4vecs.ll
  - submat.ll
  - subvec.ll
  - uncmp_ahmat.ll
  - update_h.ll
  - update.ll
  - update_u.ll
- 435.gromacs
  - 3dview.ll
  - atomprop.ll
  - binio.ll
  - block_tx.ll
  - buffer.ll
  - calcmu.ll
  - calcvir.ll
  - comlib.ll
  - ebin.ll
  - f77_wrappers.ll
  - ffscanf.ll
  - futil.ll
  - glaasje.ll
  - ifunc.ll
  - innerc.ll
  - invblock.ll
  - macros.ll
  - mdrun.ll
  - memdump.ll
  - mvdata.ll
  - names.ll
  - network.ll
  - nsb.ll
  - rando.ll
  - random.ll
  - rbin.ll
  - rdgroup.ll
  - rmpbc.ll
  - smalloc.ll
  - strdb.ll
  - string2.ll
  - symtab.ll
  - synclib.ll
  - tgroup.ll
  - typedefs.ll
  - vec.ll
  - viewit.ll
  - wgms.ll
  - wnblist.ll
  - xdrd.ll
  - xtcio.ll
  - xvgr.ll
- 445.gobmk
  - engine/hash.ll
  - engine/movelist.ll
  - engine/sgffile.ll
  - patterns/fuseki13.ll
  - patterns/fuseki19.ll
  - patterns/fuseki9.ll
  - patterns/transform.ll
  - sgf/sgfnode.ll
  - sgf/sgftree.ll
  - sgf/sgf_utils.ll
  - utils/getopt1.ll
  - utils/random.ll
- 456.hmmer
  - a2m.ll
  - alphabet.ll
  - clustal.ll
  - eps.ll
  - file.ll
  - gki.ll
  - iupac.ll
  - masks.ll
  - mathsupport.ll
  - misc.ll
  - phylip.ll
  - plan9.ll
  - revcomp.ll
  - rk.ll

  - seqencode.ll
  - sqerror.ll
  - squidcore.ll
  - sre_ctype.ll
  - sre_math.ll
  - sre_random.ll
  - stack.ll
  - translate.ll
  - types.ll
  - vectorops.ll
- 458.sjeng
  - draw.ll
  - ecache.ll
  - rcfile.ll
- 462.libquantum
  - classic.ll
  - expn.ll
  - oaddn.ll
  - omuln.ll
  - qec.ll
  - qft.ll
  - specrand.ll
  - version.ll
- 464.h264ref
  - specrand.ll
- 470.lbm
  - main.ll
- 482.sphinx3
  - agc.ll
  - ascr.ll
  - beam.ll
  - bio.ll
  - case.ll
  - ckd_alloc.ll
  - cmn.ll
  - cmn_prior.ll
  - err.ll
  - fillpen.ll
  - gs.ll
  - hash.ll
  - heap.ll
  - hmm.ll
  - io.ll
  - logs3.ll
  - parse_args_file.ll
  - profile.ll
  - spec_main_live_pretend.ll
  - specrand.ll
  - str2words.ll
  - tmat.ll
  - unlimit.ll