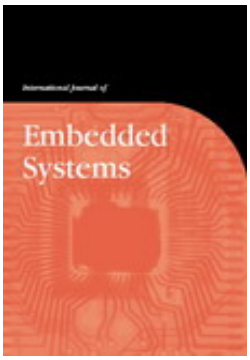


International Journal of Embedded Systems

ISSN online: 1741-1076 - ISSN print: 1741-1068
https://www.inderscience.com/ijes

Call-site tree and its application in function inlining

Arthur Ning-Chih Yang, Shih-Kun Huang, Wuu Yang

DOI: 10.1504/IJES.2024.10064933

Article History:

Received:	04 July 2023
Last revised:	06 October 2023
Accepted:	16 February 2024
Published online:	30 July 2024

Call-site tree and its application in function inlining

Arthur Ning-Chih Yang, Shih-Kun Huang and Wuu Yang*

Department of Computer Science,
National Yang-Ming Chiao-Tung University,
Hsinchu, Taiwan
Email: goldloti2@gmail.com
Email: skhuang@cs.nycu.edu.tw
Email: wuuyang@cs.nycu.edu.tw
*Corresponding author

Abstract: Traditionally, function invocation is represented as the (static) call graph or the (dynamic) execution tree in compilers. We define the new call-site tree, in which two different executions of a call-site (say α that is located within a function f) are represented by the same node if and only if the calling chains from $main$ to f in the two different executions of α are identical. Function inlining is a very profitable optimisation that replaces a call-site with the body of the called function. Intuitively, it is preferable to inline the call-sites that are executed most often. Call-sites are suitable for function inlining because they allow to adjust the execution counts of (new and existing) call-sites without re-profiling after a call-site is inlined. We also propose analysis algorithms of the call-site tree and implement an inliner in LLVM. The experimental results on SPEC INT 2006 are reported.

Keywords: inlining; call graph; call-site trees; compiler; execution tree; optimisation; transformation; programming language; LLVM.

Reference to this paper should be made as follows: Yang, A.N-C., Huang, S-K. and Yang, W. (2024) 'Call-site tree and its application in function inlining', *Int. J. Embedded Systems*, Vol. 17, No. 5, pp.1–12.

Biographical notes: Arthur Ning-Chih Yang received his Master's degree in Computer Science and Information Engineering from the National Chiao Tung University. He current works in an IC design house.

Shih-Kun Huang received his PhD in 1996 in Computer Science from the National Chiao Tung University. He is a Professor at the National Yang Ming Chiao Tung University. His research integrates software engineering and programming languages to study cyber security and software attacks.

Wuu Yang received his PhD in Computer Science from the University of Wisconsin at Madison in 1990. He has been a Professor in the National Chiao-Tung University since 1992. His research interests include systems software for embedded systems, programming languages and compilers, and attribute grammars.

1 Introduction

Traditionally, function invocation in a program is represented as the (static) call graph (Ryder, 1979), in which each node represents a function, not a call-site.The call graph is too small to distinguish different call-sites. Some compiler optimisations may perform different transformations to different call-sites of the same function. Obviously, the call graph is not useful in this case. On the other hand, the (dynamic) execution tree (see Figure 6), in which each node represents an invocation of a function, is too big, especially for deep recursive programs, for compiler optimisation. When we perform function inlining, neither the call graph nor the execution tree is suitable for our purpose. Therefore, we define a new data structure, called the *call-site trees*, which is a nice balance between tree sizes and the representation capability. In the call-site tree, two different executions of a call-site (say α that is located within a function f) are represented by the same node if and only if the calling chains from $main$ to f in the two different executions of α are identical. We also propose analysis algorithms of the call-site tree and demonstrate its application in function inlining.

Inlining replaces a call-site with the body of the function (LLVM, 2020a, 2020b). Inlining is one of the fundamental optimisations used in a compiler[1] (Baev, 2015; Calder and Grunwald, 1994). Many other optimisation algorithms start from the results of inlining some call-sites. Advantages of inlining include

- Avoid calling overhead such as allocation of stack frames and the jump instructions.
- Enlarge the body of procedures. Larger procedures make many compiler optimisations, such as constant propagation (for arguments that are constants), copy propagation (for call-by-value arguments that are not modified in the function), dead code elimination, instruction scheduling, etc. easier and more effective.
- Inlining helps with register allocation due to avoiding jump instructions. The compiler hence can allocate registers more effectively.
- There is no need to jump to and from the function body. This helps instruction and data cache performance.
- Modern processors execute instructions faster than loading data from memory. Due to the avoidance of jumps, prefetch can be issued more effectively. The first load in a function cannot be prefetched; however, that load can be prefetched if the function is inlined.

On the other hand, inlining may cause code size explosion. This may increase page faults and cache misses.

LLVM also provides an inlining facility. The facility makes use of various heuristics in making inlining decisions. In contrast, our inliner is based on the execution counts of the call-sites (and other information about programs). When a call-site is inlined, new call-sites may be created and the execution counts of existing call-sites may change. Updating execution counts by re-profiling makes inlining very slow. Therefore, we design the call-site tree. With call-site trees, no re-profiling is needed. The characteristic of our work include:

- Profiling is performed only once. Our algorithms can automatically infer the execution counts of the new call-sites that are created due to previous inlining and adjust the counts of existing call-sites.
- We develop new analysis techniques that can estimate the possible execution counts of all call-sites and all functions. We setup equations for the execution counts. These equations for non-recursive programs can be solved with existing mathematical techniques. However, solving the equations for recursive programs requires advanced mathematical techniques. Investigating these advanced mathematical techniques is left as future work.

The rest of this paper is organised as follows: Section 2 presents the background of this work. Section 3 presents the construction of the call-site tree, the algorithm for analysing the call-site tree, and the implementation of the inliner. Section 4 shows the analysis methods that determine the size of the call-site trees and the possible execution counts of call-sites. Section 5 shows the experiment results. Section 6 concludes this paper.

2 Background

2.1 Motivation

Profile information is very useful for effective inlining (Baev, 2015). But exactly what profile information is needed? This depends on the objective of inlining. In this paper, the objective is to reduce execution time. A reasonable approach is to inline the call-sites that are executed most often. Therefore, the execution counts are the profile information used in this paper.

When a call-site is inlined, call-sites and execution counts may change. Hence, profiling may need to be performed again and again. Repeated profiling is a bad idea. In this paper, we propose the call-site tree on which execution counts may be updated without re-profiling. To collect execution counts, it is natural to keep a counter at each call-site.

Example 2.1: Apply simple profiling to the program in Figure 1. Assume the β call-site is executed once, the γ call-site is executed once, the δ call-site is executed 65 times, the ω call-site is executed twice, the ϕ call-site is executed twice, the ξ call-site is executed 16 times in a particular run. We therefore choose the most executed call-site, which is δ, to inline. It is straightforward to find the call-site with the highest counts. However, the question is to find the most executed call-site **after** inlining without profiling again. In the above example, after the δ call-site is inlined, new call-sites are created. The new program is shown in Figure 2.

We want to know how often the new call-sites (δ', ω', ϕ', and ξ') are executed with the same input data. We need more detailed trace information to answer the above question. We use the execution tree in Figure 6, which is built from the trace of an execution of the program in Figure 5. The call-site tree is a run-time expansion of the traditional call graph used in traditional compilers. A downside of the execution tree is it is too big, especially for recursive programs. In order to reduce the size of the execution tree, we may condense the execution tree by merging certain nodes.

2.2 Related works

Most languages and compilers provide some kinds of inlining. Inlining is similar to macro expansion in C (Kernighan and Ritchie, 1988; Chang et al., 1992). However, macro substitution is mandatory. In contrast, inlining is usually only a hint to clang (GCC, 1987; Clang, 2020). Inlining requests may be ignored. Inlining in C is made complicated due to the `static` declaration. On the other hand, inlining in C++ is simpler (Clang, 2020; Chang and Hwu, 1989).

There is a built-in inliner in the LLVM framework. That inliner makes use of static information of the program and various heuristics (Zhongxiao, 2023; Larin et al., 2017). In contrast, the inliner in this paper makes use of (dynamic) execution counts.

Figure 1 An example program (see online version for colours)

```
g(){ ... }
 f(){
    for (i = 1; i < m; i ++) {
      if (r) { f(); }; /* δ */
    }
    if (s) { f(); } /* ω */
    if (t) { f(); } /* ϕ */
    if (u) { g(); } /* ξ */
 }
 main(){
    if (p) { f(); } /* β */
    if (q) { f(); } /* γ */
 }
```

Figure 2 Inlined Figure 1 (see online version for colours)

```
f(){
    for (i = 1; i < m; i ++) {
      if (r) {
        for (j = 1; j < m; j ++) {
          if (r) { f(); }; /* δ′ */
        }
        if (s) { f(); } /* ω′ */
        if (t) { f(); } /* ϕ′ */
        if (u) { g(); } /* ξ′ */
      }
    }
    if (s) { f(); } /* ω */
    if (t) { f(); } /* ϕ */
    if (u) { g(); } /* ξ */
  }
```

Inlining a call-site may create new call-sites. Our inliner may further inline these new call-sites if the cost is worthwhile. On the other hand, LLVM inliner will never attempt inlining new call-sites.[2] For example, Figure 3 shows three versions of the factorial program. `foo1` (upper left part) is the original program. `foo2` (lower left part) is the program resulting from one inlining operation. `foo3` (right part) is the program resulting from two inlining operations. Note that the recursive call to `foo2`, which was created in the previous round of inlining, could be inlined with our algorithm. Note that the execution counts of `foo2` is a half of that of `foo1` and the execution counts of `foo3` is a half of that of `foo2` (our inliner works on LLVM IR, not C code. But it is easier to demonstrate the inliner with C code).

Inlining is a basic optimisation in a compiler. Compiler optimisation is closely related to the performance of software. Due to the recent advancement of artificial intelligence, AI techniques have been applied to compilation. Liu et al. (2021) makes use of machine learning techniques at the source code level in order to predict the performance of software. Mature compilation framework incorporates many phases of optimisations. The combinations of optimisations are too numerous to explore exhaustively. Finding a good sequence of optimisation has been studied by several researchers (You and Su, 2022). Most practical programs are written in a high-level language, such as C or C++. The quality, such as safety and reliability, of programs is an important issue. Software tools, such as Chen et al. (2023), examine the source code and point out the weaknesses of the software.

3 Design and implementation

Figure 4 is the overall organisation of our inliner. Our inlining system consists of the following three steps:

Step 1: generate the trace

The profiling pass inserts trace code into the program before each call-site. When the program runs, the trace code will print the name of the call-site (not the name of the function) before executing a call-site. When the call returns, it will print a '`ret`' string. In the sample trace in Figure 5, there are 23 occurrences of call-sites and 23 `ret` in this trace.

Observation: A half of the trace items are the *ret* items. The other half are the *call-site* items.

Step 2: analyse the trace

In order to analyse the trace, we first build the execution tree from the trace. Because the execution tree is quite big, we transform it to the call-site tree. We simulate the inlining operations on the call-site tree. Based on the default or a user-supplied priority formula, we may decide the call-sites that will be inlined and the order of inlining. Usually the priority formula is defined in terms of the execution counts of the call-sites. Other factors, such as function sizes, may be considered as well. Our inlining system provides an interface for the user to supply his own priority formula.

Step 2.1: build the call-site tree from the trace

Function calls and returns follow the last-in-first-out stack behaviour. In the beginning, we create a node representing the operating system calling the *main* procedure of the user program. Make this node the *current node*. Then we examine the trace items one by one.

- For a *call-site* item, create a node as a child of the current node. This new node represents an occurrence of the call-site or the function that is called at the call-site. Make the new node the current node.
- For a *ret* item, make the parent of the current node as the new current node.

When all items are processed, the execution tree is built, and the *main* node is the root of the call-site tree. For the trace in Figure 5, we will create the call-site tree shown in Figure 6. A node in the call-site tree denotes an occurrence

of a call-site. Alternatively, a node in the call-site tree may be viewed as an occurrence of the function being called at that call-site.

Figure 3 Three factorial programs (see online version for colours)

```
int fool(int n) {
  auto k1, res1;
  if (n > 0) { k1 = fool(n-1); res1 = n*k1; }
  else res1 = 1;
  return res1;
}
```

```
int foo2(int n) {
  auto k1, res1, k2, res2;
  if (n > 0) {
    { if (n-1 > 0) {
        k2 = foo2(n-2); res2 = (n-1)*k2;
      }
      else res2 = 1; // return res2;
    }
    k1 = res2; res1 = n*k1;
  }
  else res1 = 1;
  return res1;
}
```

```
int foo3(int n) {
  auto k1, res1, k2, res2, k3, res3, k4, res4;
  if (n > 0) {
    { if (n-1 > 0) {
        { if (n-2 > 0) {
            { if (n-3 > 0) { k4 = foo3(n-4);
                             res4 =(n-3)*k4; }
              else res4 = 1; // return res4;
            }
            k3 = res4; res3 = (n-2)*k3;
          }
          else res3 = 1; // return res3;
        }
        k2 = res3; res2 = (n-1)*k2;
      }
      else res2 = 1; // return res2;
    }
    k1 = res2; res1 = n*k1;
  }
  else res1 = 1;
  return res1;
}
```

Figure 4 Overall organisation of the inliner system (see online version for colours)

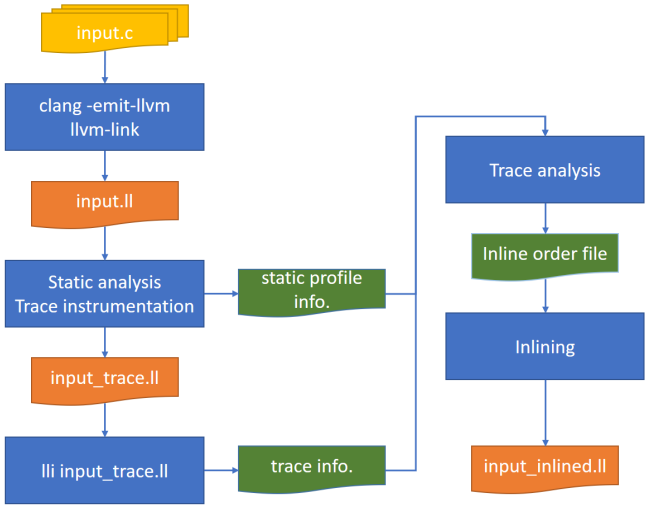


Figure 5 A sample trace produced by profiling

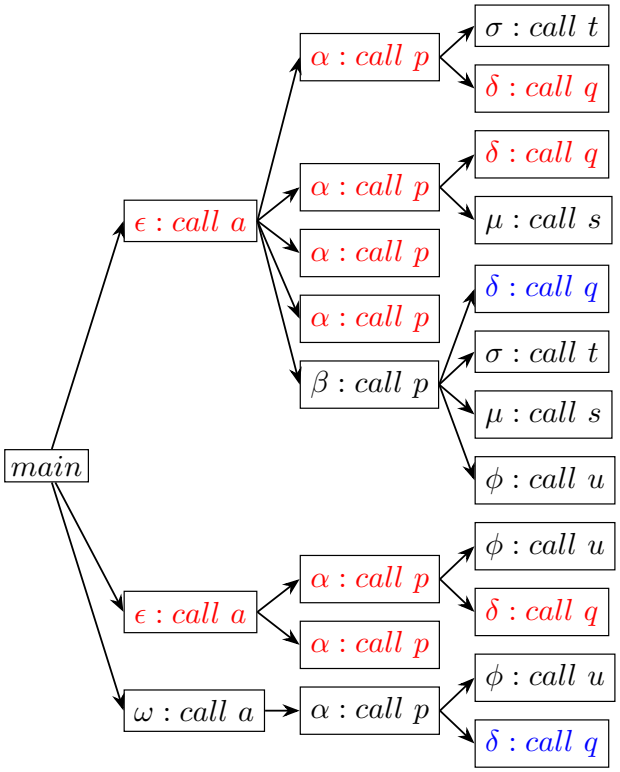
1: ϵ : *call a*	2: α : *call p*	3: σ : *call t*	4: ret	5: δ : *call q*
6: ret	7: ret	8: α : *call p*	9: δ : *call q*	10: ret
11: μ : *call s*	12: ret	13: ret	14: α : *call p*	15: ret
16: α : *call p*	17: ret	18: β : *call p*	19: δ : *call q*	20: ret
21: σ : *call t*	22: ret	23: μ : *call s*	24: ret	25: ϕ : *call u*
26: ret	27: ret	28: ret	29: ϵ : *call a*	30: α : *call p*
31: ϕ : *call u*	32: ret	33: δ : *call q*	34: ret	35: ret
36: α : *call p*	37: ret	38: ret	39: ω : *call a*	40: α : *call p*
41: ϕ : *call u*	42: ret	43: δ : *call q*	44: ret	45: ret
46: ret				

Step 2.2: condense the execution tree

The execution tree is unnecessarily large for our inlining purpose. We will merge certain nodes. We may image that each node in the execution tree carries an execution count whose value is one initially.

Starting from the *main* node, we perform a depth-first traversal. Upon visiting a node n, if n has two or more child nodes that denote the same call-site, such as the two 'ϵ : *call a*' nodes in Figure 6, these child nodes will be merged into a single call-site node. The execution count of the merged node is the sum of the execution counts of the child nodes that are merged together. The children of the original sibling nodes will become the children of the new node.

Figure 6 The execution tree (see online version for colours)



Notes: The *main* node denotes the invocation of *main* by the operating system.

After the two ϵ nodes are merged, there will be six α siblings. After the six α siblings are merged, there will be three δ siblings. The three δ siblings will be merged as well.

Note that the remaining two 'δ : *call q*' nodes cannot be merged because they are not siblings. There is no more merging. The result in Figure 7 is called the call-site tree. In Figure 7, the execution counts are added to the call-sites explicitly.

Observation: The sum of the execution counts in all the nodes in the call-site tree is equal to the number of nodes in the execution tree.

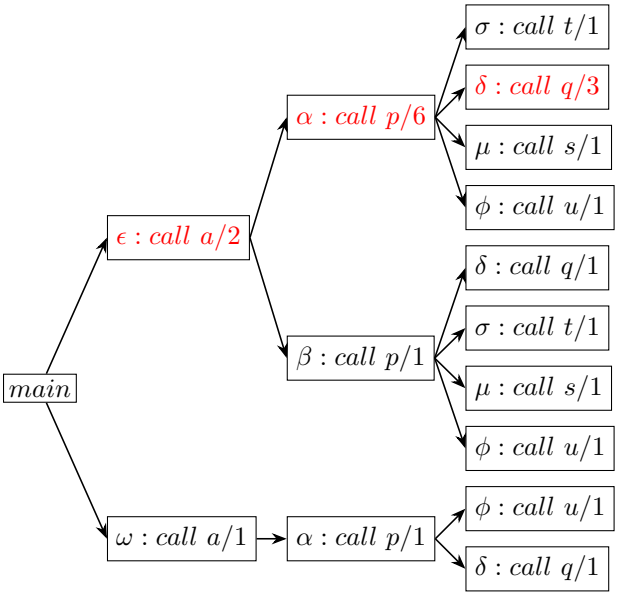
There is a significant difference between the upper limits of the sizes of the execution trees and the call-site trees. For a set of non-recursive functions, there is an upper limit on the size of the call-site tree while the execution tree could grow limitlessly.

Step 2.3: decide the call-sites that will be inlined and the order of inlining

We assign a *priority* to each call-site. The call-site with the highest priority will be inlined. After a call-site is inlined,

new call-sites may be created, and the priorities of some existing call-sites may change.

Figure 7 The tree resulting from merging the two ϵ, three δ, and six α call-site nodes (see online version for colours)



Notes: Since the remaining nodes cannot be merged, this tree is called the call-site tree. Note that an execution count is attached to each node.

The call-site tree makes it easy to find the call-site with the highest priority *after* a call-site is inlined. There is no need to re-profile the user program. The priority of a call-site is usually based on the *execution count* of the call-site because it is reasonable to inline the call-sites that are executed frequently. The execution count of a call-site is the sum of the execution counts of all occurrences of the call-site in the tree. The (estimated) *size of the function* may also be considered in deciding the priority. A user may supply his own priority formula that will be loaded into the inlining system. User-supplied priority formula are implemented as a dynamic linking library that are accessed with the dlopen call (Linux, 2021). Examples of the priority formula include $priority =_{def} ExecutionCount$, $priority =_{def} \frac{ExecutionCount}{FunctionSize}$, etc. A user may also specify the number of call-sites to be inlined in the library.

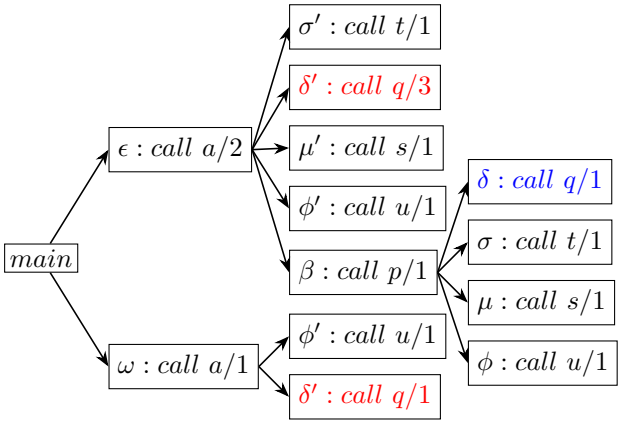
Note that after a call-site is inlined, new call-sites may be created and the execution counts of certain call-sites may change. The size of the function that contains the inlined call-site may also grow.

After a call-site is inlined, our algorithm will modify the call-site tree and compute these changes on the call-site tree. The one with the highest priority is selected as the candidate of inlining in the next round.

We may simulate the inlining operation on the call-site tree. Assume a call-site '$\alpha : call\ p$' is inlined. For every occurrence x of the α call-site in the call-site tree, we remove the node x and all children of x will become the children of x's parent and these call-sites must be given new names, such as σ', δ', μ', or ϕ' in Figure 8.

For example, there are two occurrences of the α call-site in Figure 7. After the α call-sites is inlined, the call-site tree is shown in Figure 8.

Figure 8 The call-site tree after inlining the '$\alpha : call\ p$' call-site in procedure a in Figure 7 (see online version for colours)



We repeat the above process for the next call-site to be inlined or until the (default or user-specified) cut-off condition is met. The output of this 2nd step is an ordered list of call-sites according to the computed priorities.

The inlining algorithm takes time proportional to the number of nodes in the call-site tree since it performs a depth-first traversal of the tree.

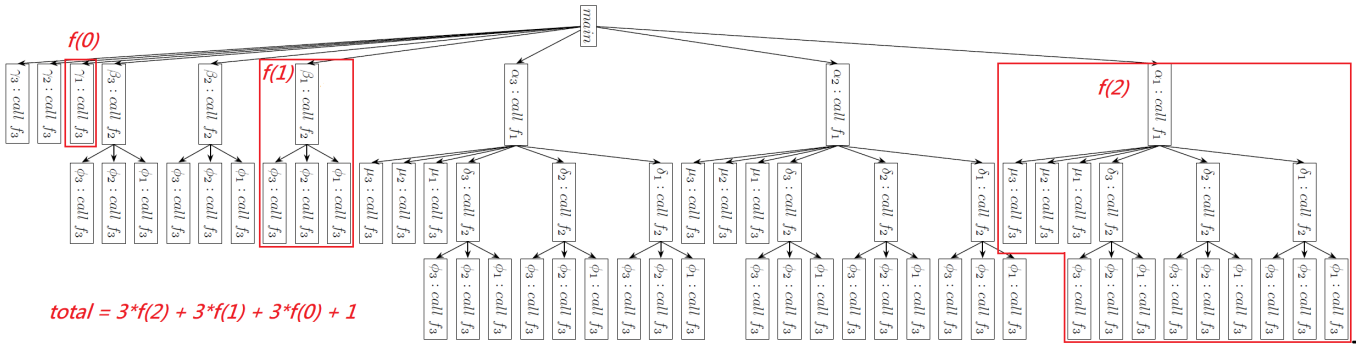
Algorithm for finding the call-sites to be inlined

begin
traverse the call-site tree and build a linked list for each call-site;
for each linked list **do**
sum up nodes' execution counts in this linked list;
compute priority of the call-site for the linked list;
end;
repeat
select the call-site, say α, with the highest priority;
modify the call-site tree by deleting all occurrences x of the α call-site and attaching x's children to x's parent;
rename the call-sites that have been moved and create the associated linked lists for the renamed call-sites;
calculate the priorities of these new call-sites;
modify the priorities of existing call-sites;
until the cut-off condition is met;
end;

Step 3: Inline the user program with an inlining pass in LLVM

The actual inlining is done on the LLVM IR with an LLVM pass. This pass modifies the user program (in the LLVM IR format) by inlining the call-sites selected in Step 2.

Figure 9 The largest call-site tree with $main$ and three other functions and no recursive calls (see online version for colours)



4 Algebraic properties of the call-site trees

In this section, we will discuss some algebraic properties of the call-site trees. Subsection 4.1 discusses the expansion property. Subsection 4.2 calculates the size of the call-site tree. Subsection 4.3 presents a method for finding the possible execution counts of call-sites and functions.

4.1 The expansion property

Note that every node in the call-site tree denotes a call-site. Consider an edge from node n_1 to node n_2 in the call-site tree. Let n_1 denotes the call-site '$\alpha : call\ p$' and n_2 denotes the call-site '$\beta : call\ q$'. Due to the construction of the call-site tree from the trace, the call-site 'β' must be inside the procedure p. Therefore, we may consider node n_1 as an instance of procedure p, node n_2 as an instance of procedure q, and the edge $n_1 \rightarrow n_2$ as an invocation $p\ calls\ q$. We can write this in a formal way.

We may define a homomorphism h from every call-site tree to the call graph of the user program as follows:

Every node n in the call-site tree may be written as an instance of a call-site '$\alpha : call\ p$'. Define $h(n) = p$. We can verify that, for every edge $n_1 \rightarrow n_2$ in the call-site tree, $h(n_1 \rightarrow n_2) = h(n_1) \rightarrow h(n_2)$ (note that $h(n_1) \rightarrow h(n_2)$ is an edge in the call graph).

The existence of the homomorphism between a call-site tree and the call graph is called the *expansion property*.

4.2 How large is the call-site tree?

Consider a program that consists of $main$, f_1, f_2, ..., f_n functions. If there are deep (direct or indirect) recursive functions, the call-site tree could be arbitrarily big. So assume there are no (direct or indirect) recursive functions. In this case the call-site tree can still be as big as you wish. For example, consider function f_1 calls function f_2 1,000 times (that is, there are 1,000 call-sites in the program). The call-site tree could contain 1,000 or more nodes. It is still quite large. Therefore, we further assume that there is a constant k such that any function can call any other functions at most k times (i.e., at most k call-sites).

Under these restrictions, how large could a call-site tree be?

We can find there are repetitions of patterns in the call-site trees. Consider an example in which there are four functions: $main$, f_1, f_2, and f_3. Assume there are no recursive functions and each function can call any other function at most k (k = 3 in this example) times. The largest call-site tree for this example is shown in Figure 9:

1 there are k copies of the largest call-site tree with three functions f_1, f_2, and f_3

2 there are k copies of the largest call-site tree with two functions f_2, and f_3

3 there are k copies of the largest call-site tree with one function f_3.

We may setup the following equations: Let $f(n)$ denote the size of the largest call-site tree with $main$ and n other functions. Then

$$\begin{aligned} f(0) &= 1\ (only\ the\ main\ function) \\ f(n) &= kf(n-1) + kf(n-2) + kf(n-3) + ... \\ &\quad + kf(1) + kf(0) + 1 \end{aligned}$$

The solution of the the above equation is $f(n) = (k+1)^n$.

4.3 Estimating the set of possible execution counts of a call-site

In this section, we will estimate the possible execution counts of each call-site. It is obviously impossible to decide the exact execution count of a call-site in general. However, we develop a new analysis method that can determinate the set of possible execution counts of each call-site more exactly.

A trivial and useless estimate of the set of possible execution counts of a call-site is {0, 1, 2, ...} (i.e., all non-negative integers). However, our method can find more exact estimates than this trivial one.

Specifically, let α be a call-site located within a function f. We define $\aleph(\alpha)$, $\mu(f)$, and $\sigma_f[\alpha]$ as follows.

1 Let $\aleph(\alpha)$ be the set of possible execution counts of α when the whole program is executed once.

2 Let $\mu(f)$ be the set of possible execution counts of function f when the whole program is executed once.

3 Let $\sigma_f[\alpha]$ be the set of possible execution counts of α when function f is executed once.

There could be different execution counts for the same call-site in different runs of the program. Furthermore, in theory, it is impossible to determine the exact execution counts of a call-site without executing the program. Therefore, the possible execution counts of a call-site is a set of non-negative integers.

In this section, we perform the following two steps to estimate the possible execution counts $\aleph(\alpha)$ for all call-sites.

1 For every call-site α (let f be the function containing α), we first calculate $\sigma_f[\alpha]$ in a syntax-directed computation based on the syntax of the *shrinked function* f_α (which will be defined later).

2 Then we will compute $\mu(f)$ from various $\sigma_g[\beta]$ and compute $\aleph(\alpha)$ from $\mu(f)$ and $\sigma_f[\alpha]$.

In this section we will use the following operations. Let s and t be (possibly empty) sets of non-negative integers.

- Define $s \oplus t =_{def} \{a + b \mid a \in s \wedge b \in t\}$.
- Define $s^0 =_{def} \{0\}$. $s^1 =_{def} s$.
- Define $s^h =_{def} s \oplus s \oplus ... \oplus s \oplus s = s \oplus s^{h-1}$ (for $h > 1$).
- Define $s^* =_{def} \{0\} \cup s \cup s \oplus s \cup s \oplus s \oplus s \cup ... = s^0 \cup s^1 \cup s^2 \cup ...$ (closure).

Figure 10 The shrinked function f_δ for the example in Figure 1

```
  f(){
S1:   for (i = 1; i < m; i ++) {
S2:     if (r) {
              f(); }; /* δ */
        }
    }
```

Notes: Note that the statement labels $S1$ and $S2$ are added for easy reference only. From the shrinked function f_δ, we wish to compute $\sigma_\delta[\delta]$, $\sigma_{S2}[\delta]$, $\sigma_{S1}[\delta]$, $\sigma_f[\delta]$ in a syntax-directed manner.

The following properties of $\oplus$ are obvious:

1 $s \oplus t = t \oplus s$

2 $s \oplus \emptyset = \emptyset$

3 $s \oplus \{0\} = s$

4 $s \oplus (t \cup u) = (s \oplus t) \cup (s \oplus u)$

5 $s \oplus (t \cap u) \subseteq (s \oplus t) \cap (s \oplus u)$

6 if $|s| = 1$ then $|s \oplus t| = |t|$.

For each call-site α located inside function f, we first create the corresponding *shrinked function* f_α by removing all declarations and statements in f except those that enclose the call-site α. For example, the shrinked function f_δ for the δ call-site in the example in Figure 1 is shown in Figure 10.

From the shrinked function f_δ, we wish to compute $\sigma_f[\delta]$ with the help of $\sigma_\delta[\delta]$, $\sigma_{S2}[\delta]$, and $\sigma_{S1}[\delta]$ (each of which is a set of non-negative integers) in a syntax-directed manner.

$\sigma_f[\delta]$ is the set of possible execution counts of the call-site δ if function f is executed exactly once. Similarly, $\sigma_{S1}[\delta]$ is the set of possible execution counts of the call-site δ if the for-loop $S1$ is executed exactly once. $\sigma_{S2}[\delta]$ is the set of possible execution counts of the call-site δ if the if-statement $S2$ is executed exactly once.

Of course, $\sigma_\delta[\delta] = \{1\}$.

We consider only a simplified language which contains block statements, call-statements, if-statements, for-loops, and while-loops. There are no goto nor exceptions. We want to estimate the possible execution counts of each call-site.

1 **S ::= aStatement**

$\sigma_S[\alpha] =_{def} \{0\}$ if the call-site α is not located within the statement S.

2 **S ::= α: call foo()**

$\sigma_S[\alpha] =_{def} \{1\}$. Here S is the call-site α.

3 **S ::= begin T end**

$\sigma_S[\alpha] =_{def} \sigma_T[\alpha]$

4 **S ::= if Q then T else skip**

$\sigma_S[\alpha] =_{def} \sigma_T[\alpha]$ if the predicate Q is true according to constant folding,

else $\{0\}$ if the predicate Q is false according to constant folding,

else $\sigma_T[\alpha] \cup \{0\}$

5 **S ::= if Q then skip else T**

Similar to **if Q then T** but in opposite way.

6 **S ::= for (i = e1; i < e2; i = i + e3) begin T end**

$\sigma_S[\alpha] =_{def}$

$\sigma_T[\alpha] \oplus \sigma_T[\alpha] \oplus \sigma_T[\alpha] \oplus ... \oplus \sigma_T[\alpha] \equiv \sigma_T[\alpha]^k$

if the compiler can determine the number of iterations is k with constant folding,

else $\{0\} \cup \sigma_T[\alpha] \cup \sigma_T[\alpha] \oplus \sigma_T[\alpha] \cup \sigma_T[\alpha] \oplus \sigma_T[\alpha] \oplus \sigma_T[\alpha] \cup ... \equiv \sigma_T[\alpha]^*$

7 **S ::= while (Q) begin T end**

Similar to the above for-loops.

8 **S ::= function ID (P) begin T end**

$\sigma_S[\alpha] =_{def} \sigma_T[\alpha]$; we use $\sigma_{ID}[\alpha]$ to denote the set $\sigma_S[\alpha]$ in this case.

According to the above syntax-directed calculation on the shrinked function, we can determine the set of possible

execution counts of a call-site located inside a function assuming the function is executed exactly once.

Suppose the whole program contains functions $main$, f_1, f_2, ..., f_k. For each function f we define $\mu(f)$ as the set of possible execution counts of function f. Because the $main$ function is invoked exactly once, we have

$$\mu(main) = \{1\} \tag{1}$$

The possible execution counts of a call-site α located inside function f is

$$\aleph(\alpha) =_{def} \bigcup\{\sigma_f[\alpha]^h | h \in \mu(f)\} \tag{2}$$

Furthermore, we can setup an equation for each $\mu(f)$ as follows: let β_1, β_2, ..., β_p be all the call-sites that call function f. Then

$$\mu(f) = \aleph(\beta_1) \oplus \aleph(\beta_2) \oplus ... \oplus \aleph(\beta_p) \tag{3}$$

Finally we may attempt to solve the above simultaneous equations (1), (2) and (3).

In summary, we wish to find $\aleph(\alpha)$ and $\mu(f)$ from various $\sigma_f[\beta]$, for each call-site α and each function f in the program.

Example 4.1: Consider the example program in Figure 11. The possible execution counts of the δ call-site are calculated as follows:

$\sigma_{S4}[\delta] = \{1\}$.
$\sigma_{S3}[\delta] = \sigma_{S4}[\delta] \oplus \sigma_{S4}[\delta] \oplus \sigma_{S4}[\delta] = \{3\}$.
$\sigma_{S2}[\delta] = \{0\} \cup \sigma_{S3}[\delta] = \{0, 3\}$.
$\sigma_f[\delta] =_{def} \sigma_{S2}[\delta] = \{0, 3\}$.

Note that the ω call-site will not affect $\sigma_f[\delta]$.

The possible execution counts of the ω call-site are calculated as follows:

$\sigma_{S6}[\omega] = \{1\}$.
$\sigma_{S5}[\omega] = \{0\} \cup \sigma_{S6}[\omega] = \{0, 1\}$.
$\sigma_f[\omega] =_{def} \sigma_{S5}[\omega] = \{0, 1\}$.

The possible execution counts of the ζ call-site are calculated as follows:

$\sigma_{Sa}[\zeta] = \{1\}$.
$\sigma_{S9}[\zeta] = \{0\} \cup \sigma_{Sa}[\zeta] = \{0, 1\}$.
$\sigma_{S8}[\zeta] = \sigma_{S9}[\zeta] \oplus \sigma_{S9}[\zeta] = \{0, 1, 2\}$.
$\sigma_{main}[\zeta] =_{def} \sigma_{S8}[\zeta] = \{0, 1, 2\}$.

Next we may setup the following equations:

$$\mu(main) = \{1\} \tag{4}$$
$$\mu(f) = \aleph(\zeta) \tag{5}$$
$$\mu(g) = \aleph(\delta) \oplus \aleph(\omega) \tag{6}$$
$$\aleph(\delta) =_{def} \bigcup\{\sigma_f[\delta]^h | h \in \mu(f)\} \tag{7}$$
$$\aleph(\omega) =_{def} \bigcup\{\sigma_f[\omega]^h | h \in \mu(f)\} \tag{8}$$
$$\aleph(\zeta) =_{def} \bigcup\{\sigma_{main}[\zeta]^h | h \in \mu(main)\} \tag{9}$$

Figure 11 Program for Example 4.1

```
S1: f(){
S2:   if (r)
S3:     for (i = 1; i < 4; i ++)
S4:       g();   /* δ */
S5:   if (s)
S6:     g();   /* ω */
    }
    g() { ... }
S7: main(){
S8:   for (j = 1; j < 3; j ++)
S9:     if (t)
Sa:       f();   /* ζ */
    }
```

Therefore,

$\mu(main) = \{1\}$.
$\aleph(\zeta) = \sigma_{main}[\zeta]^1$ $(because\ \mu(main) = \{1\})$
$= \{0, 1, 2\}$.
$\mu(f) = \aleph(\zeta) = \{0, 1, 2\}$.
$\aleph(\delta) = \sigma_f[\delta]^0 \cup \sigma_f[\delta]^1 \cup \sigma_f[\delta]^2 = \{0\} \cup \{0, 3\}$
$\cup\{0, 3, 6\} = \{0, 3, 6\}$.
$\aleph(\omega) = \sigma_f[\omega]^0 \cup \sigma_f[\omega]^1 \cup \sigma_f[\omega]^2 = \{0\} \cup \{0, 1\}$
$\cup\{0, 1, 2\} = \{0, 1, 2\}$.
$\mu(g) = \aleph(\delta) \oplus \aleph(\omega) = \{0, 3, 6\} \oplus \{0, 1, 2\}$
$= \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$.

In the above example,

$$\mu(g) = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$$

This means that, if the $main$ procedure is executed once, the g procedure may be executed 0, 1, 2, 3, 4, 5, 6, 7, or 8 times. There are no other possibilities.

Similarly,

$$\aleph(\delta) = \{0, 3, 6\}$$

this means that, if the $main$ procedure is executed once, the δ call-site may be executed 0, 3, or 6 times. There are no other possibilities.

Examples 4.1 shows only programs without recursive calls. In this case, the equations can be solved easily.

For programs with recursive calls, the equations become recursive. We could use the iterative solution: starting from the initial condition, we repeatedly calculate the values of various variables until a stable solution is found. A closed-form solution might need advanced mathematical techniques. We will leave its solution as future work.

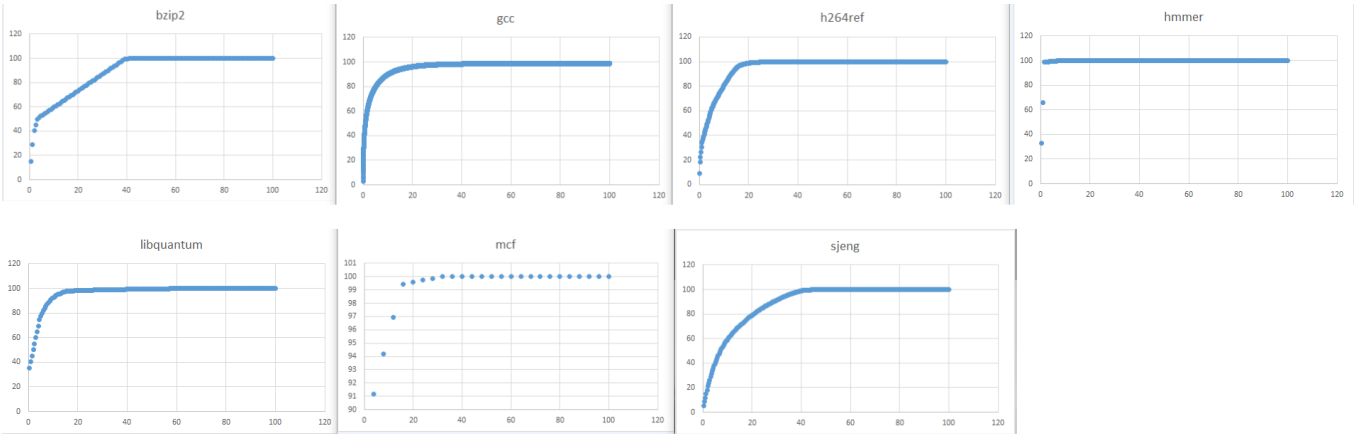
For example, suppose a function f contains a call-site α that calls f itself. Then equation (2) will become

$$\aleph(\alpha) =_{def} \bigcup\{\sigma_f[\alpha]^h | h \in \mu(f)\} \tag{10}$$

and equation (3) will become

$$\mu(f) = \aleph(\alpha) \oplus ... \tag{11}$$

Figure 12 Percentage of call-sites (horizontal axis) vs. percentage of invocations (vertical axis) (see online version for colours)



These are recursive equations involving sets of non-negative integers. Their solutions will be left as future work.

Note $\sigma_S[\alpha] \subseteq \{0\} \cup N$ because every call-site α may be executed 0, 1, 2, ... times. We wish to find a more accurate (i.e., smaller) estimate for $\sigma_S[\alpha]$. If better constant propagation can infer values of expressions more accurately during program analysis, it is possible to find more accurate estimate to each $\sigma_S[\alpha]$. Pursuing in this direction requires excellent constant propagation capability. We will leave it as future study.

Additionally, a programmer can specify the 'range of the value using metadata' (LLVM, 2023; Finkel, 2016; Lattner, 2010). With this range metadata we can compute $\aleph(\alpha)$ more precisely.

Table 1 The total number of call-sites in a benchmark (*#call-sites*) and the total number of invocations of the functions in the benchmark (*#invocations*)

	#call-sites	*#invocations*
bzip2	146	590,645
gcc	13,185	90,167,491
h264ref	549	1,055,971,530
hmmer	190	5,242,553
libquantum	197	639,878
mcf	25	87,399,944
sjen	286	143,196,950

5 Experimental results

We implemented the whole inlining system. In the experiment environment, a PC with Intel(R) Core(TM) i5-7400 CPU @ 3.00 GHz (x86-64) with 16 GB memory, running Ubuntu 20.04, LLVM 10.0, and clang in LLVM 10.0, is used. The seven benchmarks come from CINT2006 in SPEC CPU 2006 (Henning, 2006).

Execution counts of call-sites vary significantly. A very large percentage of the total invocations are due to a small percentage of the call-sites. Figure 12 shows the percentage of call-sites (horizontal axis) and the percentage of invocations (vertical axis). Figure 12 shows that for all benchmarks, roughly 80% of the invocations are due to 20% of the call-sites.

We perform three analyses of the performance of the call-site tree technique:

1 we measure the number of invocations that are eliminated by the call-site-tree technique

2 we compare the numbers of eliminated invocations with the Naïve inliner and with the call-site tree technique

3 we compare the running time of the call-site-tree inliner and the LLVM inliner.

5.1 The effect of the call-site-tree technique compared with no inlining

One characteristic of the call-site-tree technique is that call-sites that are created by previous inlining operations may be inlined again as long as the new call sites have high execution counts. We define *NCS* as the new call-sites that are created due to previous inlining operations and are inlined again.

Table 2 Column f is the ratio of the execution counts of *NCS* (column c) divided by the total number of invocations of the whole program (column e) and column g is the total number of invocations of the 56 inlined call-sites (column d) divided by column e

Benchmark	*b*	*c*	*d*	*e*	*f*	*g*
bzip2	0	0	580,520	590,645	0%	98.29%
gcc	11	3,727,636	32,112,216	90,167,491	4.13%	35.61%
h264ref	6	83,538,159	856,688,938	1,055,971,530	7.91%	81.13%
hmmer	8	1,630	5,241,992	5,242,553	0.03%	99.99%
libquantum	10	32,459	631,560	639,878	5.07%	98.70%
mcf	49	2,194,645	87,339,582	87,399,944	2.51%	99.93%
sjeng	15	12,429,470	102,019,934	143,196,950	8.68%	71.24%

First it is obvious that the more call-sites are inlined, the more invocations are eliminated. We may inline all call-sites repeatedly until there is no invocation left at

run time. Such an approach is quite useless. For this experiment, we will inline 56 call-sites that have the highest execution counts using the call-site-tree technique. The result is shown in Table 2.

In Table 2, the leftmost column lists the seven benchmarks.

Column b lists the number of *NCS* among the 56 inlined call-sites. Note that LLVM inliner will not inline *NCS*. Instead it will inliner other call-sites with few execution counts.

Column c lists the number of invocations due to *NCS* among the 56 inlined call-sites.

Column d lists the number of invocations due to the 56 inlined call-sites.

Column e lists the number of invocations of the whole program in the profile run.

Column f lists the ratios of column c divided by column e. Column f should be compared with column b divided by 56. For example, consider the *gcc* benchmark. *NCS* = 11. However, 11/56 = 19.64% which is larger than 4.13%. This observation, that is, column c/56 > column f, holds for all benchmarks. It means that the first few non-*NCS* call-sites among the 56 inlined call-sites account for a large portion of the invocations of the 56 inlined call-sites.

Column g lists the ratios of column d divided by column e. Column g, which ranges from 35% to 99%, shows the percentage of invocations that will be eliminated if the 56 call-sites are inlined. Since all values in column g are quite large, inlining 56 call-sites is sufficient to draw a reliable conclusion because, according to our experiment, each of the remaining call-sites only account for less than 0.28% of the total invocations and real inliners probably will not inline them.

5.2 Comparison with the Naïve dynamic inlining technique

The call-site-tree technique will inline *NCS* (as long as their execution counts are high enough) while LLVM inliner, which is a static tool, will never attempt to inline *NCS*. We will examine the benefit of this feature.

Table 3 The percentage of invocations eliminated with the call-site-tree technique

Benchmark	*b*	*c*	*d*	*e*	*f*	*g*
bzip2	492,072	492,072	0%	0	0	0%
gcc	27,156,150	27,713,772	2.05%	6	2,290,964	8.44%
h264ref	706,027,579	749,291,818	6.13%	6	83,538,159	11.83%
hmmer	5,259,303	5,260,783	0.03%	8	1,630	0.03%
libquantum	597,755	628,354	5.12%	1	30,925	5.17%
mcf	85,144,937	87,296,109	2.53%	33	2,151,172	2.53%
sjen	89,293,276	96,195,583	7.73%	6	8,778,036	9.83%
average			3.37%			5.40%

To measure the effect of the call-site-tree technique, we distinguish two levels of the effect: one is the direct elimination of the invocations due to the call-site-tree technique; the other is the reduction of execution time due to whatever optimisations the compiler applies to the benchmarks together with inlining. These optimisations will dilute the effect of the inlining operations. The latter effect is significantly affected by many other factors, such as whether the benchmarks admit other optimisations.

Table 4 Improvements when 4–56 call-sites are inlined

Improvements	*bzip*	*gcc*	*h264ref*	*hmmer*	*libquantum*	*mcf*	*sjeng*
i4	1.0101	1.0011	1.0027	1.0000	1.0030	1.0419	1.0025
i8	1.0018	1.0058	1.0041	0.9979	1.0029	1.0204	1.0041
i12	1.0027	1.0058	1.0027	0.9889	0.9656	1.0225	0.9982
i16	1.0070	1.0045	1.0121	0.9929	0.9680	1.0355	0.9964
i20	1.0083	1.0066	1.0150	1.0001	0.9694	1.0269	0.9878
i24	1.0094	1.0089	1.0085	1.0058	0.9688	1.0278	1.0028
i28	1.0101	1.0077	1.0119	0.9837	1.0167	1.0237	1.0014
i32	1.0080	1.0078	1.0088	0.9835	1.0106	1.0253	0.9832
i36	1.0076	1.0078	1.0079	0.9931	1.0116	1.0232	0.9326
i40	1.0101	1.0073	1.0036	1.0303	0.9924	1.0205	0.9324
i44	1.0112	1.0062	1.0057	1.0285	0.9942	1.0248	0.8721
i48	1.0094	1.0071	1.0115	1.0332	0.9972	1.0222	0.8675
i52	1.0074	1.0066	1.0132	1.0260	0.9957	1.0208	0.8739
i56	1.0074	1.0059	1.0109	1.0296	0.9927	1.0202	0.8451
i4f	0.9865	1.0044	1.0032	0.9993	1.0014	1.0311	1.0001
i8f	0.9915	1.0033	1.0045	1.0033	0.9656	1.0261	1.0003
i12f	0.9889	1.0077	1.0031	1.0018	0.9665	1.0223	0.9863
i16f	1.0058	1.0002	1.0059	1.0102	0.9961	0.9911	0.9934
i20f	1.0055	1.0004	1.0131	1.0148	0.9961	0.9872	0.9956
i24f	1.0063	0.9947	0.9747	0.9987	0.9919	0.9844	0.9895
i28f	1.0093	0.9954	0.9825	1.0034	0.9934	0.9939	0.9956
i32f	1.0056	0.9997	1.0140	1.0120	0.9948	0.9843	0.9949
i36f	1.0058	0.9985	1.0136	1.0087	0.9925	0.9960	0.9935
i40f	1.0055	0.9964	1.0130	1.0132	0.9916	0.9919	0.9923
i44f	1.0078	0.9952	1.0127	1.0160	0.9946	0.9932	0.9929
i48f	1.0061	0.9897	1.0139	1.0117	0.9925	0.9871	0.9957
i52f	1.0088	0.9928	1.0028	1.0118	0.9889	0.9911	0.9884
i56f	1.0087	0.9940	1.0056	1.0151	1.0004	1.0111	1.0014

We will compare the call-site-tree technique with the Naïve inlining technique. The Naïve inlining technique selects the call-sites to be inlined purely based on the execution counts. The Naïve technique will ignore *NCS* when selecting call-sites to be inlined.

We show the number of invocations that are eliminated. For each benchmark, we select 40 call-sites (that have the highest execution counts) using the two techniques. The result is shonw in Table 3. Column b is the sum of the execution counts of the 40 call-sites (no *NCS* included) that are selected by the Naïve technique in the respective benchmark. This number is the number of invocations that will be eliminated if the 40 call-sites are inlined. Column c is the sum of the execution counts of the 40 call-sites that are selected the call-site-tree technique. Column d is the percentage of the invocations that are saved with the call-site technique, that is $(c - b)/b * 100\%$. The average of d is 3.37%.

Column e is the number of *NCS* among the 40 call-sites in the respective benchmark. This number could be as small as 0 (the *bzip2* benchmark) and as large as 33 (the *mcf* benchmark).

Table 5 The number of call-sites and the average performance improvements at 70%, 80%, 90% invocation coverage (see online version for colours)

Dataset	*70% invocations*		*80% invocations*		*90% invocations*	
	#call sites	*perf improvement*	*#call sites*	*perf improvement*	*#call sites*	*perf improvement*
bzip2	26	1.0101	37	1.0094	48	1.0080
gcc	355	0.9850	635	0.9693	1,223	0.9400
h264ref	39	1.0042	55	1.0121	72	1.0105
hmmer	3	1.0011	3	1.0009	3	1.0024
libquantum	8	1.0026	11	1.0012	18	1.0087
mcf	1	1.0336	1	1.0360	1	1.0301
sjeng	43	0.8728	60	0.8429	83	0.8344

For *bzip2*, the same 40 call-sites are inlined in both techniques. There are no *NCS* in the 40 call-sites. Hence, there is no saving if we inlined only 40 call-sites. This means that the call-site-tree technique did not inline any *NCS*. On the other hand, for *mcf*, 33 call-sites among the 40 highest-count call-sites are *NCS*. This is because in *mcf*, the first call-site accounts for 91% of all invocations, then 8 more call-sites account for almost all the remaining invocations. The remaining call-sites are executed only once or twice. In the *hmmer* and *sjeng* benchmarks there are one or two call-sites that are created by inlining another call-site (call it δ) and the δ call-site itself is created by inlining yet another call-site. This shows that the call-site-tree technique is very good at discovering call-sites with high execution counts.

Column f in Table 3 is the sum of the execution counts of *NCS*. Column g, which is defined as f/b, is the ratio of the execution counts of *NCS* and that of all the 40 inlined call-sites that have the highest execution counts. The average of g is 5.40%. This average ratio can be roughly interpreted as the call-site-tree technique requires 5.40% fewer invocations than the Naïve technique.

5.3 Comparison with the static LLVM inliner

In our experiment, 4–56 call-sites are inlined. The experimental configurations are named as i4, i4f, i8, i8f, ..., i56, and 56f, where the number (e.g., 4) denotes the number of call-sites that are inlined. A configuration name without the f suffix implies that the priority of a call-site is simply the execution count of the call-site. A configuration name with the f suffix implies that the priority of a call-site is the execution count of the call-site divided by the number of LLVM IR instructions in the function that is inlined. After inlining is performed, clang compiles the benchmarks with the –O2 option. The benchmarks come with 1–11 test datasets.

Table 4 shows the average performance improvements of the benchmark when 4–56 call-sites that have the highest execution counts are inlined.

Because inlining is always used with other optimisations, we really want to compare the execution time of case 1 (inlining is disabled but all other optimisations are enabled) and case 2 (enable our inlining and all all other optimisations). However, when other optimisations implied by the –O2 option are enabled, they will automatically perform inlining in LLVM. Therefore, the performance improvements shown in Table 4 actually is the difference between

1 our inliner + LLVM inliner + O2 optimisations

2 LLVM inliner + O2 optimisations.

The performance of our inliner for *bzip*, *gcc*, *h264ref*, and *mcf* is always improved, though slightly (see i4, i8, i12, ..., i56). For other benchmarks, the performance does not differ much. The reason might be that the priority formula we use is quite simple. In contrast, LLVM inliner uses sophisticated cost estimation. Alternatively, we may say that with profile information, a simple priority formula achieves similar or better effect than the LLVM inliner, which is equipped with mature, sophisticated cost estimation but without profile information.

Inlining brings both positive and negative effects to performance. When more than necessary inlining is performed, performance might degrade.

Consider the *gcc* benchmark. If only 4, 8, or 12 call-sites are inlined, the performance improvement is greater than 1. When 355 or more call-sites (equivalently, 2.7% of call-sites or 70% or more coverage of invocations), the performance improvement is less than 1 (see Table 5). The reason is that when more than necessary call-sites are inlined, the code size explosion reduces the performance.

Consider the *sjeng* benchmark. If only 4, 8, 24, or 28 call-sites are inlined, the performance improvement is greater than 1. If 32 or more call-sites are inlined, the performance improvement is less than 1 (see Table 4).

For the remaining five benchmarks (*bzip2*, *h264ref*, *hmmer*, *libquantum*, and *mcf*), our inliner always achieves positive performance improvements when 70% or more invocations are covered. This is shown in Table 5.

6 Conclusions

In order to capture the function-invocation behaviour of a program, we designed a new data structure, the call-site tree, which is a nice balance between the static call graph and the dynamic execution tree. In the call-site tree, two different executions of a call-site (say α that is located

within a function f) are represented by the same node if and only if the calling chains from $main$ to f in the two different executions of α are identical. We also proposed methods for analysing the call-site tree, and demonstrated its application in function inlining. Our inliner is profile-guided, customisable, incremental and is based on LLVM IR. Experiment results for SPEC INT 2006 are also included. The call-site tree is an expansion of the traditional call graph. We proposed new analysis methods that can calculate the upper limit of the size of the call-site trees for non-recursive programs and the possible execution counts of all call-sites. Our inlining system and analysis methods can be improved with more accurate constant propagation techniques. The analysis methods need more advanced mathematical techniques in order to solve the equations for recursive programs.

Acknowledgements

The work reported in this paper is partially supported by National Science Council, Taiwan, Republic of China, under grants MOST 108-2221-E-009-050-MY3 and MOST 111-2221-E-A49-111-MY3.

References

Baev, I. (2015) *Profile-based Indirect Call Promotion*, Technical Report, Qualcomm Innovation Center, Inc., October.

Chen, C-Y., Fang, Y-A., Wang, G-R. and Chen, P-S. (2023) 'A GCC-based checker for compliance with MISRA-C's single-translation-unit rules', *Connection Science*, Vol. 35, No. 1, pp.1–4.

Calder, B. and Grunwald, D. (1994) 'Reducing indirect function call overhead in C++ programs', in *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*, pp.397–408.

Chang, P.P. and Hwu, W-M.W. (1989) 'Inline function expansion for compiling C programs', in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation – PLDI '89*, ACM Press, New York, New York, USA, pp.246–257.

Clang (2020) *INLINING, Clang 16.0.0git Documentation, Section 3.2*.

Chang, P.P., Mahlke, S.A., Chen, W.Y. and Hwu, W-M.W. (1992) 'Profile-guided automatic inline expansion for C programs', *Software: Practice and Experience*, Vol. 22, No. 5, pp.349–369.

Finkel, H. (2016) *Intrinsics, Metadata, and Attributes: The Story Continues!*, Technical Report, Argonne National Laboratory, November.

GCC (1987) *GCC, The GNU Compiler Collection*.

Henning, J.L. (2006) 'SPEC CPU2006 benchmark descriptions', *Comput. Archit. News*, September, Vol. 34, No. 4, pp.1–17.

Kernighan, B.W. and Ritchie, D.M. (1988) *C Programming Language*, 2nd ed., Prentice Hall.

Li, X., Ashok, R. and Hundt, R. (2010) 'Lightweight feedback-directed cross-module optimization', in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, New York, New York, USA, pp.53–61.

Lattner, C. (2010) *Extensible Metadata in LLVM IR*, April [online] https://blog.llvm.org/2010/04/extensible-metadata-in-llvm-ir.html.

Liu, W., Hu, E-W., Su, B. and Wang, J. (2021) 'Using machine learning techniques for DSP software performance prediction at source code level', *Connection Science*, Vol. 33, No. 1, pp.26–41.

Linux (2021) *dlopen(3) – Linux Manual Page*.

Larin, S., Jagasia, H. and von Koch, T.E. (2017) *Impact of the Current LLVM Inlining Strategy*, Technical report, Qualcomm Innovation Center, Inc., February.

LLVM (2020a) *The LLVM Compiler Infrastructure*.

LLVM (2020b) *LLVM Language Reference Manual – LLVM 10 Documentation*.

LLVM (2023) *LLVM Range Metadata* [online] https://llvm.org/docs/LangRef.html.

Ryder, B.G. (1979) 'Constructing the call graph of a program', *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, pp.216–226.

You, Y-P. and Su, Y-C. (2022) 'Reduced O3 subsequence labelling: a stepping stone towards optimisation sequence prediction', *Connection Science*, Vol. 34, No. 1, pp.2860–2877.

Zhongxiao, Y.Z.X. (2023) *LLVM Optimization-Inlining* [online] https://zhuanlan.zhihu.com/p/395552440.

Notes

1 "The ICP optimization is found to be the second most profitable (after inlining) profile-based optimization in a recent study" (Li et al., 2010).

2 A quote: "We only inline call-sites in the original functions, not call-sites that result from inlining other functions." Comments in the source code of "`LegacyInlinerBase::inlineCalls(CallGraphSCC &SCC)`."