Aggressive Function Splitting for Partial Inlining[1]

Jun-Pyo Lee, Jae-Jin Kim, Soo-Mook Moon
School of Electrical Engineering and Computer Science,
Seoul National University, Seoul, Korea
{walker, kjj7999, smoon }@altair.snu.ac.kr

Suhyun Kim
Korea Institute of Science and Technology (KIST)

dr.suhyun.kim @gmail.com

Abstract

Partial inlining is an efficient way of inlining, which inlines only part of the callee function, thus reducing the code expansion. The key problem is how to split the callee function effectively so that both the call overhead and the code expansion can be reduced. Previous techniques either lead to function splits too large to be inlined, or fail to reduce the call overhead effectively. In this paper, we propose a new technique for function splitting based on early returns in the function to achieve both goals. It also employs the cost and benefit model to allow efficient function splitting. Our preliminary experimental results show that the proposed technique can reduce the call overhead by 33%, with the code size increase of 5.8%.

1. Introduction

Inlining is known as an effective way to reduce the function call overhead by duplicating the body of the callee function into the call site of the caller function. Since inlining increases the code size, hence the memory pressure, it is not desirable to inline functions that are too large. In fact, many techniques have been proposed to reduce the code growth while preserving the benefit of inlining [4, 5, 7, 8, 9]. One such a technique is partial inlining [4, 5], which inlines only part of the callee function.

Partial inlining is usually performed in three steps: (1) Construct a subgraph of the callee function which will actually be inlined. The subgraph should include the entry basic block (BB) and at least one exit BB. (2) Split the callee function into two or more functions based on the subgraph. Those BBs that are not part of the subgraph are extracted from the original function and form a separate function, called an outlined function. For each edge from the subgraph to the outlined function, we add a transition function call, with variables used in the outlined function as parameters. Now, only the subgraph is left in the original function, which is called a leftover function. (3) Inline the leftover function into the call site(s), exactly as in normal inlining.

Figure 1 illustrates the idea of partial inlining. One BB in the left path of the function foo() is extracted to an outlined function while the entry BB, the exit BB, and the right BB compose the leftover function. A function call is added from the leftover function to the outlined function, with variables defined and used at the outlined function body as parameters. Only the leftover function will be inlined at the call site.

There are a couple of issues for partial inlining. First, it should be noted that even if the callee function is partially inlined so that the call overhead from the caller to callee is removed, there are still calls from the leftover function to the outlined function. So, it is important to minimize such calls to achieve the best effect of inlining. Secondly, if the leftover function is still too large even after the function splitting, inlining it is costly. Finally, the additional parameters in the call from the leftover to outlined function may cause some overhead, so they should be reduced.

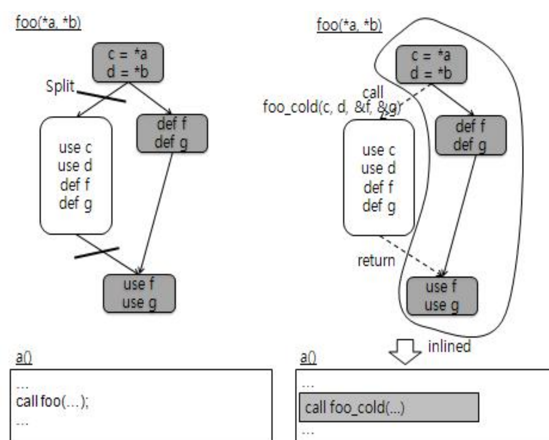

Figure 1. Overview of Partial Inlining

1 This work was supported by the IT R&D program of MKE/KEIT [KI002119, Development of New Virtual Machine Specification and Technology].

All of these issues are related to the function splitting, so the subgraph construction is most important for partial inlining. There are two previous approaches to the subgraph construction. One is removing those cold BBs that are rarely executed based on branch profiles [5]. Although this can lead to small call overhead since the call from the leftover to the outlined would be rarely executed, the leftover function would be still too large since only those BBs with extremely low frequency will be outlined.

The other approach is starting from the hottest seed BBs in the function and growing towards the entry BB and the exit BBs [4]. The problem of this approach is that the hot seed BBs are not necessarily executed in every call to the original function, i.e., many calls may return without executing hot BBs, and thus partial inlining cannot eliminate such calls.

Neither approach employs the cost and the benefit model for partial inlining, unlike in regular inlining [6]. That is, the cost involved with the size of the leftover function or with the parameter overhead of transition calls is not considered, while the benefit of reducing calls is not estimated. This would lead to some adhoc way of the subgraph construction.

In this paper, we propose a new subgraph construction approach which employs the edge to an exit BB as a seed. Since this edge has a return ratio, it can be used to assure how much call overhead is eliminated by including this edge. Also, we apply a detailed cost and benefit analysis for each subgraph and select one that has the highest performance effect.

The rest of the paper is structured as follows. Section 2 introduces motivating example of our work. Section 3 presents key implementation details with algorithms and the concept of subgraph we used for partial inlining. Section 4 shows the implementation on top of LLVM. Preliminary experimental results are reported in Section 5. Related work is discussed in section 6 and the summary follows in Section 6.

2. Observation of Function Splitting

Figure 2 (a) shows the control flow graph (CFG) of an example function, with its edges annotated with the relative execution frequency. It has a hot loop where bb4 is the hottest BB. The two exit BBs, `exit1` and `exit2` (which include a return statement), cover 25% and 75% of calls, respectively.

Let us construct a subgraph for this function. If we take the approach in [5], figure 2 (b) would be the result. Only the BB `exit2` is outlined since the branch profile indicates that both edges to `exit2` are highly biased, meaning that it is too cold.

On the other hand, if we take the approach in [4], we construct a subgraph starting from bb4 as a hot seed BB. Figure 2 (c) would be the result where bb3 and `exit1` are outlined.

Unfortunately, neither result is attractive. In Figure 2 (b), only one BB is outlined, so the leftover function is still too large to be inlined, although it can remove 75% of calls when inlined since it includes `exit1`. In Figure 2 (c), the leftover function has fewer BBs, but it can eliminate only 20% of the calls when inlined and the remaining 80% of the calls will be transferred to the outlined function via a function call.

Figure 3 (a) would be a better function splitting result. The leftover function can reduces 75% of calls when inlined, while only four BBs are left. Figure 3 (b) shows the final CFG after function splitting is completed. A separate outlined function is created for each transition edge, which leads to the duplication of `exit2`. This is simpler and cheaper than making a single outlined function that can handle all entry points.

We can make two observations from these results. One is that even BBs with highest execution frequency would better be outlined sometimes, so taking the hottest BB as a seed is not always desirable. Instead, it is better to take an edge to an exit BB with a high execution frequency as a seed since if the edge is included in the leftover function, that frequency of calls are guaranteed to be removed. We expand the seed by including predecessors of the chosen exit BB recursively until we include the entry BB. In Figure 2 (a), the edge from bb3 to `exit1` can be taken as a seed, which is expanded by including their predecessors, bb1 and entry, composing the subgraph in Figure 3 (a).

Another observation is that we must split functions considering the call overhead of the outlines functions, which includes the parameter passing cost as well as the call frequency itself. Creating a separate function for each transition edge is a way of reducing the call overhead. We also need to consider the increase of the code as a result of partial inlining. Consequently, it is desirable to introduce a cost-benefit model for partial inlining.

These observations lead to our function splitting algorithm, which is based on *early returns*.

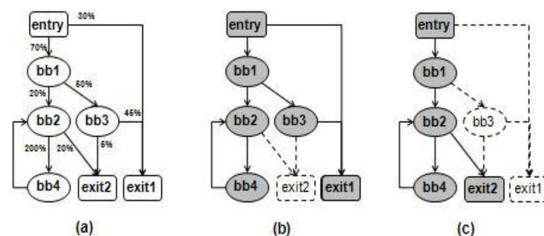

Figure 2. Sample CFG and Previous Function Splitting

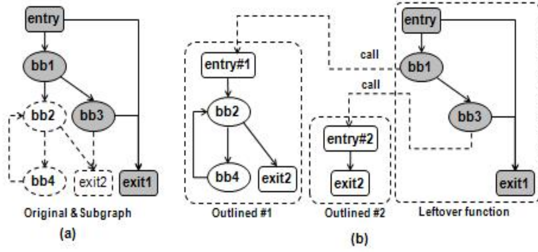

Figure 3. Better Function Splitting Result

3. Function Splitting based on Early Returns

Previous section introduced the approach of function splitting based on an exit edge with high execution frequency, which leads to a small-sized leftover function and a large reduction of call overhead. It also proposed the cost-benefit model to function splitting. This section provides the detail of such function splitting.

3.1. Early Return Subgraph

Let us define a *return edge* as an edge whose target is an exit BB containing a return instruction. Then, an *n-early return subgraph* (*n-ER subgraph*) can be defined as a *minimum* subgraph which contains *n* different return edges of the CFG. If there is a path from the entry BB to one of those n return edges, all the BBs on the path should be included in the n-ER subgraph. For a given *n-ER* subgraph, its *call coverage* is defined as the sum of execution frequencies of its return edges. Figure 4 (a) shows the original CFG. Figure 4 (b)-(d) shows examples of n-ER subgraphs which are composed of shaded BBs.

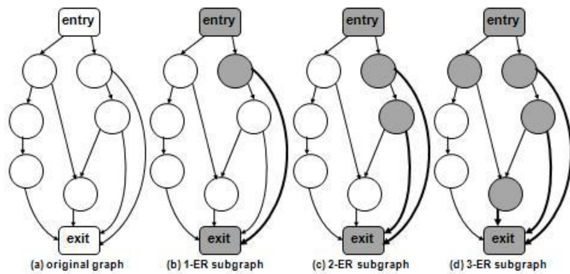

Figure 4. n-ER subgraph

3.2. ER Subgraph Construction

The 1-ER subgraph can be easily constructed by taking a return edge and adding its source BB and exit BB. Then, we take the source BB as a seed and recursively add its predecessor BBs until the entry BB is added. The n-ER subgraph also can be constructed by simple recursion: (1) we first construct every possible 1-ER subgraph, and then, based on benefit-to-cost ratio (see Section 3.3), we choose the 1-ER subgraph with the highest ratio; (2) we then construct every possible 2-ER subgraph by adding one more return edge (and the corresponding BBs) to the 1-ER subgraph constructed in (1) and then choose the best 2-ER subgraph with the highest benefit-to-cost ratio; (3) we repeat this process until the (*n+1*)-ER subgraph is too large to be inlined. Then the n-ER subgraph will be the leftover function. Figure 5 describes this process in more detail.

```
function UpdateERG(RetEdge, PrevEdges, PrevER)
  CurrEdges := PrevEdges ∪ {RetEdge};
  CurrER := PrevER ∪ {RetEdge.Src, RetEdge.Dst};

  Stack .push(RetEdge.Src);

  while Stack is not empty
   BB := Stack.pop();
   for each predecessor P of BB
       if P ∉ CurrER
         CurrER := CurrER ∪ {P};
         Stack.push(P);
         for each successor S of P
          if S is return edge;
               CurrEdges := CurrEdges ∪ {P, S};
          fi;
          end for;
       fi;
     end for;
  end while;
  if CurrER is larger than threshold
    return (NIL, NIL);
  else
    return (CurrER, CurEdges);
  fi;
end UpdateERG

function ConstructERG(Func)
   TotalEdges := set of total return edges of Func;
   do
     for each edge E in TotalEdges
       (CurrER, CurrEdges) := UpdateERG(E, PrevER,
PrevEdges);
       if Benefit2CostRatio CurrER > Benefit2CostRatio ER
         (ER, Edges) := (CurrER, CurrEdges);
       fi;
      end for;
    (PrevER, PrevEdges) := (ER, Edges);
   while ER is updated
   return ER
end ConstructERG
```

Figure 5. *n*-ER Subgraph Construction Algorithm

3.3. Cost and Benefit Estimation

Generally, the cost and benefit of the regular inlining can be estimated by the increased code size and the reduced number of function calls, respectively. With partial inlining, however, both estimations should be updated such that the code size increase comes only from the leftover function inlined while only part of the calls are reduced even after inlining since those calls to outlined functions are added. Moreover, the overhead of calling outlined functions possibly with more parameters than the original function parameters and the code size increase for the outlined function should be considered as well.

We first define the benefit of partial inlining as follows:

$$\text{Benefit} = \text{reduced function call overhead by ERs} - \text{additional overhead by calling outlined function}$$

Function call incurs several overhead including control transfer, register save/restore, parameter passing, and etc. Although the overhead can vary significantly with the location of call site and the time when the call happens, it can be generalized as sum of the fixed call overhead (called *FixedCallCost* below) and dynamic call overhead which is proportional to the number of parameters. To make the calculation simple, FixedCallCost is defined as the relative overhead of general function call when the overhead of each parameter passing is estimated to one (this is architecture dependent value, though).

$$\text{CallOverhead} = \text{FixedCallCost} + \text{\# of parameters}$$

Even though the outlined function is called, function call count is not changed, since the original call is already inlined. So, based on the CallOverhead definition above, the increased overhead of calling outlined function can be defined as the difference in number of parameters between original function and outlined function. Including this, the benefit can be defined as below. An outlined function is created for each exit from the subgraph to outlined function, which can have different number of parameters.

$$\text{Benefit} = (\textstyle\sum \text{ER ratio}) * (\text{FixedCallCost} + \text{\# of params}) - \textstyle\sum_i \text{exit_ratio}(i) * (\text{diff. in \# of params of outlined}_i)$$

Now we discuss the cost, which can be simply estimated by the increased code size. It should be noted that only the leftover function is inlined at each call site, while the outlined functions are shared among them. Also, the cost of outlined function size should be shared among multiple call sites within the leftover function. Thus, the cost of partial inlining can be defined as below.

$$\text{Cost} := (\text{size of leftover func.}) * (\text{\# of call site}) + \textstyle\sum \text{size of outlined function}$$

Finally, we need to define *Benefit2CostRatio* which controls subgraph selection. Simple benefit to cost ratio is not attractive, since it may prevent aggressive function splitting. For example, let us assume that 1-ER subgraph has relatively large call coverage with small code size. If another return edge has similar call coverage, but requires slightly larger code size increase than the code size of previous 1-ER graph. In this case, 1-ER has always highest benefit to cost ratio and further opportunities to reduce call overhead cannot be exploited. Thus, we need a metric which allows aggressive subgraph construction when total code size is relatively small. This is possible by adding constant to Cost, and Benefit2CostRatio is defined below. Here, CostNormalizeFactor is determined by extensive testing.

$$\text{Benefit2CostRatio} := \text{Benefit} / (1 + \text{Cost} * \text{CostNormalizeFactor})$$

3.4. Function Splitting

After the ER Subgraph is identified, we clone the function so that the cloned function is used for function splitting. The original function will be used for indirect call sites or call sites with low execution frequency. With the cloned function, we extract those BBs which are not part of the ER subgraph and form an outlined function. The control and data flows between the ER subgraph and the outlined function should be reconciled as follows:

(a) ER subgraph → outlined function

As to the control flow, a function call is needed at the transition edge. As to the data flow, all used and defined variables in the outlined function are passed as parameters (however, by duplicating the return block, it can be avoided to pass the address of defined variables as parameters, and the outlined function should have same return value as in the original function). If there are multiple transition edges from the ER subgraph to the outlined function, there can be two choices: (1) make a separate function for each transition edge, or (2) make a single function which serves all the transition edges. For the case of (2), another parameter is needed to indicate which edge is taken to enter the function, in addition to the superset of all the parameters needed at each edge. Consequently, (1) is simpler and cheaper to implement

and its code size overhead is already considered by the cost and benefit analysis.

(b) Outlined function → ER subgraph

Control joins from the outlined functions to the ER subgraph is avoided by code duplication. If any BB within the ER subgraph has predecessor BBs from the outlined function, the BB and its successors are duplicated. With this duplication, if the outlined function is called, the control never returns to the ER subgraph, but returns to the caller directly.

4. Implementation on top of the LLVM

We implemented the proposed partial inlining on top of the low-level virtual machine (LLVM) compiler infrastructure (version 2.6) [1]. LLVM has its own virtual instruction set architecture called LLVA [2], which is language and architecture neutral. The same LLVA representation is used uniformly throughout the compile time and the link time, and the LLVA code is available even after linking [3]. Proposed partial inlining can be applied at any stage, yet we apply it after linking in the current implementation. This helps isolating the impact of partial inlining, but it also keeps the leftover function from becoming larger by inlining other functions.

We employed the edge profiler of the LLVM to get the edge profiles. Also, after partial inlining is done, several optimization passes in the LLVM are applied to exploit additional optimization opportunities which partial inlining allows. When we measure the runtime impact of partial inlining, the base version is created by applying the same optimization passes after linking is done.

5. Preliminary Experimental Results

This section shows some preliminary experimental results with the proposed partial inlining.

5.1. Experimental Setup

Our experiments are performed on a machine with Intel Core i5 CPU 2.67GHz, 3GB memory, 32KB L1 cache, 256KB L2 cache, and 8MB L3 cache. The benchmarks are SPEC2000 integer benchmarks [10]. All the experiments are done using one set of the reference input (if there are multiple set), except the training run which uses the train input. For runtime measurement, we take the minimum value in five consecutive identical runs.

5.2. Static Result

Table 1 shows the inlining statistics of partial inlining and the code size result. Each function is categorized into three groups:

- Cold represents a function for which partial inlining is not employed, since the profile information indicates that there is no beneficial call site to that function.
- NoSubG represents a function for which our partial inliner fails to construct a beneficial subgraph based on the cost and benefit analysis.
- Inlined represents a function which is partially inlined successfully.

Table 1. Static Results of Partial Inlining

	# of functions			# of inlined callsites	Binary size increase
	Cold	NoSubG	Inlined		
gzip	15	3	1	10	1.001
vpr	54	3	4	5	1.003
gcc	790	107	147	602	1.236
crafty	37	13	10	69	1.073
parser	24	47	28	96	1.192
eon	488	2	4	6	1.000
perlbmk	692	16	8	16	1.013
gap	691	18	15	57	1.043
bzip2	4	5	1	3	1.000
twolf	55	10	7	14	1.022
Average					1.058

Table 1 shows that `eon`, `perlbmk` and `gap` has a large number of Cold functions compared to other functions. These benchmarks have many indirect call sites which cannot be identified by the profiler. In order to apply partial inlining for these benchmarks, other techniques such as devirtualization [15] would be needed.

On average, the code size increases by 5.8%. In `gcc` and `parser`, the code size increases substantially. Even for these two benchmarks, many functions are not inlined, so the code increase is not due to excessive partial inlining. In the next section, we will see if the code size increase contributes for reducing call overhead.

5.3. Runtime Results

Figure 6 shows the effect of partial inlining in terms of the call count, by comparing the total number of call counts for the original program and the partial-inlined program. The graph shows that partial inlining achieves around 33% call count reduction on average. Unfortunately, this does not directly lead to the performance improvement, as shown in Figure 7. It

shows the runtime speedup with partial inlining. There are some marginal (1%~4%) performance improvements for 6 out of 10 benchmarks.

One reason would be that the LLVM already has an aggressive inlining policy, and thus, the majority of call overhead would have already been removed by the LLVM since we apply partial inlining after linking. Moreover, the LLVM is known to perform powerful inter-procedural analyses and optimizations, which are usually enabled by its inlining. For the case of our partial inlining, we applied limited optimizations only. So we have to investigate the impact of partial inlining to existing compiler optimization fully, so as to allow more optimizations with partial inlining.

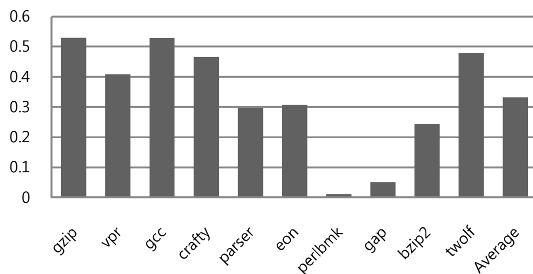

Figure 6. Total Call Count Reduction Ratio

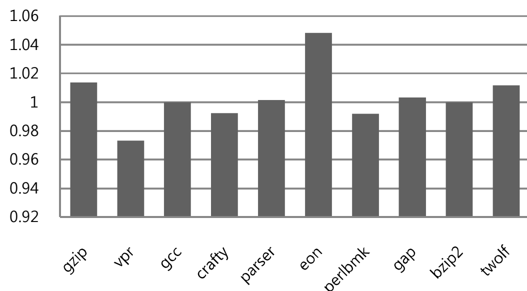

Figure 7. Runtime Speedup

5.4. Dynamic Call Count Statistics

Figure 8 shows the decomposition of dynamic call count when partial inlining is applied using profile information by train input. Each portion represents dynamic call count of corresponding functions which follows the definition in section 5.2 except Cold portion. Due to low execution frequency, there are call sites which call the original function even though the callee function is partially inlined. So, the call count of the original function is counted as part of Cold portion.

Large portion of Cold in several benchmarks

For those benchmarks which have a large portion of Cold in the static count, we can see a similar result here. To check the impact of indirect call, we manually subtracted indirect call count from the Cold portion and obtained Figure 9. Since a function can be called from both a direct call site and an indirect call site, subtracting indirect call count from Cold portion is not totally valid. However, in most benchmarks, it shows that Cold portion would have a meaningless value if the indirect call effect is eliminated. `Perlbmk` is one of exceptional case, where large portion of Cold comes from incomplete profile information (with profile information by reference input, a large portion of Cold is moved to NoSubgraph). The other exception is `gcc`. Several functions in `gcc` usually have hundreds of call sites, of which, the majority of them are not inlined due to low execution frequency. However, there are hundreds of those call sites, the summation of those call sites contributes to the relatively large portion of Cold in `gcc`.

Large portion of NoSubgraph

In many benchmarks, NoSubgraph takes large portion. This is more clear when the impact of indirect call is removed in figure 9. The portion of NoSubgraph means that our subgraph construction algorithm fails to identify useful subgraph for inlining, and thus we may miss some opportunity to reduce call overhead more. We investigated the reason why subgraph construction failed for several hot functions, and found that in many cases, there exists only one or two return edge(s). Since our algorithm relies on the return edge, if there is only one return edge, it cannot construct subgraph.

As a future work, our subgraph construction algorithm needs to be improved to handle such a case where only limited number of return edges exists. One solution would be splitting control join BB to make multiple return edges. First, selects a control join BB which is nearest from the return edge. Second, duplicates conrol join BB for each incoming edge to the BB. Third, duplicates other BBs between control join BB and return edge if exists. Now, we have a return BB which has multiple return edges, and can apply the subgraph construction algorithm.

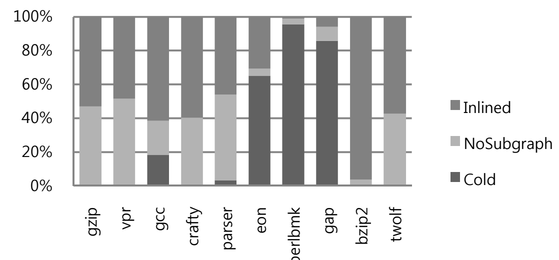

Figure 8. Dynamic Call Count Decomposition

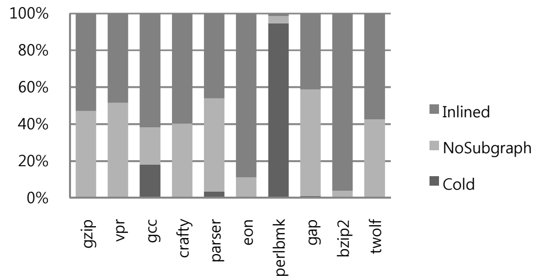

Figure 9. Dynamic Call Count Decomposition (Manually subtracting indirect call count from Cold)

5.5. Evaluation of Subgraph Construction Algorithm

Figure 10 shows the effectiveness of our subgraph construction algorithm, when partial inlining is performed using profile information by train input. To show the effectiveness, we define the following ratio:

- *subgraph reduction ratio* is defined as average of the subgraph size to entire function size ratio.
- *effective inline ratio* is a the probability of inlined function does not call outlined function, and defined as below.

$$\textit{effective inline ratio} := 1 - \text{outlined function call count}/\text{inlined function call count}$$

Subgraph reduction ratio means how small the subgraph is constructed compared to the original function. If this value is low enough, then we can say that our algorithm successfully reduces a function for inlining. In [5], the average subgraph reduction ratio is reported as around 15%. So, the average number slightly around 50% is very promising result.

Effective inline ratio means that for each inlined function, what portion of call overhead is really reduced by partial inling. In most benchmarks, this ratio is over 80%. Around 80% of effective inline ratio is relatively high compared to its low counterpart, subgraph reduction ratio.

From the result, we can say that on average, our partial inliner eliminates over 80% call overhead by inlining only around 50% code of original function. So, the gap between these two ratios can be seen as indicating the effectiveness of our subgraph construction algorithm.

6. Related Work

The most relevant work is done by R. Muth and S. Debray [4]. As in our approach, they defined subgraph

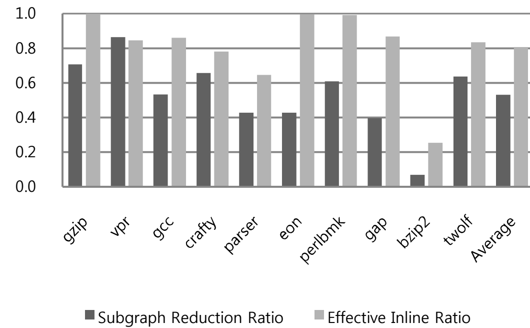

Figure 10. Effectiveness of subgraph construction algorithm

(which is called critical subgraph) for inlining purpose and the remaining basic blocks are extracted. Also, they do not allow control join from the outlined function to critical subgraph. However, they generate single outlined function for all transition edges from the subgraph, which seems like to increase of calling outlined function. Also, their purpose is not to enable aggressive inlining of large functions, but to replace full inlining with partial inlining. Thus, they calculate critical graph to cover large portion (usually over 90%) of dynamic instruction count from the hottest blocks as seed. As a result, their critical subgraph is fairly large compared to ours and usually contains loop. According to their result, although it is successful to reduce code growth, the performance is degraded compared to full inlining counterpart. It also looks like their implementation is not fully integrated with the existing compiler environment.

Another similar work is done by P. Zhao and JN. Amaral [5]. They used partial inlining not to replace conventional inlining, but to increase inlining opportunity of large function. Like us, all the existing compiler optimization passes can be applied after partial inlining is performed, although in our case, we can achieve this due to LLVA representation which makes it possible to perform any compiler optimization even after link-time. Similar to us, the extracted cold region forms separate function. Unlike our approach, however, when extracting cold region, control join to leftover function is allowed. Even though they tried to minimize the overhead related to control join, their approach is moving most of overhead to cold region and keeping the overhead within hot region as small as possible. Thus, the overhead of calling outlined function remains still large. This prevents aggressive outlining and results in large leftover function. According to their experimental result, the average function size is only reduced by 10% ~20% for most benchmarks.

In dynamic compilation environment, Suganuma et al. [7] proposed similar partial inlining mechanism in their region-based compilation framework for Java. Since it is based on dynamic compiler, the outlined function is not generated until it is really called. Usually, calling outlined function costs fairly high due to on stack replacement (OSR), and thus their region selection is very conservative, i.e., they extracts only rarely executed blocks.

There are several works regarding general inlining heuristics [6, 8, 9] and inline analysis phase ordering [13] which can be used at online and/or offline. In our work, we focused on the function splitting, and inlining is only controlled by execution frequency of call site. So, most of work regarding inlining decision is orthogonal to our work and can be used together.

7. Summary

Our function outlining approach aggressively splits a function based on the subgraph which is obtained starting from a return edge. Return edge is very crucial in constructing a subgraph since the whole purpose is reducing call overhead. Our algorithm successfully constructs a subgraph whose size is the half of the original function on average.

The price of smaller leftover function is frequent calls to outlined function. To control the overhead of calling outlined function, we provide two solutions. First, the overhead of calling outlined function is minimized by code duplication. For each transition edge from the subgraph to the outlined function, separate function is created. Also, if there exists a control path from outlined function back to the subgraph, required BBs are duplicated not to allow control join, which makes the outlined function simple. Second, the cost of calling outlined function is considered when calculating the cost and benefit of partial inlining, which prevents creation of large outlined function with high execution frequency. Our cost and benefit analysis achieves relatively low call ratio to outlined function, which assures that the overhead of calling outlined function is well controlled.

With experimental study, several issues are found. First, there are still many opportunities which are missed by current approach: indirect call site and a function with single return edge. Our work should be extended to deal with those cases. Second, the runtime performance does not improved much. To make partial inlining more attractive, additional optimization passes should be studied further to achieve better runtime result.

8. References

[1] http://llvm.org

[2] V. Adve, C. Lattner, M. Brukman, A. Shulka, and B. Gaeke, “LLVA: A Low-level Virtual Instruction Set Architecture”, In *Proc. Int’l Symposium on Microarchitecture (MICRO-36)*, San Diego, CA, Dec. 2003.

[3] C. Latter and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”, In *Proc. Int’l Symposium on Code Generation and Optimization (CGO’04)*, Palo Alto, CA, Mar. 2004.

[4] R. Muth and S. Debray, “Partial Inlining”, *Technical Report*, Department of Computer Science, University of Arizona, 1997.

[5] P. Zhao and JN. Amaral, “Ablego: a function outlining and partial inlining framework”, *Software:Practice and Experience*, **37**(5):465-491, Apr. 2007.

[6] P. Zhao and JN. Amaral, “To Inline or Not to Inline? Enhanced Inlining Decisions”, In *Proc. 4th Workshop on Languages and Compilers for Parallel Computing(LCPC)*, College Station, Texas, Oct. 2003.

[7] T. Suganuma, T. Yasue and T. Nakatani, “A Region-Based Compilation Technique for a Java Just-In-Time Compiler”, In *Proc. Int’l Conference on Programming Language Design and Implementation(PLDI)*, San Diego, CA, Jun. 2003

[8] J. Cavazos and M.F.P. O’Boyle, “Automatic Tuning of Inlining Heuristics”, In *Proc. Supercomputing 2005 Conference*, Nov. 2005

[9] X. Zhou, L. Yan, and J. Lilius, “Function Inlining in Embedded Systems with Code Size Limitation”, In *Proc. Int’l Conference on Embedded Software and Systems(ICESS)*, 2007

[10] http://www.spec.org

[11] K. Pettis and RC. Hansen, “Profile Guided Code Positioning”, In *Proc. Int’l Conference on Programming Language Design and Implementation(PLDI),* New York, 1990.

[12] A. Ayers, R. Schooler and R. Gottlieb, “Aggressive Inlining”, In *Proc. Int’l Conference on Programming Language Design and Implementation(PLDI),* May, 1997.

[13] D.R. Chakrabarati and S.-M. Liu, “Inline Analysis: Beyond Selection Heuristics”, In *Proc. Int’l Symposium on Code Generation and Optimization (CGO’04)*, 2006.

[14] D.R. Chakrabarati, L.A. Lozano, X.D. Li, R. Hundt and S.-M. Liu, “Scalable high performance cross-module inlining”, In *Proc. Int’l Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2004.

[15] D.F. Bacon and P.F. Sweeny, “Fast static analysis of C++ virtual function calls”, In *Proc. ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, 1996.