Mutual Inlining: An Inlining Algorithm to Reduce the executable Size

Yosi Ben-Asher
Western Digital Tefen and
The University of Haifa CS
yosi.Ben-Asher@wdc.com

Nidal Faour
Western Digital Tefen
nidal.faour@wdc.com

Ofer Shinaar
Western Digital Tefen
ofer.shinaar@wdc.com

Abstract

We consider the problem of selecting an optimized subset of inlinings (replacing a call to a function by its body) that minimize the resulting code size. Frequently, in embedded systems, the program's executable file size must fit into a small size memory. In such cases, the compiler should generate as small as possible executables. In particular, we seek to improve the code size obtained by the LLVM inliner executed with the -Oz option. One important aspect is whether or not this problem requires a global solution that considers the full span of the call graph or a local solution (as is the case with the LLVM inliner) that decides whether to apply inlining to each call separately based on the expected code-size improvement. We have implemented a global type of inlining algorithm called Mutual Inlining that selects the next call-site ($f()callsg()$ to be inline based on its global properties. The first property is the number of calls to $g()$. Next property is determining if inlining $g()$ to $f()$ may prevent inlining other more beneficial neighboring call-sites. Finaly repeated inlining iterations over the call graph are performed until there are no more beneficial inlinings to perform. Hence, considering the effect of previously made inlinings on the next call-site to be inline. Our results show small but consistant improvement compare to LLVM's Oz.

1 Introduction

Frequently, in embedded systems, the program's executable file size must fit into a small size memory. In such cases, the compiler should generate as small as possible executables. There are other reasons why a smaller executable is desired, including: 1) It may run faster; 2) It can save power due to reduced number of I-cache-misses and DRAM size; 3) It can free the RAM needed for the dynamic parts of the program and other parts of the application, such as the real-time operating system. Several techniques can reduce the executable size, in particular using Overlays [1] wherein the program dynamically loads different parts of the executable, however here we consider using selective inlining (a compiler optimization) as a way to reduce the resulting executable.

Inlining [2] is a well-known compiler optimization that replaces a call statement (call-site) with the body of the called function. For example in the following code the function $g()$ (referred to as the "caller") contains two call statements to $f()$ (the "callee").

```
int A[1000],x2;
{
 while(A[t] > z) { A[t]+=z; t--; }
```

David C. Wyld et al. (Eds): EMSA, SEA, AIFU, NLCAI, BDML, BIoT, NCOM, CLOUD, CCSEA, SIPRO - 2022
pp. 01-16, 2022. CS & IT - CSCP 2022 DOI: 10.5121/csit.2022.120601

```
 return(t);
}

int g(int x, int y)
{
 x = f(x,y)+y+22;
 x += f(x,y);
 return(A[x]);
}
```

When inlining $f()$ into $g()$ the compiler may decide to replace the three calls to $f()$ by the body of $f()$ obtaining the following program.

```
int g(int x, int y)
{ int ty;
 ty=y;
 while(A[ty] > x) { A[ty]+=x; ty--; }
 x = ty+y+22;
 ty=y;
 while(A[ty] > x) { A[ty]+=x; ty--; }
 x +=ty;
 return(A[x]);
}
```

Note that in the two inlined bodies of $f()$ in $g()$ the variable x is directly used in the inlined body of $f()$ while since y is modified in the inlined body we need to save it in a temporary variable ty. This inlining affected the resulting size by the following factors:

- It saved the two call-instructions of $f()$ in $g()$ and the two return-instructions from $f()$ to $g()$.
- It possibly saved the instructions needed to pass the two parameters x, y from $g()$ to $f()$. The term "possibly" refers to the possibility that in the original program x, y have been passed by registers, not by the stack.
- It possibly saved the instructions used to save and restore some of $g()$'s registers according to the calling convention.
- It increased the size since the body of $f()$ is now duplicated twice instead of once in the original program.
- It increased the size due to the use of $ty = y$ compared to the original code.
- The size can also increase due to the application of optimization after the inline was applied. For example, dead-code elimination may eliminate instructions in the callee that is no longer needed due to some parameters with constant values in the call.

It thus follows that inlinings may increase the executable size mainly because the body of the callee may be duplicated in several callers. However, inlining can also reduce the executable size

as parameters passing + the calling sequence are eliminated. In addition following optimizations can reduce the size of the inlined callee even further. Inlining also affects the execution time.

We remark that inlining affects the execution time due to many other reasons (apart from the reduction in code size):

- Inlining may increase the size of the instructions-cache, creating in-cache-copies of the same function competing with each other.
- Inlining can improve the scheduling increasing the ILP.
- Inlining can allow us to apply optimizations such as CSE and invariant code motion across functions if they happen to be inlined inside the same function.

Previous works on inlining techniques considered a different problem than the one addressed here, namely that of finding the subset of inlinings that increase performance the most while not exceeding a given budget of memory size. This memory budget refers to the executable size that is obtained when no inline is applied. Typically these works present a heuristic that balances the expected increase in performance due to a given inline with the expected size increase due to that inline. The execution frequency of that call usually estimates the expected performance due inline a given call. This differs from the problem considered here, namely to select the best subset of inlinings (caller-callee pairs) that minimize the resulting executable size the most.

The proposed algorithm computes such a subset of profitable call-sites (caller+callee). Unlike the LLVM's Oz inliner that scans the call graph and separately decides whether or not to inline a given call, the proposed algorithm considers a more global type of solution wherein the effect of inlining a call on its neighboring calls is taking into account. This is because inlining one profitable call site may cause an even more profitable neighboring call site to become non-profitable. As such, the proposed algorithm selects a better subset of inlinings comparing to the LLVM's -Oz inliner.

Section 2 contains a formal graph based model for evaluating the effect of inlining. Next, section 3 specifies the proposed inlining algorithm, while section 4 compares the main features of the propozed algorithm to those of the regular -Oz inliner. Finaly, section 5 compares the executable's size of the Oz-inliner vs. the proposed MI-inliner for a selected set of C/C++ programs.

2 Formalizing the problem

Here we describe a simplified model for the inlining problem of finding a subset of call sites from a given call graph that minimizes the size of the resulting executable. The model contains the call graph wherein each edge indicate a call; it is a simplification of the real situation. Thus with every part of the model, we indicate in what way it is a simplification:

- We are given a call graph G (directed a acyclic graph) whose nodes are functions $f_1, \ldots, f_n$ and an edge $f_i \longrightarrow f_j$ indicates a call to f_i from f_j.

  The simplification part is that we assume that all the calls of the program are given as edges. This ignores the fact that some calls in a program will not be specified as edges in G as:

  - There are call-sites that call a function via a pointer (indirect calls); hence, the callee is unknown, and no edge for this call will be produced in G.

  - G is, in fact, the call graph per module as usually, applications contain multiple modules that are compiled separately. Hence, if the callee is an external function, its body may not be available, and G will not include an edge for this call.

- Each node f_i has a size $|f_i|$ indicating the size in bytes of f_i's code. If we do not apply any inline the cost (total size of the executable) is $S = \sum_{i=0}^{n} |f_i|$.

  This is a simplification as at the compilation stage in which the inlining is performed The function's code is given in its Intermediate representation (IR instruction) form [3] not in the final machine code, which will be generated after several stages of optimizations and mi-code generation. The functions' size we get are estimations of the IR instructions that will be eliminated by the following optimizations and to the number of machines instruction will be finally generated.

- Each edge $f_i \longrightarrow f_j$ of G is labeled by a size reduction e_i indicating the reduction in the size of f_j if we will apply the inline of f_i into f_j. This size corresponds to:

  - The elimination of the calling sequence (call+return instructions).
  - Elimination of the callee's instructions for saving/restoring the caller's registers, as now the callee's body becomes a natural part of the caller's code.
  - Elimination of parameter's passing in the caller, as after inlining the callee's instructions will directly access the caller's registers.

  This holds if we ignore further possible reduction in the size of $f_j + f_i$ caused by optimizations such as dead-code elimination and function specialization applied after inlining. Also, this does not account for a possible increase in the register pressure caused by adding the callee's body to the caller (possibly adding more spills to the caller).

- we assume that all the incoming edges (calls) to a function f_i has the same saving e_i. since the saving due to inlining depends on the call to f_i which is the same for every f calling f_i.

  This is a simplification since different calls to the same function may have other savings, as the saving of each call depends varies due to the number of parameters set to constants in every call. This assumption is only made for simplification in the description, but in the proposed algorithm's actual realization, we compute the estimation to each call separately.

- The sum of the e_is of outgoing edges of f_j (all the calls made by f_j) must be smaller than the size of f_j so that the result of inlining all the calls of f_j will not produce negative numbers. Thus inlining may reduce the size of f_j but never eliminate it.

Thus if we inline $f_i \longrightarrow f_j$ then

- The size of f_j becomes $|f_j| + |f_i| - e_i$.
- The edge $f_i \xrightarrow{e_i} f_j$ is eliminated.
- For every $f_k \xrightarrow{e_k} f_i$ (function called by f_i) we need to add an edge $f_k \xrightarrow{e_k} f_j$ to G.
- If the number of calls to f_i is zero it is eliminated from the graph.

Figure 1 depicts two inlining steps and there effect on the resulting size. Note that for inlining a node with *outdegree* = 1 is allays beneficial, $S = S - e3$ after inlining $f3$ into $f4$, however the second inlining $f4+f3$ into $f5$ ($f3+f4$ has *outdegree* = 2) add the cost $S = S+|f4|+|f3|-e3-e4$. Thus we need that $|f4| + |f3| - e3 - e4 \leq 0$ in order for this inlining to be profitable.

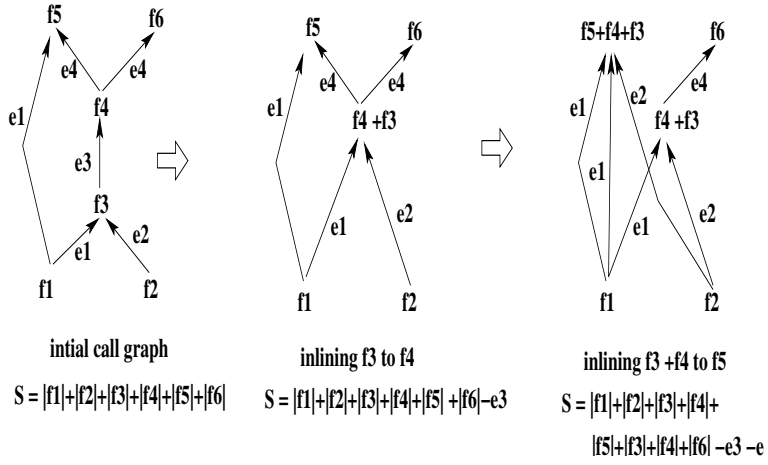

Figure 1: *Sequence of inline steps and the resulting size.*

Defining inlining this way allow us to specify the problem as follows:

Definition 2.1 *Given a call graph G^0 such that*

- *The nodes of G^0 are functions $f_1, f_2, \ldots$ and the size of each f_i ($|f_i|$ is attached.*
- *The edges of G^0 $f_i \longrightarrow f_j$ are labeled by the saving $f_i \xrightarrow{e_i} f_j$ that will be obtain after inlining the call $f_i \longrightarrow f_j$.*

The goal is to find a sequence of inlinings:

$G^1 = inline(f_i \xrightarrow{e} f_j \in G^0)$, $G^2 = inline(f_k \xrightarrow{e} f_m \in G^1)$, $\ldots \ldots$ $G^k = inline(f_z \xrightarrow{e} f_r \in G^{k-1})$,

for which The size $S = \sum_{f_i \in G^k} |f_i|$ is the smallest overall possible subsets of inlinings (excluding recursive calls).

Note that there are finite number of possible subsets hence this definition is valid as after each inline the number of edges in the resulting $G^t = inline(f_z \xrightarrow{e} f_r \in G^{t-1})$ is smaller than the edges in G^{t-1}.

3 Proposed Algorithm

LLVM inlining algorithm works by inlining G's nodes bottom-up in topological order. All its call-sites (calls to it from other functions) are examined to see if they are profitable for each node. A call site (edge) $f_i \xrightarrow{e} f_j$ is profitable if $|f_j| - e \leq threshod$. For the Oz option, this threshold = 5, basically meaning that the overall size is reduced due to this inlining. However, this bottom-up

order may not always be the correct one (i.e., obtain the optimal result). Figure 2 depicts a case wherein by selecting different values to e_i and $|f_i|$ it may be better to do either the upper inline (inlining-1) or the bottom inlining (inlining-2). If $|f_3| - 2e_3 < -e_1 - e_2$ then inlining-1 is preferable otherwise inlining-2 (LLVM style) is preferable. Also as can be seen eventually we get the same cost which, compare to the original cost S, is now $S + |f1| + |f2| + |f3| - 2 \cdot e_1 - 2 \cdot e_2 - 2 \cdot e_3$. This may not be beneficial compare to doing just one inlining. This proves that applying LLVM inlining (bottom-up) may not always be beneficial.

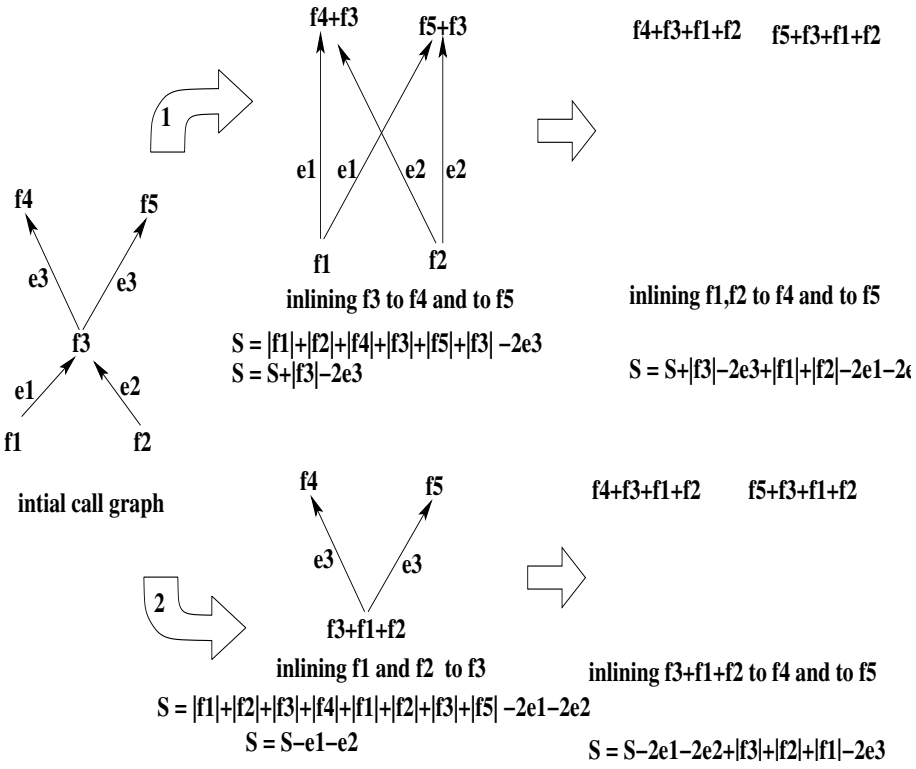

Figure 2: *Depend on the values of e_i and $|f_i|$ different inlining should be selected.*

Next we differentiate between two types of nodes:

Tree nodes- Nodes/functions with a single, i.e.functions with only one call-site. Tree nodes should always be inlined so that maximal tree-like sub-graphs are always inlined to the root caller. This is a safe move as, after inlining, the body of the callee does not remain in the executable, and size reduction is guaranteed. Hence inlining a node with in-degree one is always part of the optimal subset of selected inlinings.

Star nodes- Nodes/functions with more than one caller. A profitable star is a node that inlining it to all its callers will decrease the overall code size. Given $outdegree = d$ and edges $f_i \xrightarrow{e_i} f \;\; i = 1 \ldots \; d$ a profitable star satisfies that $\sum_{i=1}^{d} e_i \; + |f| \; \geq \; \sum_{i=1}^{d} |f|$.

Note that we have defined a profitable inline of a star as inlining of all the calls to this node. Thus we excluded the possibility that inlining some (but not all) calls to this node can be profitable.

This is based on the claim that if it is not profitable to inline all the node's outgoing edges, then inlining a partial subset of its edges can not be a part of an optimal solution (one that achieves the minimal size). The validity of this claim is due to the additive linear combination of the overall size and the fact that each outgoing edge of a node contributes the same saving to the final cost. Figure 3 is an example showing that partial inlining of a star's calls is not profitable. The initial G has seven nodes and a star at $f1$ with three calls ($f2 \longrightarrow f1,\ f3 \longrightarrow f1,\ f4 \longrightarrow f1$. The initial cost (size) is 206. The figure depicts three inlinings of the f1-star (left to right):

- Inlining only $f4 \longrightarrow f1$ and then completing the remaining inlining of tree nodes $f5 \longrightarrow f4 + f1$ and $f6 \longrightarrow f5 + f4 + f1$. This reduce the overall size to 196.
- Full inlining $f4 \longrightarrow f1$,$f3 \longrightarrow f1$,$f2 \longrightarrow f1$ and then completing the remaining inlining of tree nodes $f5 \longrightarrow f4{+}f1$ and $f6 \longrightarrow f5{+}f4{+}f1$. This increase the overall size to 236. and then completing the remaining inlining of tree nodes $f5 \longrightarrow f4{+}f1$ and $f6 \longrightarrow f5{+}f4{+}f1$.
- Inlining none of the calls to $f1$ and then completing the remaining inlining of tree nodes $f5 \longrightarrow f4$ and $f6 \longrightarrow f5 + f4$. Indeed this option obtains the best score reducing the overall size to 186 which is better then what is obtained by the partial inlining $f4 \longrightarrow f1$.

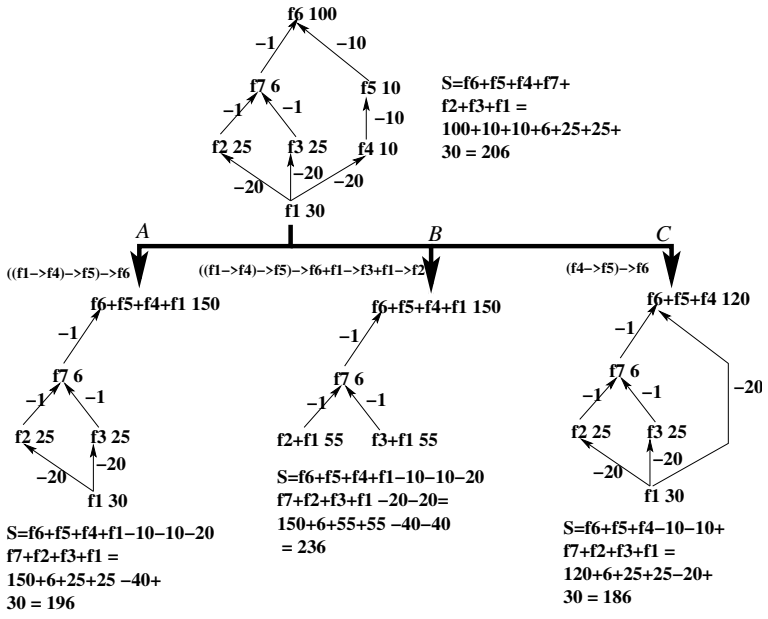

Figure 3: *Partial inlining of a star's calls is not beneficial.*

This could have lead to the following greedy algorithm:

1. Inline all the tree nodes until we remain with only star-nodes.
2. Compute the profit of the star nodes and inline all profitable stars in the order of their expected profit (i.e., inline the most profitable nodes first).
3. Repeat these two steps until there are no more sub-trees and profitable starts to inline.

The need to inline profitable stars according to their expected profit is illustrated in figure 4. Initially there are two profitable stars: one at node $f5$ and one star at node $f2$. The star at node $f5$ yields a profit of $f+7-6-6 = -5$ and the star at $f2$ yields a smaller profit of $10-6-6 = -2$. Inlining all other stars ($star-f1$, $star-f4$, $star-f3$) are not profitable since they have a positive cost, e.g. ($cost(star-f4) = 10-3-3 > 0$). Indeed starting with inlining the star at $f5$ yields a better size of 72.

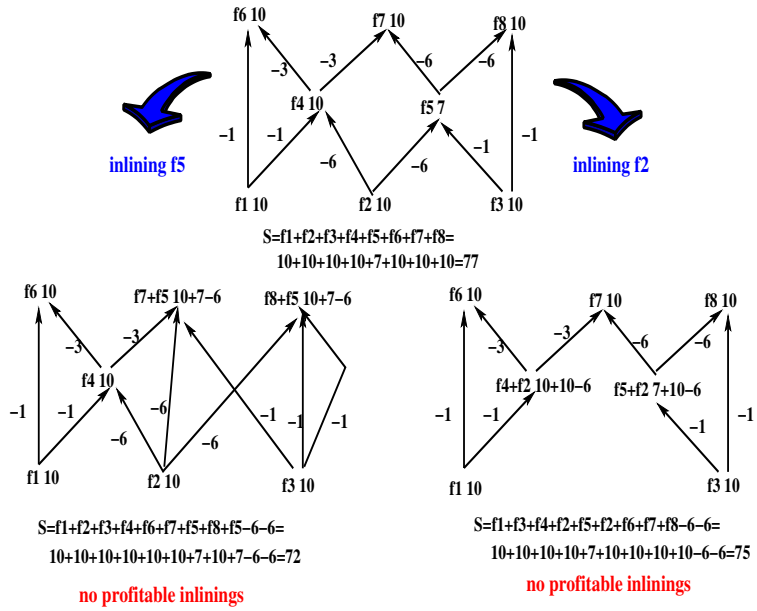

Figure 4: *Inlining the most profitable star.*

Figure 2 (the one used earlier to show that bottom-up order may be wrong) also shows that inlining trees first is not optimal. This is because we can set the numbers $(e_i, |f|)$ such after inlining $f3 \longrightarrow f2$, $f3 \longrightarrow f1$ the inlining of the star at $f3$ is no longer profitable. Recall that the figure depicts a case wherein by selecting different values to e_i and $|f_i|$ it may be better to either perform the upper inline (inlining-1) first or the bottom inlining (inlining-2). If $|f_3| - 2e_3 < -e_1 - e_2$ then inlining-1 is preferable otherwise inlining-2 is preferable. This reveals another more general problem, namely the "mutual effect" wherein inlining a node f can affect the profitability of inlining f's neighboring nodes (all its callers and its callee nodes). In figure 2 it can happen that after inlining $f3 \longrightarrow f1$ the neighboring star at $f3$ is now at size $|f3+f1-e_1$ which may become too "heavy" for inlining. Also, the numbers can be set such that if we inline the f3-star first, we transform the two tree nodes $f1, f2$ to stars (as is depicted by inlining-1). Now it can happen that originally the two tree-inlinings at $f1$ and $f2$ are more profitable than the inline of the f3-star; however, inlining $f3$ first will prevent us from inlining the more profitable $f1, f2$ inlinings. Figure 5 depicts the mutual effect of inlining a star-node on inlining its neighboring tree/star-nodes and the opposite effect of inlining a tree node on its neighboring star nodes. Clearly, inlining a tree node can not affect the profitability of inlining its neighboring tree nodes.

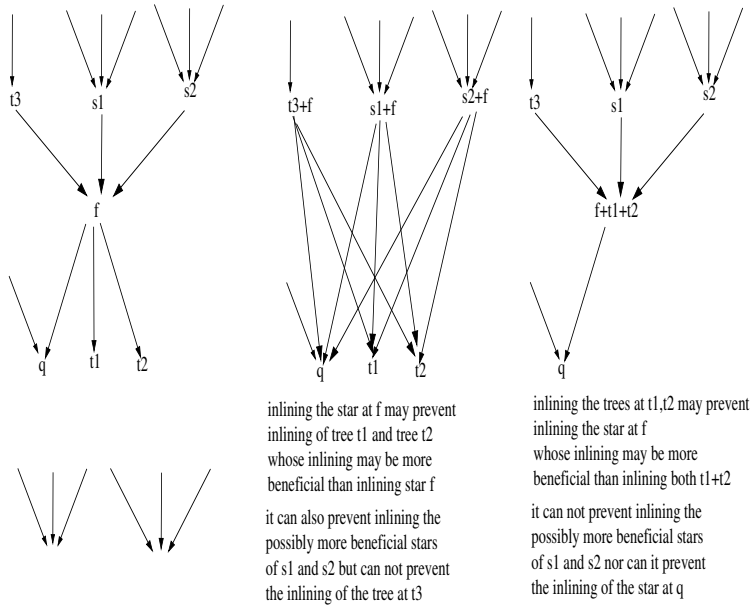

Figure 5: *Different cases of the mutual inlining effect.*

We can simplify the number of cases of the mutual effect by assuming that

- a star node $s1$ always prevents the inlining (conflicts) of all its more profitable parent star-nodes (callers).
- a star node $s1$ always prevents the inlining (conflicts) of all its more profitable sons tree (out-degree = 1) nodes (the callees).
- a tree node always prevents the inlining (conflicts) of all its more profitable father star nodes.

This leaves us with only two mutual-effect cases, i.e., conflict cases: A) between two profitable neighboring star-nodes and B) between a profitable tree-node and a profitable star-node.

Given the initial graph G^0 where

- The nodes of G^0 are functions $f_1, f_2, \ldots$ and the size of each f_i ($|f_i|$ is attached.
- The edges of G^0 $f_i \longrightarrow f_j$ are labeled by the saving $f_i \xrightarrow{e_i} f_j$ that will be obtain after inlining the call $f_i \longrightarrow f_j$.

the proposed inlining algorithm that consider the mutual inlining effect (called the MI algorithm) works as follows:

1. $t = 0$;
2. Select all the profitable nodes (star/tree-nodes) in G^t. If no profitable nodes are found, then exit the algorithm.
3. Compute a conflict graph CG whose nodes are all the profitable nodes of G^t and its edges (non-directed) correspond to conflicts between two neighboring nodes of CG.

4. Compute the Maximum Independent Set ($MXIS_{G^t}$) of CG [4]. Since Computing MXIS is np-hard we are using a greedy approximation algorithm. This greedy algorithm forms a maximum independent set by, at each step, choosing the next node f in CG with the highest $\frac{profit(v)}{degree(v)}$ and removing its neighbors.

5. The nodes of $MXIS(G^t)$ are inlined one after the other in the order they were added to $MXIS(G^t)$. This inlining of $MXIS(G^t)$ nodes forms a newly updated graph G^{t+1} as defined by the inlining operation.

6. A limited set of optimizations is applied on the nodes of G^{t+1}.

7. The saving on the edges and the size $|f|$ on each node are updated as well. For example, inlining $f_3 \xrightarrow{e_3} f_1$ will result in a new node $f3+f1$ with new weight $|f_3|+|f_1|-e_3$. In addition the saving of each of $f3+f1$ callers will be recomputed to adapt to the new modified body of f_3+f_1 as now more instruction may be eliminated in $f3+f1$ body due constants passed as parameters in $f3+f1$ caller. Changes in $|f_i|$ and e_i of G^{t+1} are also affected by the optimizations we have applied.

8. Iterate these steps until no more profitable nodes are found.

4 Comparing MI with LLVM's -Oz

Here, we compare the main features of the Mutual Inlining Algorithm (MI) to that of the LLVM Inliner with the -Oz option (Oz). We compare the inlining decisions of LLVM Oz vs. those of the MI algorithm. We use small examples and use actual printings of the call-graph G made by the LLVM's inliner (Oz/MI) during compilation (in dot format).

Inline Cost estimation - Both algorithms use the following measurements:

- function size (FS), namely number of instructions in the function body where the Oz counts the number of the IRs, and the MI counts the expected size in bytes of the IRs (using the expected size after code generation). This is done via unique conversion of the IR instructions to the expected RISC-V instructions (in bytes). This conversion was added to LLVM's inlineCost.cpp analyzer.
- Simplifies instructions (SI). For a given CS (call-site) SI counts the remaining instructions in $g()$'s body after eliminating instructions related to constant values passed to the callee at the call-site. Again Oz uses IRs; MI uses the byte size of IRs.
- Saving of inlining a CS due to elimination of parameters' passing and the call/return sequence (again Oz counts IRs NI counts byte-size of IRs).

Calculating the effectiveness of inlining a given CS - The Oz is pessimistic; it assumes that the body of the caller will remain either due to external calls from other modules or due to the possibility of an additional indirect call to g via a function-pointer which will not be inlined. The Oz, therefore, apply the inlining only if

$$SI - (Saving + staticbonus) < Threshold$$

where a static bonus is some value given in case that $g()$ is a static function. The MI is optimistic it assumes most functions are not referenced externally and that additional indirect calls to a function can be detected and hence inlined-avoided. Moreover, if there are external calls, the user could copy these functions to a common header file so that they will be inlined in the other module. The MI, therefore, assumes that if all calls to a function have been inlined then the body of this function will not remain in the executable. The MI, therefore, inline a CS ($f \longrightarrow g$):

- In case that there is only one call to g that is not in a conflict with a more profitable neighboring node, the MI will always inline; in comparison, the Oz may refrain from inlining if the body of g is too big. A simple program with a chain of calls (all nodes are tree nodes) was used to demonstrate the difference. Figure 6 depicts the call graph G of a given program. Here the saving on the edges contains two numbers: the left number is the cost in IRs as computed by the Oz inliner, and the right number is the saving in bytes as computed by the MI. The Oz will inline only if the cost (left number) is less equal to the threshold (5). None of the edges in figure 6 is profitable for Oz; consequently, the Oz did not inline any of the callees in the chain. The MI will inline all the tree-nodes as none of them is conflicting (the star at $f()$ is not profitable) and obtained the inlined program of figure 7. Consequently, compared to the Oz, the MI reduced the code size from 1266 bytes to 1154 bytes.

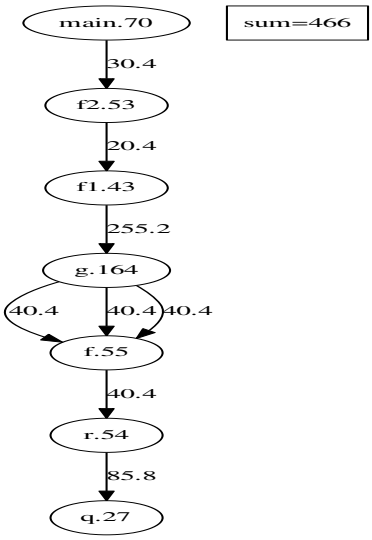

Figure 6: Program with a chain of calls

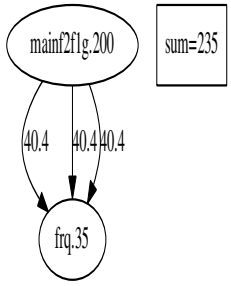

Figure 7: The Same program after MI's inlinings.

- In case that there is $k > 1$ calls to the same function (a star), the MI will either inline all the k calls or none depending if the total saving is greater than $(k-1) \cdot |callee|$ or not. The Oz examines each of the star's calls separately and decides to inline if the coast (the left number on edge) is less equal the $threshold = 5$. This can lead to two problems:
  - The Oz will refrain from inlining the star as none of its call-site is profitable despite

the fact that inlining the whole star is profitable. This is depicted in figure 8 showing a star at $f1()$ with two calls from $main()$. As the Oz cost for each call is 10, the Oz will inline none of $f1()$ calls. However, for the MI the f1-star is profitable, and compare to the Oz, the MI inlined the star reducing the size from 1036 bytes to 1020.

– The Oz will inline the stars calls since each one is profitable; however, this doesn't seem right since the start as a whole is not profitable. This is depicted in figure 9 showing a star at $f1()$ with ten calls from $main()$. All these calls are inlined by the Oz while the MI avoids inlining them ($9 \cdot 10 \leq 10 \cdot 9$). Reducing the size from 1324 bytes to 1172.

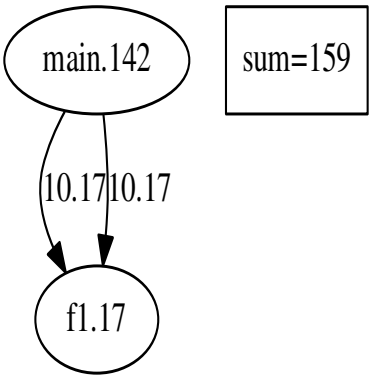

Figure 8: Oz refrain from inlining a star which is in fact profitable

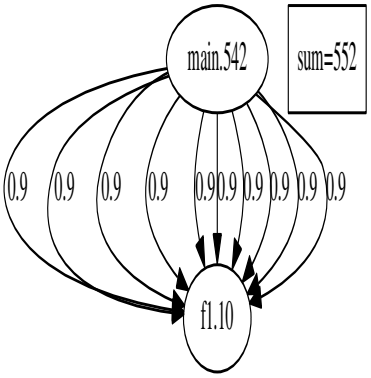

Figure 9: Oz inline a star which is in fact to profitable.

The mutual effect and multiple-rounds- Inlining one CS may render the usefulness of inlining another neighboring CS, which potentially could be more profitable than the first one. The MI checks such cases and selects the most profitable CS out of a conflicting set of CSS while the Oz checks each CS separately. In addition, the MI performs repeated rounds of inlining over G until no profitable rounds are found, while the Oz performs only one round of inlinings. Note that after each inline, the MI updates G, recomputing the callee body size and saving the neighboring CSS. Figure 10 depicts a program whose inline requires checking the mutual inlining effect and performing multiple inlining rounds. Here the Oz inline $f1() \longrightarrow g1()$ following the bottom-up inline order. This inline prevents the inlining of $g1() \longrightarrow q1()$ as $g1 + f1()$ is now too big, and indeed, the Oz does not perform any more inlinings. The MI's mutual inline effect prevent it from inlining $f1() \longrightarrow g1()$ first and instead the MI first performs the inline of $q1() \longrightarrow main()$ resulting by pushing the star at $g1()$ into main, i.e., $main + q1()$ now contains the four calls to $g1()$. This passes constant parameters to $g1()$ calls

making its inlining profitable. Thus next inline-round of MI will inlining all the remaining calls reducing the size from 1124 bytes to 1052.

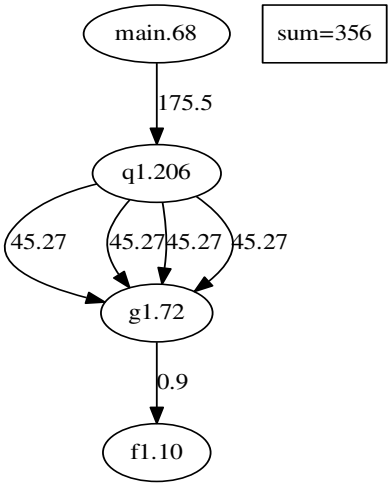

Figure 10: A program demonstrating mutual effect and multiple rounds

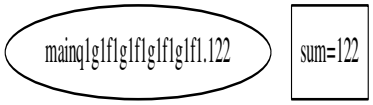

Figure 11: Final outcome.

5 Results

Results are given in the following table for a set of selected programs some from SPEC and some where arbitrarly selected from what was available at hand. The executable size where obtained using RISCV-unknown-elf-size measues the size inbytes adding the $text + data + bss$ sizes The Oz row indicate the sizes obtained by the LLVM's inliner with -Oz option used for minimizing the executable size. While the MI row show the size obtained by the MI algorithm. Some of the programs are in pure C-langauge and some are in C++ (as the inliner is part of the LLVM's optimization and hence works for any language supported by the LLVM).

```
Desc Calculator
   text            data     bss   total  diff
Oz 75928    2504     124   78556
MI 75528    2504     124   78156  400

SPEC BZIP2:
Oz 86066    5592    4380   96038
MI 85710    5592    4380   95682  356

Compiler GCC :
Oz 2906454    7428  561100 3474982
```

```
Particle simulation NBODY :
Oz 18581       0      40   18621
MI 18553       0      40   18593  28

SPEC MCF
Oz 38536    2484    6244   47264
MI 38356    2484    6244   47084 180

Highlevel synthesis tool ONE
Oz 65552    2486   96456  164494
MI 65212           2486   96456  164154 340

SPEC HMMER
Oz 162400    4874   81604  248878
MI 161988    4874   81628  248490 388

SPEC SOPLEX
Oz 572922  127331    6460  706713
MI 577354  123397    6460  707211 -498
```

6 Related works

the code bloat constraints can be mapped to the knapsack problem, which has been shown to be NP-complete [5]. [6] use inlining trails wherein the expected benefit of a possible inline can be computed by evaluating the costs and benefits resulting by pseudo-inlining a given call site. The saving in code size is determined by "group analysis", i.e., accounting for the effect of passing known static values into the callee's parameters (as explained, this is also used in LLVM and consequently in the method described in this work). [7] studies aggressive inlining and how it affects the performance and shows that such an inline improves performance significantly. [8] study different strategies for inlining different versions of the callee, where a version refers to how many inlines (from previous steps) this callee contains. It proposes to use a greedy strategy wherein, at each step, they select the call site that saves the maximal number of dynamic calls. [9] propose a branch-and-bound search algorithm to find a subset of n given functions such that the resulting code size is less than a given limit and that the performance is maximized. They assume that performance is related to the number of calls that occurs during execution; hence the selected subset of functions to be inlined should minimize this number. Profiling and simulation are used to determine the expected performance. [10] studies the effect of static and profile-based inlining heuristics for the Jalapefio dynamic optimizing compiler for Java. They also consider the problem of finding the most performance-profitable subset of inlinings under a restricted code-size budget and formalize this inlining optimization problem as a variant of the Knapsack problem. This work is mainly concerned with building the call graph and obtaining profile information to handle dynamic class-loading in Java programs. [11] improves the "temperature" inlining heuristic of the ORC compiler, where "temperature" combines the time spent in a procedure and the size of the procedure. They modify the temperature such that more inlinings will be performed more aggressively for small-size

benchmarks than large-size benchmarks and decrease the inlining of callees containing loops with a high trip count.

[12] uses a fast classifier called random forests to optimize inlining performance the execution time is a module that measures the longest execution path in the control flow graph of the compiler (called WECT analysis).

[13] compares three inlining techniques genetic algorithm, trained neural networks and using a pruned decision tree. All three methods are based on measuring the values of an elaborate set of features such as: caller/callee memory operations, execution frequency, caller/callee nested loops, and methods invocation. Their results (speedup gain) on specJVM 2008 show that all three methods had the same impact.

References

[1] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109, 2004.

[2] Wikipedia contributors. Inline function, 2019.

[3] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[4] Gen-Huey Chen, MT Kuo, and JP Sheu. An optimal time algorithm for finding a maximum weight independent set in a tree. *BIT Numerical Mathematics*, 28(2):353–356, 1988.

[5] Robert W Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, 1977.

[6] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 273–282, 1994.

[7] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. *ACM SIGPLAN Notices*, 32(5):134–145, 1997.

[8] Owen Kaser and CR Ramakrishnan. Evaluating inlining techniques. *Computer Languages*, 24(2):55–72, 1998.

[9] Rainer Leupers and Peter Marwedel. Function inlining under code size constraints for embedded processors. In *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No. 99CH37051)*, pages 253–256. IEEE, 1999.

[10] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 52–64, 2000.

[11] Peng Zhao and José Nelson Amaral. To inline or not to inline? enhanced inlining decisions. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 405–419. Springer, 2003.

[12] Paul Lokuciejewski, Fatih Gedikli, Peter Marwedel, and Katharina Morik. Automatic wcet reduction by machine learning based heuristics for function inlining. In *3rd workshop on statistical and machine learning approaches to architectures and compilation (SMART)*, pages 1–15, 2009.

[13] Sameer Kulkarni, John Cavazos, Christian Wimmer, and Douglas Simon. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12. IEEE, 2013.