Inlining in the Jive Compiler Backend

Christian Chavez

Tuesday 16th June, 2015

Abstract

Inlining is one of many compiler optimization techniques used in the process of optimizing programming code. The art of inlining sufficiently and correctly to achieve optimized executables is a delicate one. One must also extra take care when inlining recursive functions, lest that the compilers ends up in a non-terminating inlining cycle.

We implement an inliner in the new compiler backend Jive, and test it with a subset of the files in the SPEC2006 Benchmark Suite. The implemented inliner is also designed to be order to the performed inlines, following a strict top-down or bottom-up approach of nested call stacks.

The focus of our implementation is to handle as wide a variety of applications from the SPEC2006 Benchmark Suite as possible, and checked specifically for the change in count of operations in the program after the inliner had executed.

We also comment on the benefits of inlining in Jive, since it is able to convert Clang IR into a *demand-based* and *directed acyclic graph*, namely a Regionalized Value State Dependence Graph.

Our results show that both our focus on ensuring that the call sites are inlined in a top-down or bottom-up fashion trough the nested call stack is potentially generalizable. We managed to achieve a 6.78% and 6.98% reduction in count of operations, with a top-down and bottom-up traversal respectively.

Contents

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, in order to improve the quality of the emitted code. Common techniques are *Common Subexpression Elimination* (CSE) [1, Ch. 8.5], *Dead Code Elimination* (DCE) [1, Ch. 8.5], and *inlining* [1, Ch. 12.1].

Inlining 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(). However, if both invocations are inlined, code code duplication might be mitigated through CSE. Additionally, if both of C()'s invocations are inlined, then the function body of C() may be removed through the application of DCE since C() is static and no longer invoked.

```
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 predominant approaches for handling inlining of recursive functions are:

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 exploration of the parameter space. How the RVSDG helps the design of our 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 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.

In Section 6 we summarize the existing related literature found in our background study for this project. Following, in Section 7, we conclude and discuss the advantages/disadvantages the RVSDG brings to the implementation of an inliner, as well as its impact, 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.

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, and need to match the operation.

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);
```

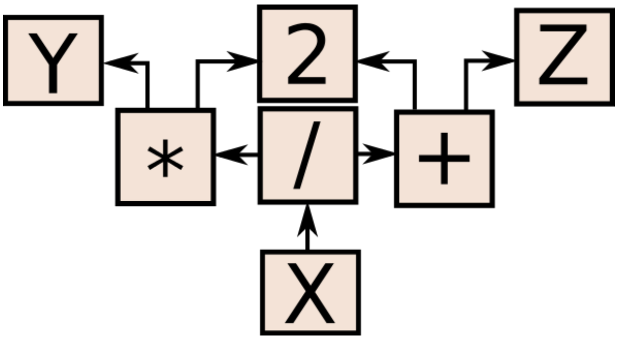

Figure 1: Example depicting an RVSDG subgraph representing the arithmetic operations and their data dependence edges corresponding to the C/C++ on the left.

2.1 Edges

The RVSDG has data- and state- 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;
```

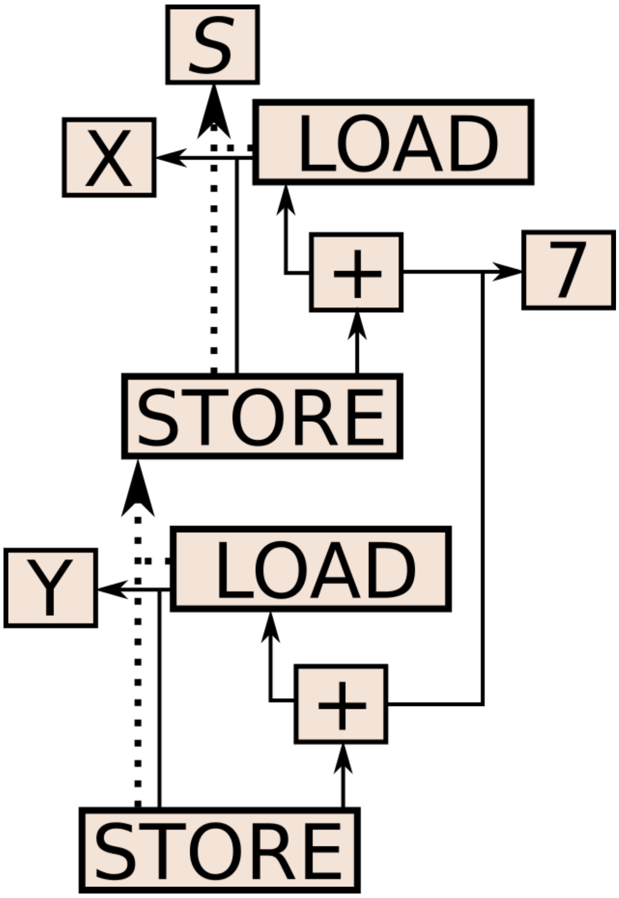

Figure 2: Figure of an RVSDG subgraph exemplifying the need for and use of state dependence edges. The RVSDG represents the equivalent of the C/C++ on the left. The *S*-node in the figure supplies the needed state edge for x's *load*- and *store*-nodes.

2.2 Nodes

The RVSDG has two kinds of nodes: simple and complex nodes.

2.2.1 Simple Nodes

Simple nodes are used to represent primitive operations, such as addition and subtraction. 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 function, 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 sub-region's internal inputs being mapped to the external outputs.

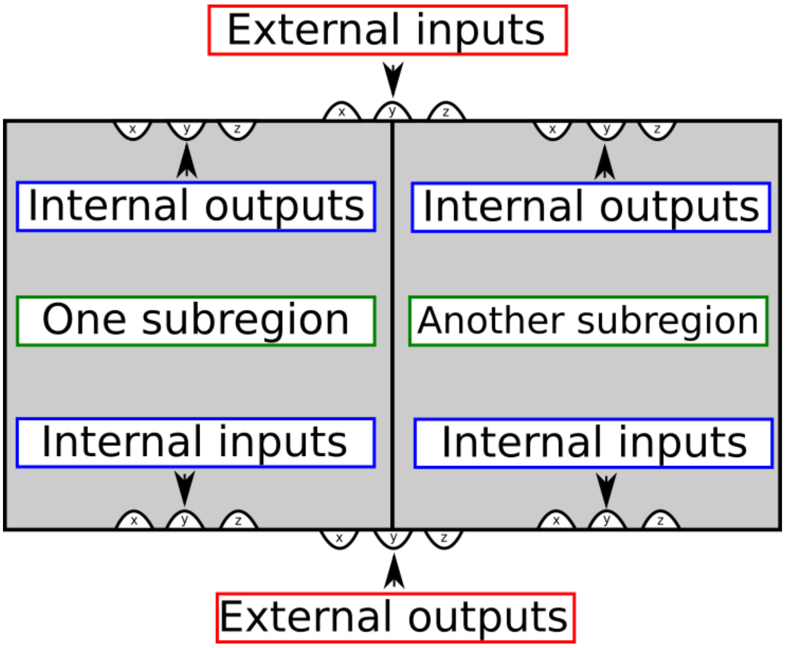

Figure 3: An example of a complex node, showing which inputs/outputs are external/internal, and how they can have multiple subregions.

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);
}
```

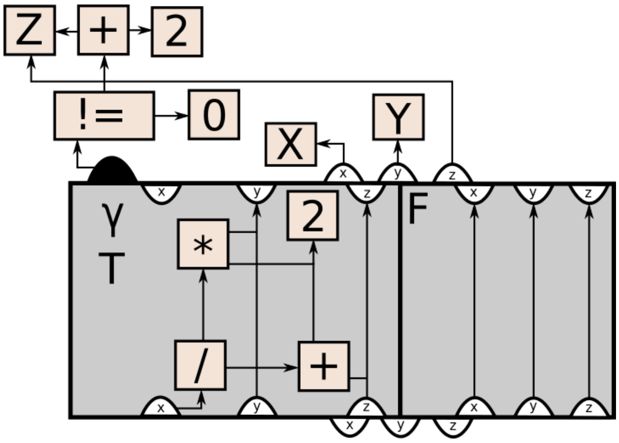

Figure 4: An example of an RVSDG subgraph containing a γ-node, representing an if-statement, corresponding to the C/C++ on the left.

- **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 in the 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 behavior 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.

```
uint32_t i = 0;
uint64_t r = 1;
do {
  i += 1;
  r *= i;
} while(i < n);
```

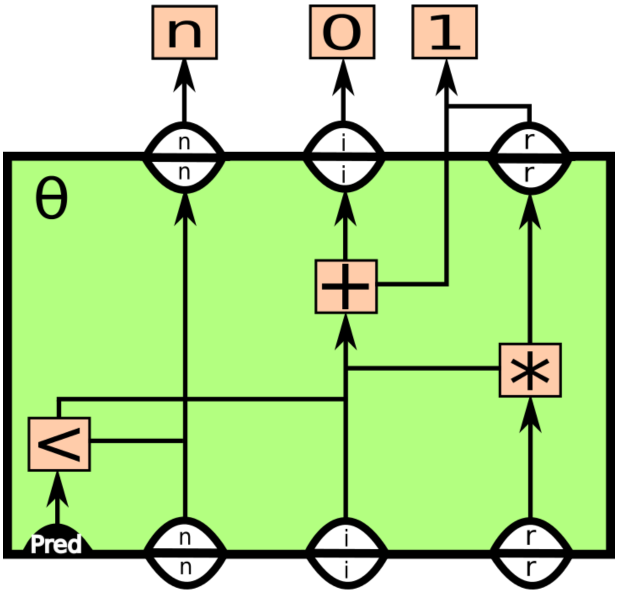

Figure 5: An RVSDG subgraph depicted on the right, on the right, containing a θ-node representing the C/C++ do-while loop on the left.

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.

```
uint64_t
   fac(uint32_t n){
  uint32_t i = 0;
  uint64_t r = 1;
  do{
    i += 1;
    r *= i;
  } while(i < n);
  return r;
}
```

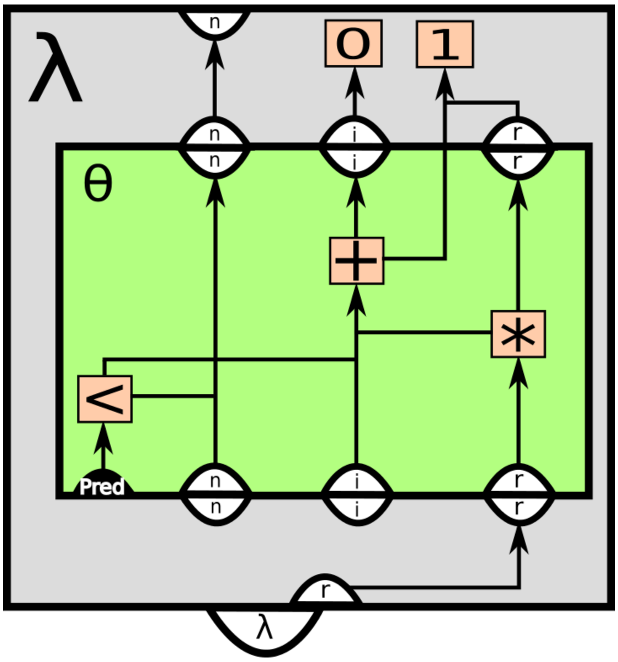

Figure 6: Example of an RVSDG subgraph depicted on the right, containing a λ-node representing the C/C++ function on the left.

- **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.

```
uint64_t
   fac(uint32_t n){
  uint64_t r = 1;
  if(n > 1){
    r = n*fac(n-1);
  }
  return r;
}
```

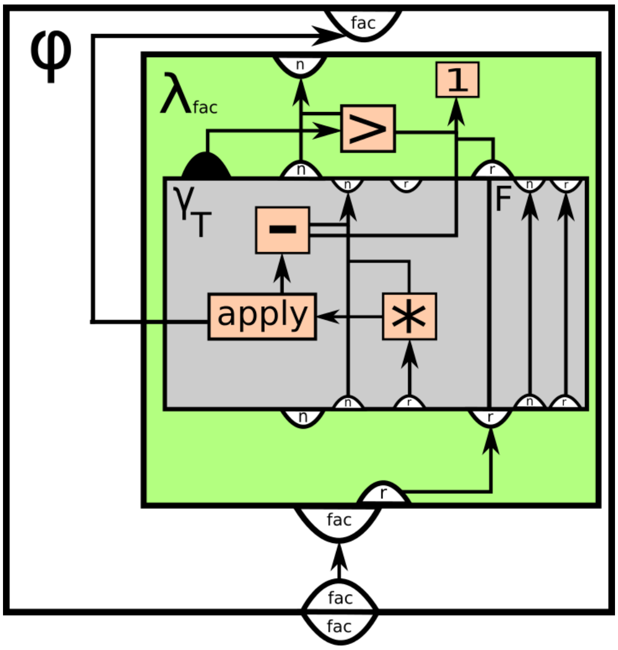

Figure 7: Example of an RVSDG subgraph depicted on the right, containing a ϕ-region containing a representation of the recursive C/C++ function on the left.

3 The Inliner

Our inliner utilizes Jive's RVSDG IR functionality to perform the inlining on the inputted assembly code program. As such, the inliner is both compiled into Jive, and utilizes functionalities offered by Jive. This section details the actions performed by the inliner, and the reasoning behind these. First, we give an overview of the actions performed by the inliner, before we go more into depth on the reasoning behind them.

Following is the algorithm representing the steps executed by the inliner:

1. For all recursive environments (ϕ-regions):

   (a) Use the approach described in Section 3.1 to fill a list of *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 formed by the *apply*-nodes represents a Strongly Connected Component (SCC).

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

1. Mark the λ-node as "visited".
2. Collect all the *apply*-nodes contained within this λ-node which *do not* invoke a function from the list of *loop breakers* (initially empty).
3. If one of the collected *apply*-nodes 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 made by the call graph inside a ϕ-region, that all λ-nodes are visited. It is also guaranteed that for every cycle in the SCC, there is at least *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], implementing one in our inliner is outside the scope of this project.

3.2 The order of call sites inlined

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

Given that the heuristic deciding whether an *apply*-node is inlined or not depends upon whether the *internal node count* of the function it invokes is less than four, the

[1] In the attempt of minimizing the amount of loop breakers in a ϕ-region, as mentioned in Section 8.2.

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

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.

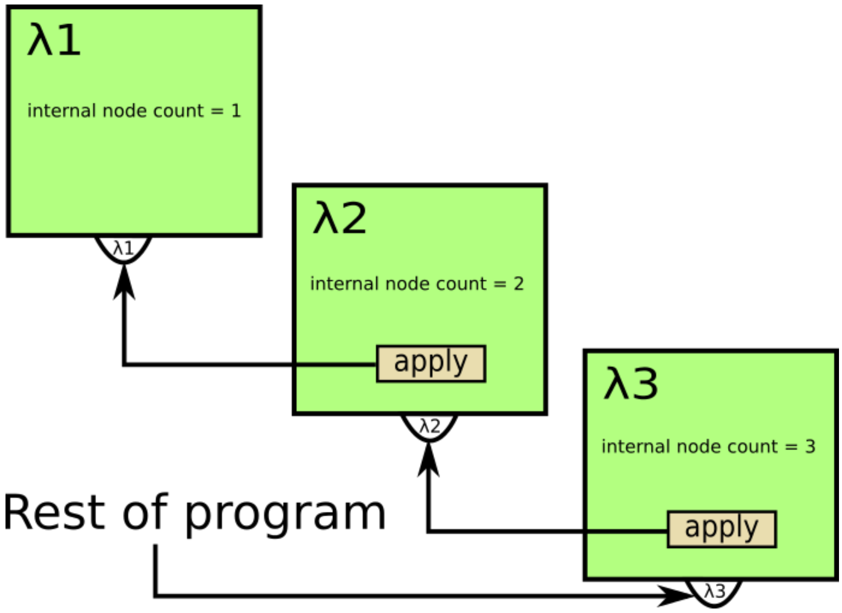

Figure 8: An example of an RVSDG subgraph, depicting a function call order in a program.

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

As our example shows, the order in which the *apply*-nodes are inlined matters. And because the order matters, 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 which inlining is performed, we decided upon our 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 values.

While the work of Waterman [13], which our approach builds upon, utilizes a hillclimber algorithm to adaptively find decent values for the placeholder variables for each individual run of the compiler, this is outside the scope of our project.

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 and the rationale behind CNF clauses used for testing, before explaining how the files tested from the SPEC2006 Benchmark Suite were chosen.

4.1 The Inlining Conditions

The ICs utilized in this project are as follow:

- **Node Count** (NC)

  This function property represents the number of operations (nodes) contained within a function. A function's node count is an IC we want to utilize because it gives us an indication for what the upper limit of potential code duplication might become, 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, and the function is not exported, then the call site can always be inlined since 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 can especially be the case when type conversion is required [13]. In some cases, the computational cost of an inlined function 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.

- **Exported**(EXP)

  This inlining condition tells us whether or not the function called is exported or not. If it is not, it is of greater interest to inline, because if all call sites are inlined, the declaration of the function can be removed through *Dead Code Elimination.*

4.2 The CNF

Section 3.3 explains the generic form a CNF may have, and how our inlining conditions introduced in Section 4.1 can help build such.

While a certain CNF may prove quite helpful with one program, there is no guarantee that other programs would benefit equally, if at all, from the same CNF. In fact, a CNF which proves to be the best for the execution time of one program, may increase the execution time of another.

Hence, care must be taken when selecting a CNF. Since today's compilers are expected to run in seconds or less, it is a difficult challenge finding compromises between a CNF that gives decent results for many different applications, yet is efficient enough not to slow down the compilation time too much. The potential code size increase of the compiled program cannot be forgotten either.

With these limitations in mind, and through some inspiration from GCC Waterman [13], we decided on a CNF built from the following clauses:

1. EXP == false && SCC == 1

   What this clause asks for is whether or not the function being called is not exported, and that there are no other call sites invoking this function. If so, then there is nothing to lose by inlining it. Not only is the function invocation cost removed, but there is no risk of negatively affecting the size of the compiled end result. This most often occurs after enough call sites have been inlined, so that SCC becomes 1.

2. NC < X

   This clause only asks whether or not the amount of operations inside the function being invoked are less than the placeholder value X. This is to remove all small nodes which perform simple tasks, such as getter's, setter's, increments, and so forth. The code size does not run a big risk of being negatively affected by inlining these. In fact, inlining such function can conceivably increase potential optimizations such as *Common Subexpression Elimination.*

3. CIN < Y

   With the intention of having the placeholder Y at lower values, this clause attempts to seek out the leaf nodes in nested calls. Research [13] shows that a lot of execution time are spent in these, and hence they are of interest to inline, in the hope that other optimization techniques may be applied on their operations.

4. NC < Z && CPC > V

   This clause wants to inline call sites whose constant parameters exceed the placeholder value V. The term NC < Z is there to prevent code size explosion. limiting the inlining to functions whose node count is less than Z. If the input parameters of the function are constant, then there's conceivably a bigger potential for other optimization techniques to be applied, if inlined.

5. NC < W && LND > T

   The final clause is very similar to the previous one, but instead of looking for call sites with constant parameters, it looks for call sites residing inside of loops. It is well known that most of the execution time of a program is spent inside of loops. This makes the unveiling of potential optimizations of function calls inside of loops tempting. Again, to avoid code size explosion, and to help minimize the placeholder value T, no functions with internal node count larger than W are inlined by this clause.

Thus, the final CNF has the form:

$$\text{(EXP == true \&\& SCC = 1)|| NC < Z || CIN < Y ||}$$
$$\text{(NC < Z \&\& CPC > V)|| (NC < W \&\& LND > T)}$$

See Section 5.3 for the final form with the placeholder values chosen.

4.3 The SPEC2006 Benchmarks

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

1. The Benchmark Suite program code files were written in C.
2. Clang-3.3 (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 instructions in the .ll files generated by Clang and construct RVSDGs from them.
4. The files could to be tested within a time limit of 120 seconds[2], executed single-process, on a Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz, with 3072 KiB cache size, Ubuntu 14.04 64-bit based Linux distribution.

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

Benchmarks	perlbench	bzip2	gcc	mcf	milc	gromacs	gobmk	hmmer	sjeng	libquantum	h264ref	lbm	sphinx3
# files	7	4	47	9	52	85	19	45	8	8	1	2	32

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

[2] This due to a bug in the library needed for compiling an executable using Jive.

5 Results

In this section, we first profile the Benchmarks introduced in Section 4. Then, we profile the *apply*-nodes in the Benchmark Suite specifically, before we explain the reasoning behind the placeholder values we decided upon for the CNF we used in our testing.

Thereafter we show and discuss the results of our testing. We test with both the top-down and bottom-up traversals discussed in Section 3.2. Some focus is put on the status of Jive, and how its status may have affected our test results.

All averages discussedi in this section are linear averages.

5.1 Profiling SPEC2006 and its functions

To decide upon the placeholder values for the the CNF introduced in Section 4.2, we profiled the SPEC2006 Benchmarks for the following properties among others:

First and foremost, while a higher node count may still result in faster execution in some cases, minimizing the amount of operations in a programs' RVSDG was one of our main motivations behind deciding the placeholder values of the CNF.

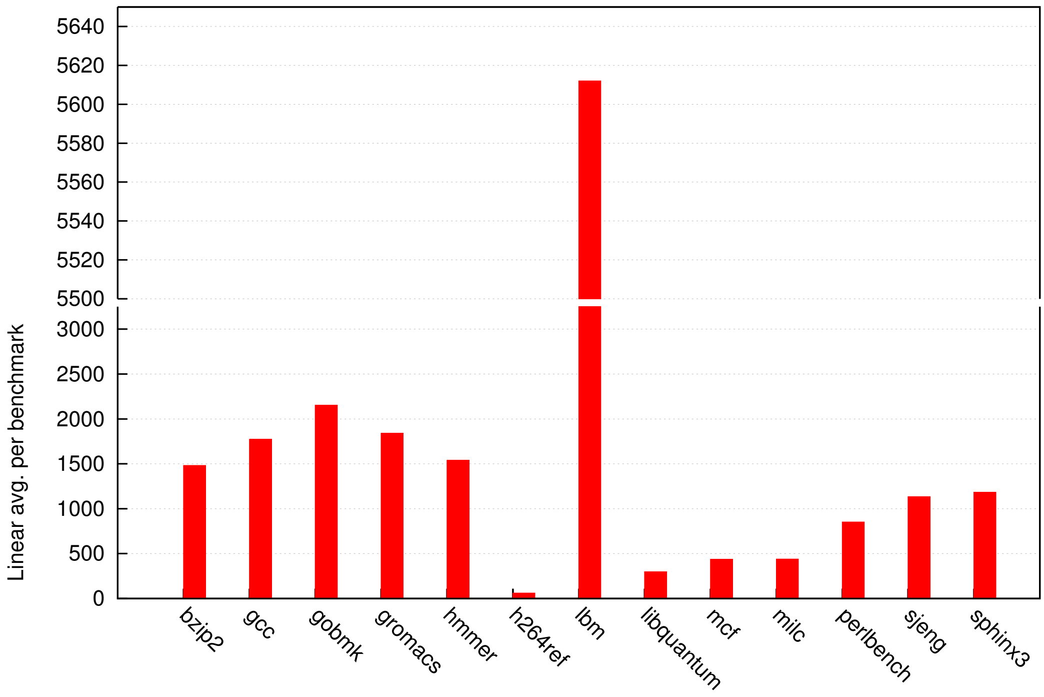

Figure 9: Histogram showing the average amount nodes present in each SPEC2006 Benchmark. The y-axis is broken up due the two benchmark's *h264ref* and *lbm*'s big difference in average node count.

Figure 9 shows the averaged node count for each of the SPEC2006 Benchmarks used. We use Figure 9 as our baseline, to which we compare the results of our testing in Sections 5.4 and 5.5.

The reason behind *lbm*'s high node count average is caused by one of *lbm*'s files which contains functions that fill/modify large IO_FILEs with content. *h264ref* also stands out, though not as much as *lbm*. It stands out due to its low average of nodes per file. It does so because it only has one file which made it past the requirements listed in Section 4.3, specifically requirement 3, and the majority of this file consists of two leaf functions.

This observation raises the obvious point that by profiling the functions in the benchmarks, we can observe how many of the functions are exported, seeing as Jive is unable to link the program files at compile time. This would give us an upper bound for how many functions we can remove through *Dead Code Elimination* (DCE). Figure 10 shows us how many functions there are on average in each Benchmark, and how big of an average of these are exported.

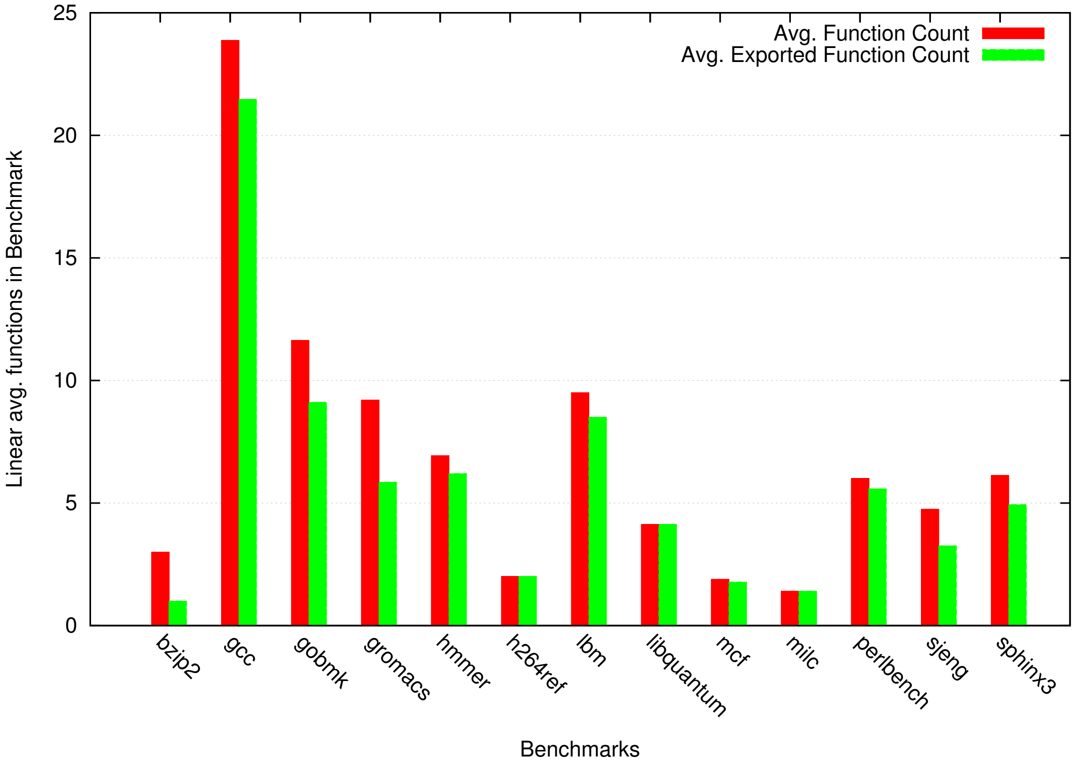

Figure 10: Histogram showing that there is a small number of functions we can hope to eliminate through DCE. The left bar shows the average count of λ-nodes per benchmark, while the right bar shows the average count of exported λ-nodes in the same benchmark.

One of the IC most used in our final CNF form is *Node Count* (NC). Hence, knowing what NC functions have on average in our test files is helpful. Figure 11 depicts this in a scatter plot.

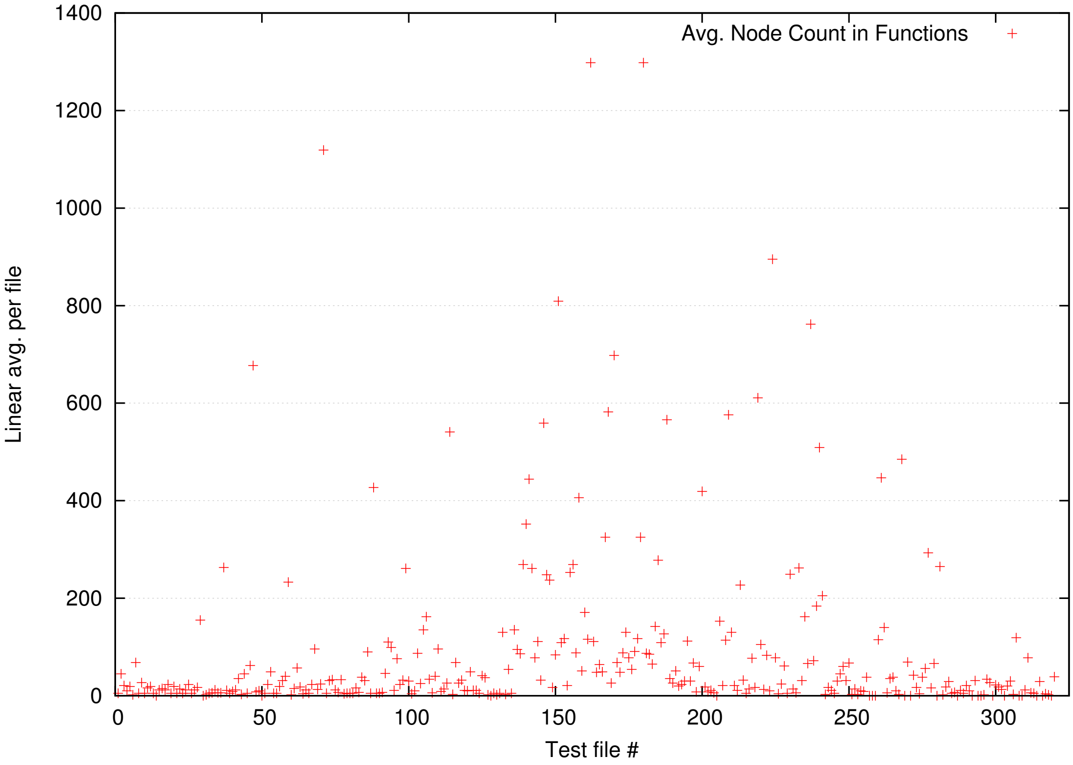

Figure 11: Scatter plot showing the spread of average amount of nodes contained within each function per file in the Benchmarks Suite. The files are grouped by Benchmark along the horizontal axis.

The spread of data points in Figure 11 tells us that while there are some functions like the aforementioned of *gcc*, the majority of them contain less than 100 nodes. Which conforms well with the stipulation that most applications majority of nodes are leaf nodes, which generally contain few operations. This makes it somewhat easier for us to decide what limits NC should present with regards to inlining, since we can see that a majority would be inlined if we inlined all functions satisfying $\text{NC} < 200$.

The histogram in Figure 12 tells us that there are no recursive environments containing more than one function in the Benchmarks tested. In other words, that all the recursive functions are self-recursive. As such, we did not get to test our algorithm detailed in Section 3.1 with the SPEC2006 Benchmark Suite, but in our own testing we did confirm that it permitted us to safely inline *some* functions inside a recursive environment containing multiple functions.

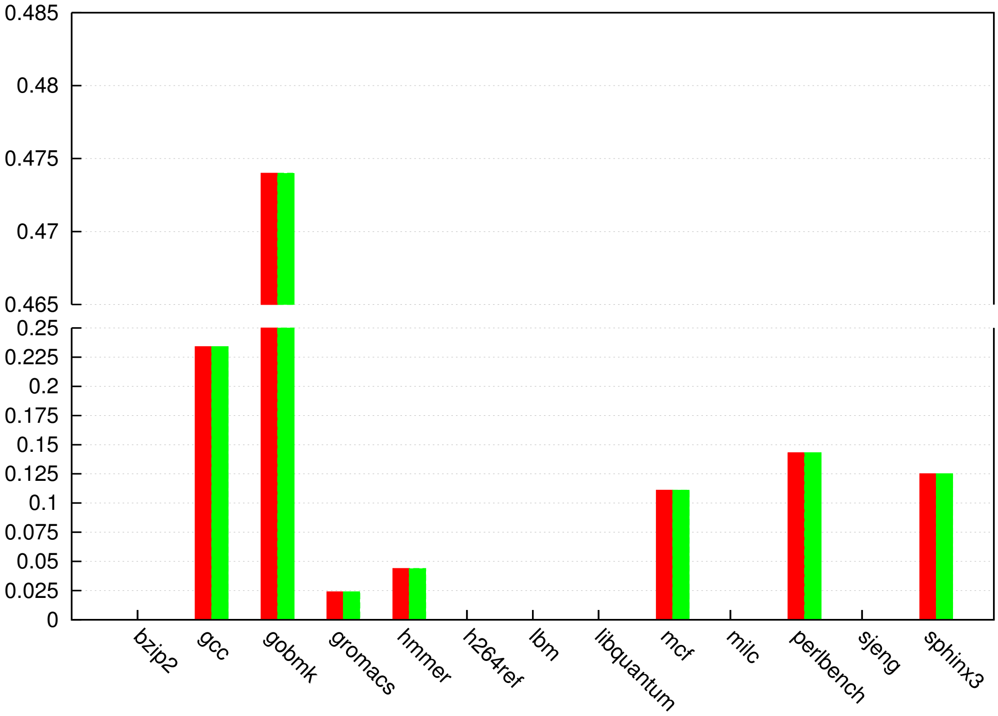

Figure 12: Histogram showing the average amount of ϕ-regions per benchmark with the left bar. The left bar shows the average amount of λ-nodes inside each ϕ-region present in a benchmark.

Another property of functions of interest to us is the average amount of call sites contained within functions. This property gives us an idea as to the proportion of how many leaf nodes there are, containing no function calls, in a benchmark, in addition to the proportion of functions with X amount of call sites on average.

Figure 13 shows us that on average the balance between functions with no call sites, and functions with, is not sharply skewed one way or the other. We do have some outliers, like the aforementioned files from *gcc*, but one can safely say from the plot that a majority of the functions have less than two hundred call sites contained within.

The histogram in Figure 13 shows that beneath the average of two hundred call sites in function mark on the y-axis, there is a slight predominance for functions to have about one hundred or less. These observations are helpful in giving us upper bounds for how many functions could potentially be inlined by this IC.

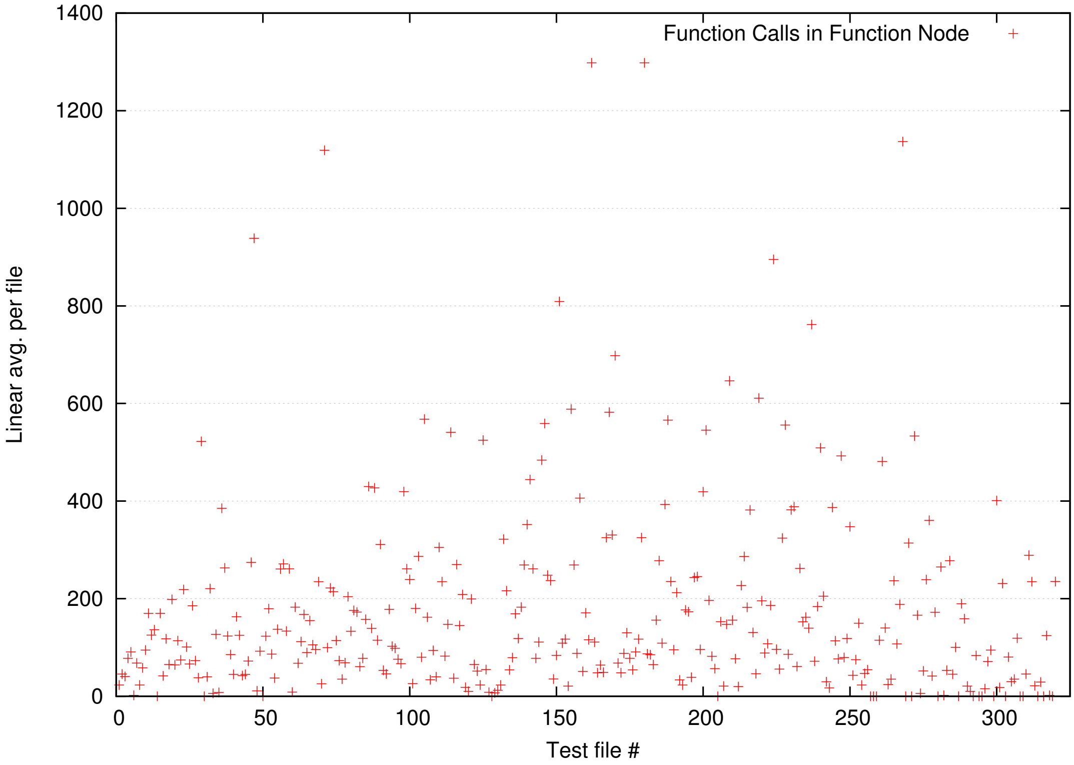

Figure 13: Scatter plot showing the spread of average amount of call sites contained in functions per file in the Benchmarks Suite. The files are grouped by Benchmark along the horizontal axis.

The final function property we profile is the amount of call sites invoking a function. As the first clause of our CNF form states, if the function is not exported, and there's only one invocation of the function, it is an easy decision to inline.

However, by graphing the spread of how many call sites each files' functions have on average, it gives us an idea of how many more functions could potentially be inlined if just some of its invocations are inlined. Figure 14 shows us a scatter plot of this data across all the test files used in the Benchmark Suite. The few functions in the bottom right corner of the plot are functions which have no call sites inside the same file, but are exported, meaning that if the benchmark's program files were statically linked during compilation, there would be no functions with zero static call sites.

What the data in Figure 14 tells us, is that the majority of functions are called just once. Seeing as Jive has not statically linked the files in the benchmark during the creation of the RVSDG, this is understandable. If Jive was able to build the RVSDG with statically linked files, our suspicion is that the range of the vertical axis, and spread data along it, would most likely grow.

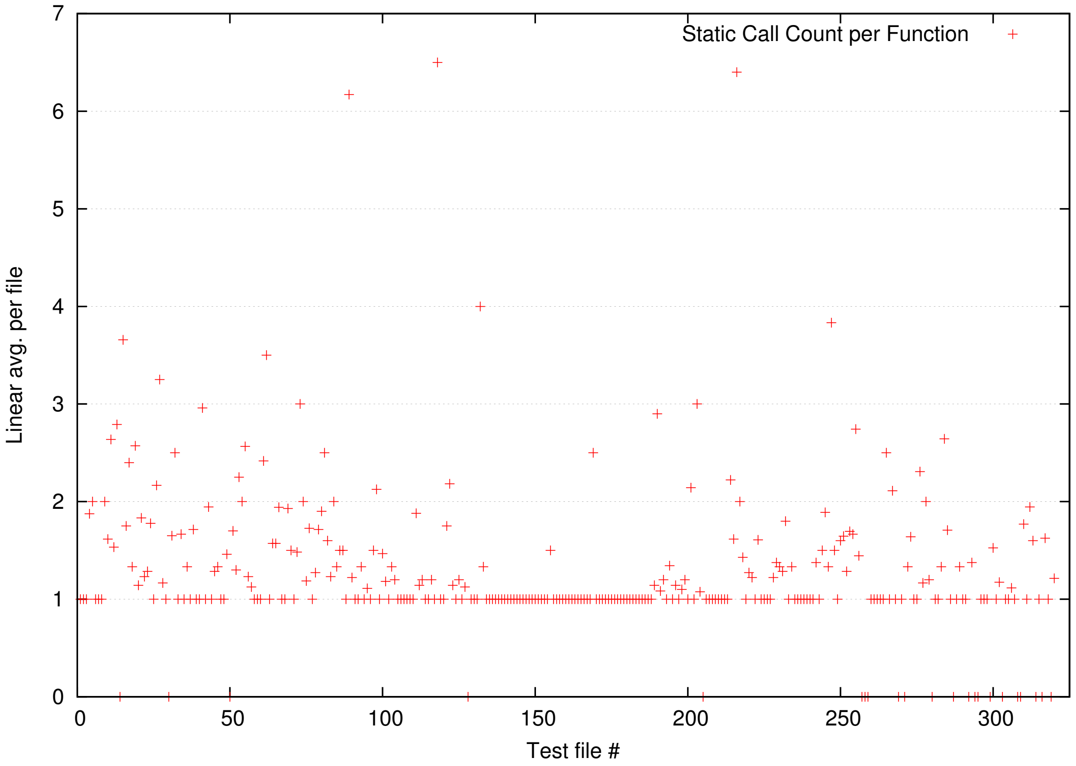

Figure 14: Scatter plot showing the spread of average Static Call Counts for functions per file in the Benchmarks Suite. The files are grouped by Benchmark along the horizontal axis.

However, as the data stands, we can see that if a function is not exported, and two of its invocations are inlined, there is a decent chance of applying DCE on the functions.

5.2 Profiling SPEC2006's *apply*-nodes

In our profiling, the data we are most interested relates to the *apply*-nodes in the benchmarks. First we check in Figure 15 how many *apply*-nodes each benchmark contains, and which of these we can potentially inline. Our definition of *statically known* apply-*nodes*, is as follows: if the λ-node invoked is directly connected to the *apply*-node in the RVSDG, or resides within a ϕ-region, which in turn is connected directly to the *apply*-node, it is statically known.

We cannot inline *apply*-nodes which are not statically known, and since Jive is unable to link files at compile time, there is conceivably large portion of *apply*-nodes not statically known. If Jive had been able to create and process RVSDGs composed of linked object files, instead of just disparate files, the proportion of statically known *apply*-nodes may have been higher. Figure 15 shows the) average count of *apply*-nodes per SPEC2006 Benchmark in the left bar, and the) average count of the statically known in the right bar.

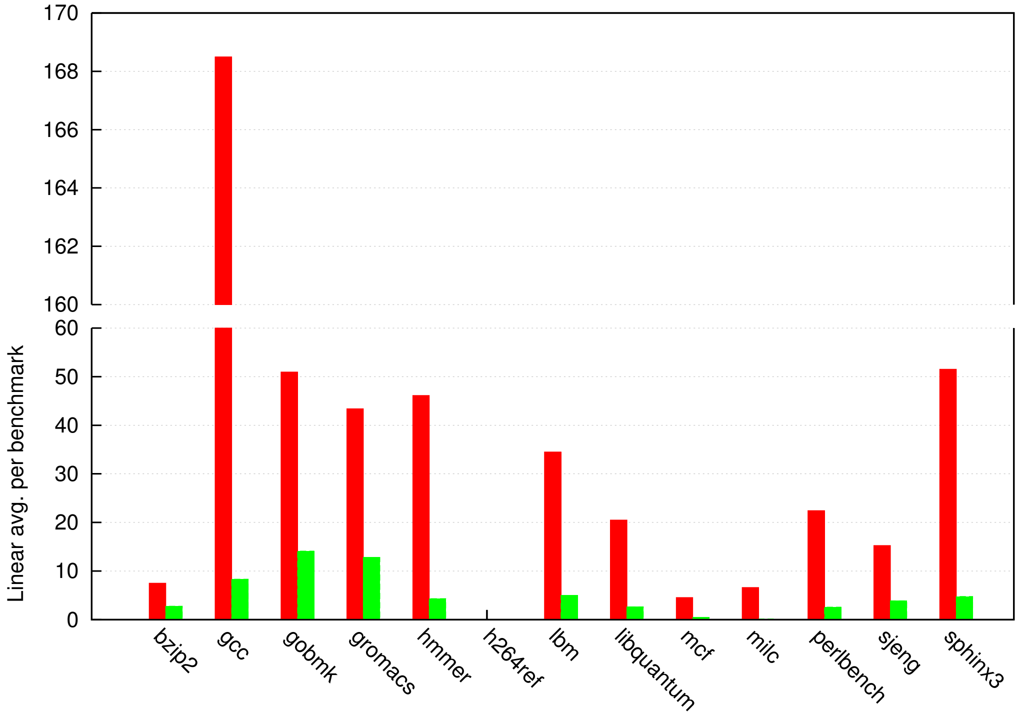

Figure 15: Histogram showing the average count of *apply*-nodes (left bar), and the average count of statically known *apply*-nodes (right bar), per benchmark.

As Figure 15 also shows, there are only two benchmarks with more than ten statically known *apply*-nodes and λ-nodes each. This time *gcc* is the one with numbers high above the rest with regards to the λ-node average, but what is shown is that *gcc*'s *apply*-node average also stands above the rest. Further investigation reveals that there is one lone file inside the Benchmark which has over six thousand *apply*-nodes and almost eight hundred λ-nodes. This particular file contains almost only functions, many of which invoke multiple others, thereby achieving these numbers.

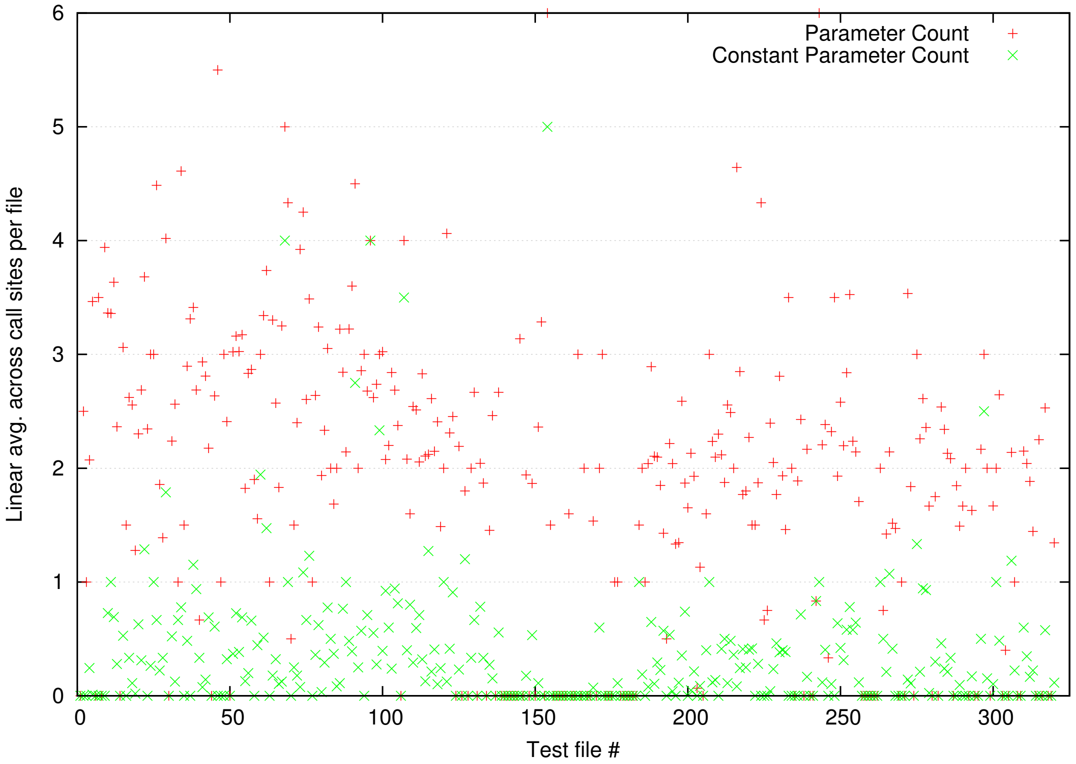

Figure 16: Scatter plot showing the spread of average Parameter Count per file in the Benchmarks (plus signs), and the spread of average Constant Parameter Count (depicted by tilted plus signs). The files are grouped by Benchmark along the horizontal axis.

The data spread in Figure 16 shows that there are few function calls with up to six parameters, some of which have up to five constant parameters in the invocation. The region of average parameter counts is most dense in the area between 1.5 and 2.5 on the vertical axis. To avoid inlining the majority of function calls, the lower limit of constant parameters needed should be above two. Thus we avoid code size explosion by hindering function invocations with only one or two constant parameters, while we permit function invocations with more to become inlined.

Finally, one of the most important properties of a call site with regards to inlining is the amount of nested loops it resides within. Figure 17 plots the average LND count of the call sites in a Benchmark Suite test file. The scatter plot in Figure 17 tells us that there are no call sites (*apply*-nodes) residing within more than two nested loops. While research shows that most of a programs' execution time is spent inside of loops, it is useful to have an idea of the proportion of call sites within one, two, or zero loops. If the CNF clause checking for LND of our CNFs final form sets the threshold at three or higher, the clause would be useless, since there are no call sites satisfying the requirement present in the test files.

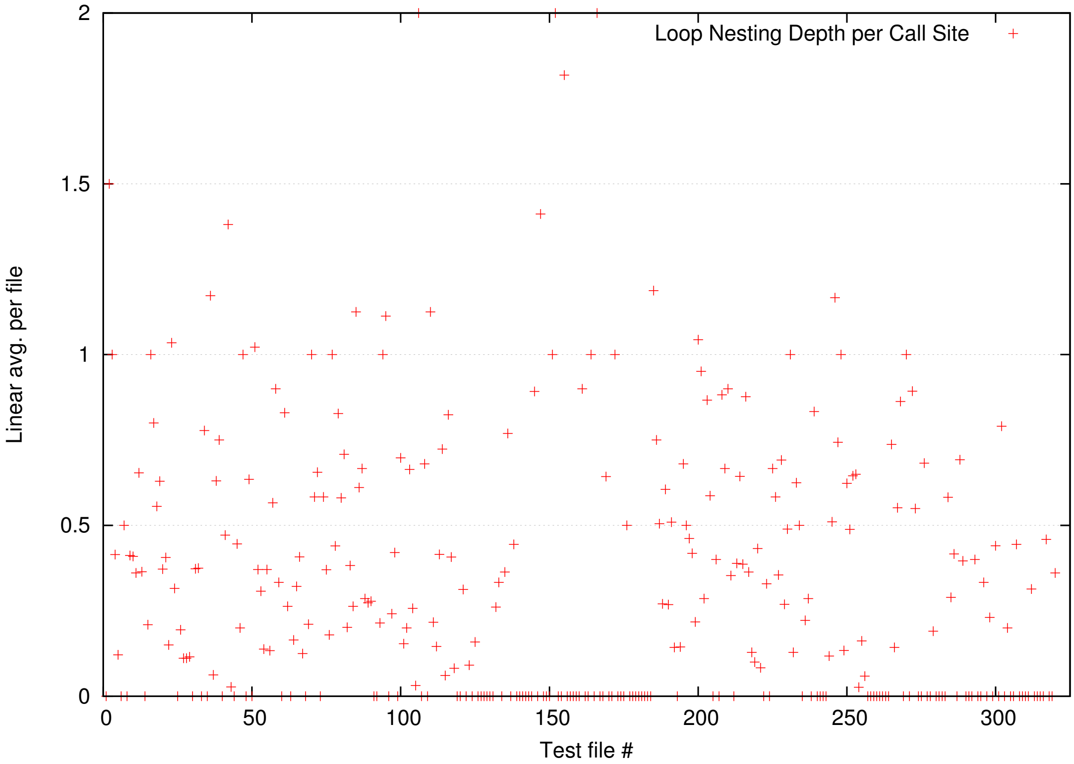

Figure 17: Scatter plot showing the spread of average Loop Nesting Depth for call sites per file in the Benchmarks Suite. The files are grouped by Benchmark along the horizontal axis.

5.3 Final CNF

With the observations made from the profiling data, and some testing to find the optimum values with regards to average node count size in the benchmarks, the following values were used for our test results:

$$\text{(EXP == false \&\& SCC = 1)|| NC < 25 || CIN < 3 ||}$$
$$\text{(NC < 200 \&\& CPC > 2)|| (NC < 200 \&\& LND > 0)}$$

5.4 Inlining results: top-down traversal

Since Jive is not able to produce executable code, we are unable to test the timing of the RVSDGs our inliner has been executed on. Hence, as mentioned in Section 5.1, our motivation has been to reduce the amount of operations. This because we interpret the NC to be generally representative of the time it would have taken to execute the optimized code.

Figure 18 shows a histogram where the averaged NC per benchmark is represented before and after the inliner has been executed. While we repeat that these results must be taken with the proverbial grain of salt due to Jive's inability to statically link the files of the benchmarks, we do hope that they are somewhat representative of how the effects of our inliner.

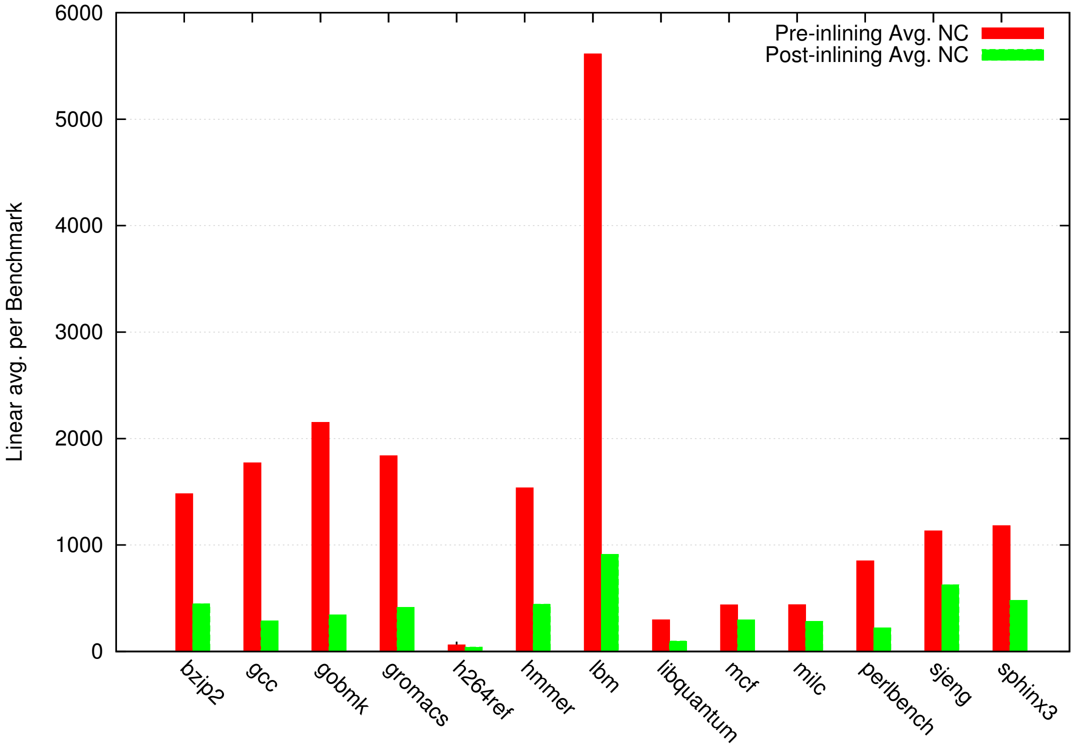

Figure 18: Histogram showing the average NC per SPEC2006 Benchmark before (left bar), and after (right bar) inlining.

Figure 19 shows the spread of how much time the inliner needs for processing the files in from the Benchmark Suite. Again, the results are somewhat skewed due to Jive's status. Jive does not have all of the common compiler techniques implemented either. Optimization techniques such as auto vectorization and alias analysis are lacking in their entirety, and what optimizations it does have are not completely bug-free.

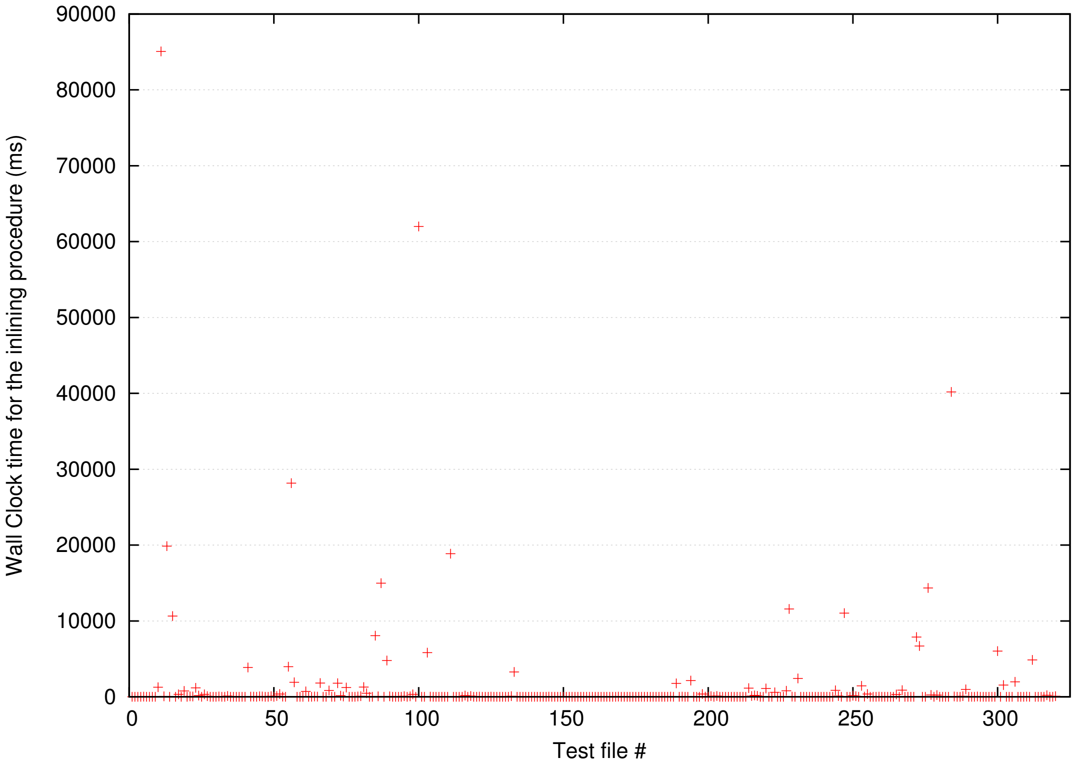

Figure 19: Scatterplot showing the spread of the time it takes in (wall clock ms.) to execute the inliner per .c file tested in a top-down traversal order.

However, with the exception of the few outliers, we see that the majority of test files were executed in under one second. As to the outliers needing up towards 80 seconds to execute, due to Jive's status as under development, it is difficult to determine for sure whether the majority of the time is spent in the traversal of call sites, or in the necessary functions offered by Jive[3].

Table 1 summarizes the NC results per SPEC2006 Benchmark pre- and post-inlining, giving a bit more detail and summarizing the effects across all the files in the Benchmark Suite. The weighted average of post-inlining NC divided by pre-inlining NC across all benchmarks give us a 6.79% average reduction in NC after executing our inliner.

[3] Discussed further in Section 7.1.

Table 1: Top-down post- and pre-inlining Node Count averages per SPEC2006 Benchmark

SPEC2006 Benchmarks	**Average Node Count Pre-Inlining**	**Average Node Count Post-Inlining**	**% Difference in Node Count**	**# Files per Benchmark**	**% of files total in Benchmark**
bzip2	1481.250	494.750	33.401%	4	1.254
gcc	1771.660	1368.319	77.234%	47	14.734
gobmk	2152.158	2001.421	92.996%	19	5.956
gromacs	1838.741	1700.941	92.506%	85	26.646
h264ref	61.000	61.000	100.000%	1	0.313
hmmer	1537.667	1482.222	96.394%	45	14.107
lbm	5612.000	5627.500	100.276%	2	0.627
libquantum	297.000	321.750	108.333%	8	2.508
mcf	437.222	438.667	100.330%	9	2.821
milc	439.231	441.019	100.407%	52	16.301
perlbench	850.857	872.571	102.552%	7	2.194
sjeng	1133.125	1058.375	93.403%	8	2.508
sphinx3	1182.406	1199.875	101.477%	32	10.031

5.5 Inlining results: bottom-up traversal

The testing for the bottom-up ordering was performed identically as the top-down orderings tests. Thus we will display the same table and figures, but with new data and our guesses as to the difference between the two. The one notable exception between the two being that one of the test files in the *hmmer* benchmark was unable to complete the execution within the allotted 120 seconds.

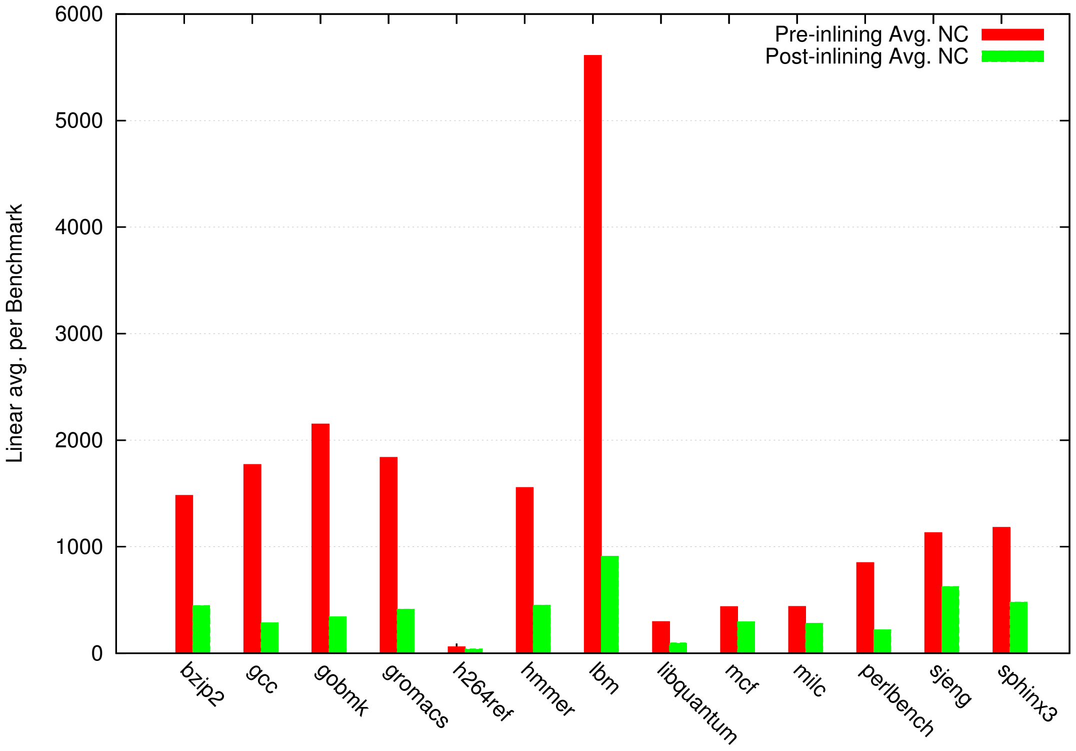

Figure 20: Histogram showing the average NC per SPEC2006 Benchmark before (left bar), and after (right bar) inlining.

Figure 20 shows a very similar trend to Figure 18. So from NC alone there does not seem to be much difference between the averaged NCs per benchmark.

The spread in Figure 21 shows that while the spread along the vertical axis does not go as high in the bottom-up traversal as for the top-down traversal, there are more test files requiring more than half a second to execute with the inliner. While the top-down ordering takes an average 1279.213 ms. to execute, the bottom up ordering takes 1576.028 ms.

Table 2 summarizes the same results from the SPEC2006 Benchmark Suite as Table 1, though with bottom-up traversal instead. However, what is worth noticing, is that the weighted average of post-inlining NC divided by pre-inlining NC across all benchmarks gives a 6.98% average reduction in NC after executing our inliner.

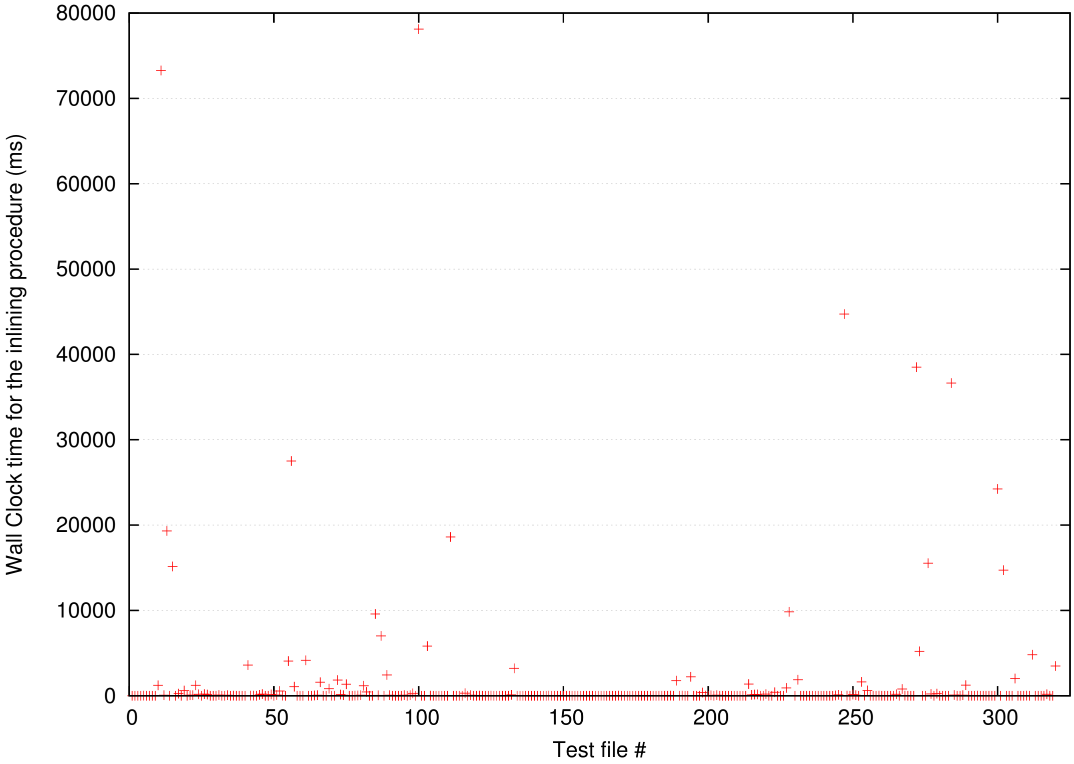

Figure 21: Scatterplot showing the spread of the time it takes in (wall clock ms.) to execute the inliner per .c file tested in a bottom-up traversal order.

The discrepancy between the top-down and bottom-up traversals averages of NC and execution time may be explained by the following: Top-down traversal inlines leaf nodes first into their callees, while bottom-up inlines the first functions in nested functions calls first. While some research suggests that the majority of execution is sometimes spent in leaf nodes, the call sites in side of nested loops, which we also seek to optimize, will be the ones most likely to be inlined first with the bottom-up ordering. This could account for the increased reduction of operations, since as more operations are collected closer to the root function[4], the chances for applying DCE increases.

The timing can also be accounted for, since leaf nodes tend to have a lesser NC, the cost for Jive to inline a leaf node is very likely smaller than inlining a function from the the top of the nested call stack into its predecessor. Thus, for optimal results, a balance could perhaps be found between the two.

[4] Comparable to C/C++'s main().

Table 2: Bottom-up post- and pre-inlining Node Count averages per SPEC2006 Benchmark

SPEC2006 Benchmarks	***Average Node Count Pre-Inlining***	***Average Node Count Post-Inlining***	**% Difference in Node Count**	**# Files per Benchmark**	**% of files total in Benchmark**
bzip2	1481.250	494.750	33.401%	4	1.258
gcc	1771.660	1351.766	76.299%	47	14.780
gobmk	2152.158	1974.842	91.761%	19	5.975
gromacs	1838.741	1690.400	91.932%	85	26.730
h264ref	61.000	61.000	100.000%	1	0.314
hmmer	1555.455	1523.886	97.970%	44	13.836
lbm	5612.000	5627.500	100.276%	2	0.629
libquantum	297.000	325.375	109.554%	8	2.516
mcf	437.222	438.667	100.330%	9	2.830
milc	439.231	440.635	100.320%	52	16.352
perlbench	850.857	872.571	102.552%	7	2.201
sjeng	1133.125	1077.875	95.124%	8	2.516
sphinx3	1182.406	1188.656	100.529%	32	10.063

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 Ph.D. 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

In this section we first re-iterate our results from Section 5, and reflect on the implications Jive's status has on our results. Thereafter, we discuss the what we consider to be the more crucial points of our inliner, before we finish with acknowledging all who helped with this project.

7.1 Summary & reflection of results

The implemented inliner's effect on the linear average of operations in the test files from the SPEC2006 Benchmark Suite is disputable, due to the fact that Jive is still under development. This brings the risk that the compilations are not bug-free, and the challenge that it is not able to statically link files at compile time. Nor is Jive able at this point in time to produce executable code we could test.

Yet in spite of these drawbacks, Jive shows promise, enabling our inliner to reduce the linear average of operations across the Benchmark Suite with 6.78% and 6.98%, when traversing the call sites in a top-down or bottom up order respectively. If improvements like the ones discussed in Section 8 could be implemented, we have little doubt that the strengths exhibited by our inliner could be further improved and utilized to beneficial effect.

Our scientific contribution in this project is the notion of traversing the call sites in a top-down or bottom-up order in the RVSDG of the program. While Section 8.3 expands more on our thoughts on this, we do want to point out that our implemented traverser described in Section 3 has an algorithmic complexity of $O(N^2 LogN)$. Effort was not put into optimizing the traversal algorithm, since that was not a part of the project description. Hence, we speculate that while the traversal ordering could be worthwhile, some effort should be put into optimizing the traversal of the algorithm.

7.2 Acknowledgments

First of all, I would like to thank Nico Reißmann for his tutelage, guidance, and last but not least, patience in his role as my supervisor for this project. While the experience has taught me a great deal of things, his contribution of efforts and his own time to both the project and my supervision cannot be understated.

While this project is not on par with a Master Thesis, it is the most challenging one I've had to date. Thus, I'd also like to thank Torje Digernes, Bjørn Åge Tungesvik, Einar Johan Sømåen, Jørgen Kvalsvik, for their tips, debugging help, support, and

tutorials teaching me to utilize the tools needed for this project, some of which I had little prior experience with.

Finally I'd also like to thank Dag Frode Solberg, and all the others who helped me by reading through my reporting, and/or just giving me their time and effort in improving my project and report.

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, discuss potential future work already described in literature [6, 13, 8], as well as quickly their applications to this project and/or Jive. Thereafter we finish with a more original idea for future work in Section 8.3.

8.1 Parametrization and Adaptive Compilation

As mentioned in Section 3.3, Waterman [13] implemented a hillclimber algorithm to adaptively re-compile a program for finding the most decent inlining heuristic for each compilation.

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 is done adaptively like Waterman proposes.

The work of Cooper, et al. [6] shows how the parametrization of Waterman's [13] inliner 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 with regards to inlining, but was neither part of this project's scope, nor did we have the resources to implement it.

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 den Heijer [8] proposes, should be utilized because it makes an effort in choosing the minimum amount of loop breakers needed. This enables a larger amount of the functions residing in recursive environments to potentially be inlined.

Again, we believe the addition of using den Heijer's approach based on the Directed Blackout Feedback Vertex Set problem would have been beneficial to implement in Jive's inliner.

8.3 The order of inlining

To our knowledge, no other literature exist on the subject of the consequences of the ordering of inlined functions. While this could be because of the extra time it takes to execute our $O(N^2 LogN)$ traversal, we were unable to find anything else related to this notion.

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. Section 5.5 also shows that there changing the ordering has consequences for both the time it takes to execute the inliner, as well as the total amount of operations in the compiled program.

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 other inherent properties that can be discovered, when ordering the call sites in a top-down ordering as opposed to a bottom-up ordering, beyond the ones discussed in Section 5.5.

Another approach could be to find the aggregate cost of inlining nested function calls: profiling them first, before inlining them. This could perhaps become useful, if not even more effective, when the inliner has a large stack of nested function calls. If the sum of inlining all the function calls in the chain exceeds the limits set by the inliner heuristic, could the profiling help find out which parts of the chain are most advantageous to inline first?

Finally, while we have not been able to come up with a workable algorithm that would enable this, the reasons we propose behind the different consequences of ordering in Section 5.5 may perhaps point to another ordering entirely: alternating between the bottom and top of the nested call stack.

While these notions are interesting, they are outside the scope of our project.

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.c
  - locale.c
  - pad.c
  - perlapi.c
  - specrand.c
  - stdio.c
  - taint.c
- 401.bzip
  - blocksort.c
  - crctable.c
  - huffman.c
  - randtable.c
- 403.gcc
  - alloca.c
  - asprintf.c
  - c-convert.c
  - c-errors.c
  - cfg.c
  - cfganal.c
  - cfgbuild.c
  - cfgcleanup.c
  - cfglayout.c
  - cfgrtl.c
  - convert.c
  - cppdefault.c
  - cpperror.c
  - cpphash.c
  - dependence.c
  - doloop.c
  - dominance.c
  - dwarfout.c
  - fibheap.c
  - genrtl.c
  - getpwd.c
  - hex.c
  - hooks.c
  - insn-emit.c
  - insn-peep.c
  - intl.c
  - langhooks.c
  - lbasename.c
  - line-map.c
  - main.c
  - mbchar.c
  - mkdeps.c
  - obstack.c
  - partition.c
  - predict.c
  - print-rtl.c
  - reorg.c
  - rtl-error.c
  - safe-ctype.c
  - sdbout.c
  - sibcall.c
  - varray.c
  - vasprintf.c
  - version.c
  - vmsdbgout.c
  - xcoffout.c
  - xmalloc.c
  - xstrerror.c
- 429.mcf
  - mcfutil.c
  - output.c
  - pbeampp.c
  - pbla.c
  - pflowup.c
  - psimplex.c
  - pstart.c
  - readmin.c
  - treeup.c
- 433.milc
  - addmat.c
  - addvec.c
  - byterevn.c
  - check_unitarity.c
  - clear_mat.c
  - clearvec.c
  - d_plaq4.c
  - dslash_fn2.c
  - gauge_info.c
  - gaussrand.c
  - grsource_imp.c

  - io_nonansi.c
  - l_su2_hit_n.c
  - layout_hyper.c
  - m_amat_hwvec.c
  - m_amatvec.c
  - m_amv_4dir.c
  - m_amv_4vec.c
  - m_mat_an.c
  - m_mat_hwvec.c
  - m_mat_na.c
  - m_mat_nn.c
  - m_matvec.c
  - m_mv_s_4dir.c
  - make_ahmat.c
  - make_lattice.c
  - mat_invert.c
  - msq_su3vec.c
  - path_product.c
  - r_su2_hit_a.c
  - rand_ahmat.c
  - ranmom.c
  - ranstuff.c
  - realtr.c
  - rephase.c
  - reunitarize2.c
  - s_m_a_mat.c
  - s_m_a_vec.c
  - s_m_mat.c
  - s_m_s_mat.c
  - s_m_vec.c
  - su3_adjoint.c
  - su3_proj.c
  - su3_rdot.c
  - su3mat_copy.c
  - sub4vecs.c
  - submat.c
  - subvec.c
  - uncmp_ahmat.c
  - update.c
  - update_h.c
  - update_u.c

- 435.gromacs
  - 3dview.c
  - atomprop.c
  - binio.c
  - block_tx.c
  - buffer.c
  - calch.c
  - calcmu.c
  - calcvir.c
  - clincs.c
  - comlib.c
  - coupling.c
  - csettle.c
  - disre.c
  - do_fit.c
  - dummies.c
  - ebin.c
  - edsam.c
  - ewald_util.c
  - f77_wrappers.c
  - fatal.c
  - ffscanf.c
  - filenm.c
  - futil.c
  - gbutil.c
  - ghat.c
  - glaasje.c
  - ifunc.c
  - index.c
  - init_sh.c
  - innerc.c
  - invblock.c
  - ionize.c
  - macros.c
  - maths.c
  - mdatom.c
  - mdebin.c
  - mdrun.c
  - memdump.c
  - mshift.c
  - mvdata.c
  - mvxvf.c
  - names.c
  - network.c
  - nrjac.c
  - nsb.c
  - nsgrid.c
  - pbc.c
  - pdbio.c
  - pme.c
  - pppm.c
  - princ.c
  - psgather.c
  - pssolve.c

- psspread.c
- pullinit.c
- pullio.c
- pullutil.c
- rando.c
- random.c
- rbin.c
- rdgroup.c
- replace.c
- rmpbc.c
- shakef.c
- shift_util.c
- smalloc.c
- splittop.c
- stat.c
- strdb.c
- string2.c
- symtab.c
- synclib.c
- tables.c
- tgroup.c
- txtdump.c
- typedefs.c
- vcm.c
- vec.c
- viewit.c
- wgms.c
- wnblist.c
- xdrd.c
- xtcio.c
- xutils.c
- xvgr.c

- 445.gobmk
  - engine/filllib.c
  - engine/hash.c
  - engine/interface.c
  - engine/movelist.c
  - engine/semeai.c
  - engine/sgfdecide.c
  - engine/sgffile.c
  - engine/showbord.c
  - engine/utils.c
  - patterns/fuseki13.c
  - patterns/fuseki19.c
  - patterns/fuseki9.c
  - patterns/transform.c
  - sgf/sgfnode.c
  - sgf/sgftree.c
  - sgf/sgf_utils.c
  - utils/getopt.c
  - utils/getopt1.c
  - utils/random.c
- 456.hmmer
  - a2m.c
  - aligneval.c
  - alignio.c
  - alphabet.c
  - clustal.c
  - cluster.c
  - dayhoff.c
  - display.c
  - emit.c
  - emulation.c
  - eps.c
  - fast_algorithms.c
  - file.c
  - getopt.c
  - gki.c
  - hmmcalibrate.c
  - hmmsearch.c
  - iupac.c
  - masks.c
  - mathsupport.c
  - misc.c
  - modelmakers.c
  - msa.c
  - msf.c
  - phylip.c
  - plan7.c
  - plan9.c
  - postprob.c
  - prior.c
  - revcomp.c
  - rk.c
  - seqencode.c
  - shuffle.c
  - sqerror.c
  - squidcore.c
  - sre_ctype.c
  - sre_math.c
  - sre_random.c
  - stack.c
  - stockholm.c
  - trace.c

  - translate.c
  - types.c
  - vectorops.c
  - weight.c
- 458.cjeng
  - attacks.c
  - draw.c
  - ecache.c
  - eval.c
  - leval.c
  - rcfile.c
  - seval.c
  - ttable.c
- 462.libquantum
  - classic.c
  - expn.c
  - oaddn.c
  - omuln.c
  - qec.c
  - qft.c
  - specrand.c
  - version.c
- 464.h264ref
  - specrand.c
- 470.lbm
  - main.c
  - lbm.c
- 482.cphinx3
  - approx_cont_mgau.c
  - agc.c
  - ascr.c
  - beam.c
  - bio.c
  - case.c
  - ckd_alloc.c
  - cmn.c
  - cmn_prior.c
  - dict2pid.c
  - err.c
  - fillpen.c
  - hash.c
  - heap.c
  - hmm.c
  - io.c
  - kb.c
  - kbcore.c
  - lextree.c
  - lm.c
  - lmclass.c
  - logs3.c
  - parse_args_file.c
  - profile.c
  - spec_main_live_pretend.c
  - specrand.c
  - str2words.c
  - subvq.c
  - tmat.c
  - unlimit.c
  - utt.c
  - wid.c