Function Outlining

Peng Zhao, José Nelson Amaral

Department of Computing Science

University of Alberta, Edmonton, Canada

{pengzhao, amaral}@cs.ualberta.ca

Abstract

Large functions that are frequently executed, *i.e. hot*, are common in non-numeric applications. These functions present challenges to modern compilers not only because they require more time and resources for compilation, but also because they might degrade runtime performance by preventing optimizations such as function inlining. Fortunately, often large portions of the code in hot functions are rarely executed, *i.e. cold*. This paper describes a function outlining technique to split cold regions out of the hot host function so that these functions become smaller and more amenable for other optimizations. Unlike other work, our function outlining occurs in the very early phase in a compiler backend. This early outlining will enable more aggressive optimization in the down-stream phases of the compilation, such as inlining and code placement. We describe challenges to function outlining transformations and our solutions. We found that, with a carefully selected strategy, function outlining reduces the sizes of large hot functions significantly (up to 97% and 39% in average) without hurting performance. This is a very promising starting-point for potential client optimizations such as partial function inlining and code placement.

1 Introduction

Algorithms used in optimizing compilers are often applied to the scope of a function. Many of these algorithms have super-linear time and spatial complexity on their inputs. Thus compiling a program with large functions demands large memory storage and is time-consuming. Large functions also impose limitations on other optimizations such as function inlining and code placement. The inlining heuristics used in most compilers avoid inlining call sites that target large callees. The goal of such heuristics is to prevent excessive code growth, also referred to as the *code bloat problem* [5, 6, 7, 27]. For example, large functions are the most prominent cause that prevent the Open Research Compiler (ORC) from eliminating frequent called, *i.e. hot*, sites. ORC only eliminates about 30% of the runtime function invocations for `gcc` and 57% for `perlbmk` [27]. Moreover, more than 50% of the hot call sites are not inlined because the callee is too large.

Large functions also undermine inter-procedural code layout algorithms. For instance, Pettis and Hansen's "closest is best" code layout algorithm tries to place a caller function next to its most frequently invoked callee [18]. The intuition is that proximity between call site and callee in memory enhances performance. However, if the caller itself is very large, its most frequently callee may still be placed far away from the call site, defeating the code placement heuristic.

Fortunately, not every statement in a frequently called large function is equally important or executed as often as its host function. There are many examples of large but infrequently executed, *i.e. cold*, code in hot functions [14, 15]. For instance, only 8.1% of the code in the BSD version of the TCP network protocol implementation is hot[15]. Another example is the function *regmatch* in `perlbmk`. *Regmatch* contains a switch-case statement with about 900 lines of C code to handle 57 string matching scenarios. Although these 900 lines of code are evenly distributed through the 57 cases, only 12 cases occur frequently. Splitting cold code out of hot functions, *i.e. outlining*, is a natural solution to overcome the negative impact of mixing codes with heterogeneous execution frequency. The advantages of outlining cold regions of a hot function are at least three-fold:

Enabling Inlining. When a large cold region is outlined from a hot function, the hot function

might become small enough to enable its inlining.

Improving Cache Efficiency. Without outlining, the aggregation effect of large cache lines reduces spatial locality. Cold statements may be loaded into the cache when hot statements are loaded. Segregating hot and cold regions into separate functions enables better code placement to improve the cache utilization.

Improving Instruction Fetch Bandwidth. Modern superscalar and VLIW architectures demand high instruction bandwidth of the memory hierarchy. A sufficient number of useful instructions must be fetched into the cache for full utilization of the functional units in the processors. For instance, Mosberger *et al.* found that limited instruction bandwidth results in almost 70% of CPU cycles idle in some architectures [14]. Separating the hot code from the cold code also improves the utilization of instruction fetch bandwidth.

A negative performance impact of outlining is that extra function calls are introduced to transfer control between the outlined region and the other parts of the program unit. An efficient implementation of outlining should minimize this performance penalty. This paper describes the following contributions:

- An abstract syntax tree (ABS)-based region formation that efficiently exploit high level control flow structures and their associated feedback information. Compared to Hank's region formation algorithm [10], which is based on control flow graph (CFG), ABS-based region formation is straightforward and results in less outlining performance penalty.
- An argument that the *Optimal Outlining Problem* (OOP) is NP-hard.
- An effective heuristic to analyze the benefit of outlining a region. This heuristic decision weighs the benefit of reducing the host function size against the frequency of the extra function calls introduced. We also describe how to patch the control flow and data flow to preserve program semantics after outlining.

- A novel technique, *alias agent*, to disambiguate parameters created to pass to outlined functions from their counterparts in the host function. Because outlining is an early code transformation, it may negatively impact existent downstream optimizations. Our experiments show that more complex alias relationships created by these parameters causes the major impact on downstream optimizations and result in the introduction of substantial memory spills.

- A study of two orthogonal function splitting strategies: (1) *collective* VS. *independent* splitting; and (2) splitting with VS. without alias agent. This study shows that selecting the correct strategy is crucial. Independent splitting with alias agent reduces function sizes significantly while minimizing the performance penalty of outlining.

Section 2 introduces the intermediate representation where outlining is implemented and the concept of region. Section 3 describes the design and implementation of outlining. Section 4, compares the different outlining strategies. Finally, we discuss about the related work in Section 5.

2 Background

An important motivation for function outlining is to enable more aggressive inlining, which is a major component of inter-procedural optimization (IPO) in ORC. ORC performs IPO very early to enable aggressive function-level compilation. It is thus natural to also implement outlining early in this compiler. The outlining analysis and transformation described in this paper is a transformation of the abstract syntax tree, called WHIRL in ORC, of the function affected.

2.1 WHIRL Tree Introduction

ORC's intermediate representation has five levels, from *very high WHIRL* to *very low WHIRL* [20]. At the higher levels the WHIRL representation of a function is close the original source code. We implemented outlining on very high WHIRL where high level hierarchical control flow constructs — such as if, *loop* and *switch* — have not been transformed to flat constructs —- such as conditional

branches and *gotos*[1]. Thus outlining can take advantage of these hierarchical constructs and their associated frequency information to identify the cold code segments in a single pass through the WHIRL tree.

A contrived function, $HotPU$ shown in Figure 1, illustrates the WHIRL tree representation. Statements are annotated with their execution frequency obtained from runtime profiling. Assume that $HotPU$ is frequently invoked. The shaded code segments or nodes are the cold parts of $HotPU$.

In very high WHIRL, three control flow constructs may lead to infrequently executed code in a hot function:

if* statement.** An ***if node in a WHIRL tree has two children: a ***THEN*** block and an ***ELSE*** block. The feedback information contains the execution frequency of each branch. For example, in Figure 1 both if statements have skewed execution frequency.

switch* statement.** In Figure 1, each CG node corresponds to an enumerated case in a ***switch statement. If the switch expression (or key) equals to n, CG_n node is executed and the program jumps to A_n node (the corresponding action code for case n). If the switch expression doesn't equal to any of the enumerated cases, program jumps to the A_d node (the default action) by the DG node. Feedback information associated switch statement indicates the execution frequency of each case. Studies have shown that many switch statements have skewed execution frequency distribution [28]. In Figure 1, only two of the cases in the *switch* statement are hot.

Early return. Early return occurs when the ***return*** statement or an *exit* function call appears early in a function. Each *return* statement is annotated with its execution frequency. A hot early return implies that the rest of the function is cold. In Figure 1, there are three early returns at line 12, 15 and 18 (or the node $return0$, $return1$ and $return2$ in the WHIRL tree). However, only $return2$ (at line 18) is hot.

[1]Programming with *gotos* is a heavily criticized and very infrequent practice.

HotPU //1000

```
1.  switch (key)
2.      case 1: ... break; // 500
3.      case 2: ... break; // 0
4.      case 3: ... break; // 500
5.      case 4: ... break; // 0
6.      default: ... // 0
7.  endswitch
8.  if (i > 100) // 1, if1
9.      while (1) // 2
10.         ... // loop body
11.     endwhile
12.     return 0; // 1, ER(Early Return)
13. else // 999
14.     if ( i == 101) // 0, if2
15.         return 1; // ER
16.     else // 999
17.         i - -;
18.         return 2; //999, frequent ER
19.     endif
20.
21.     printf("1. not touched"); // 0
22. endif
23.
24. printf("2. not touched"); // 0
25. printf("3. not touched"); // 0
```

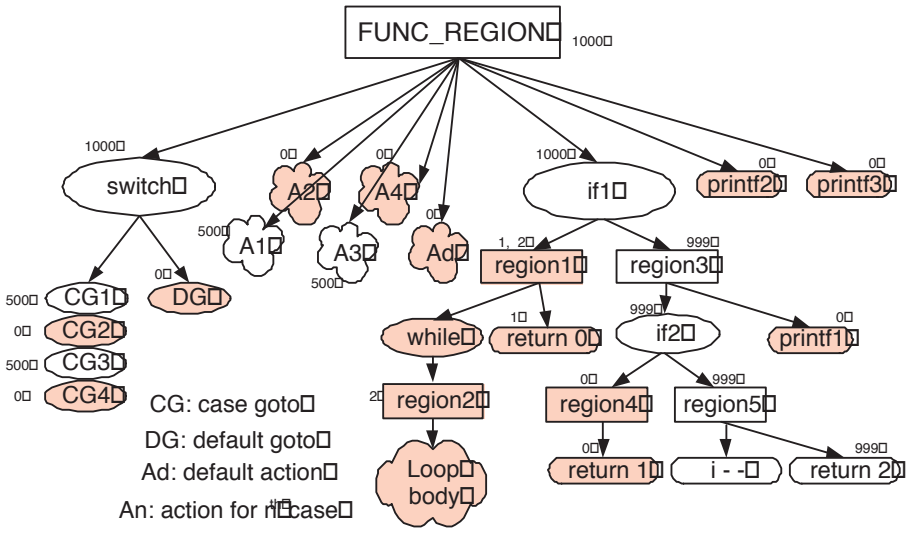


Figure 1: Example Source Code & WHIRL Tree

Analyzing high level control flow constructs and their corresponding frequency annotation makes spotting the cold code in a hot function very straightforward.

2.2 Region

In this paper, region is a sequence of code in the program that is guarded by a high level control flow construct such as *if* and loop statements. For instance, for an *if-then-else* statement, the code executed under the *then* branch consists of a region and the code executed under the *else* branch forms another region. Likewise, the loop body of a *while* statement is a region.

A region is represented in WHIRL by a subtree with a BLOCK node as the root. This node has an arbitrary number of children representing the content of the region. The order of a child c in a region R, $order(c, R)$ is the one from the source code. In Figure 1, $order(while,\ region1) = 1$ and $order(return0,\ region1) = 2$. A WHIRL node w has a parent, $parent(w)$, and a set of ancestors, $ancestors(w)$. In Figure 1, $parent(while) = region1$ and $ancestors(while) = \{region1, if1, FUNC\_region\}$. If $s \in R$, R must be an ancestor of s. The position of s in R, $Pos(s, R)$, is given by:

$$Pos(s,\ R) = \begin{cases} order(s,\ R) & if\ parent(s) = R \\ order(s_A,\ R) & if\ (s_A \in ancestors(s)) \wedge (parent(s_A) = R) \end{cases}$$

Thus, if an ancestor of s is a child of R, the position of s in R is the order of that ancestor. In Figure 1, $Pos(return2, region5) = order(return2, region5) = 2$ and $Pos(return2, region3) = order(if2, region3) = 1$.

Given a node z in a WHIRL tree W, the level of z in W, $Level(z, W)$ is the number of edges that have to be traversed from the root of W to reach z. The root of W is at level 0.

Given a WHIRL tree W and two nodes $y \in W$ and $z \in W$, the nearest common ancestor of y and z, $NCA(w, v)$ is a WHIRL tree node $s \in W$ such that all the following conditions are true:

1. $s \in ancestors(y) \wedge s \in ancestors(z)$
2. $\nexists\, t \in W, (t \neq s) \wedge (t \in ancestors(y)) \wedge (t \in ancestors(z)) \wedge (Level(t, W) > Level(s, W))$

An early return statement short-circuits the rest of the current function. However, the short-circuited code might reside in different levels and different regions in the whirl tree. For example, $return2$ leads to three unexecuted *print* statement: $printf1$ in $region3$; $printf2$ and $printf3$ in

$FUNC\_Region$. The code short-circuited by an early return er in region R, $SC(er, R)$ is defined by:

$$SC(er,\ R) = \left\{ s \;\middle|\; \Big(A = NCA(er, s)) \wedge \Big(Pos(s, A) > Pos(er, A)\Big) \right\}$$

In the example, $SC(return2,\ region3)$ includes $printf1$ and $SC(return2,\ FUNC\_region)$ includes $printf2$ and $printf3$.

3 Function Outlining

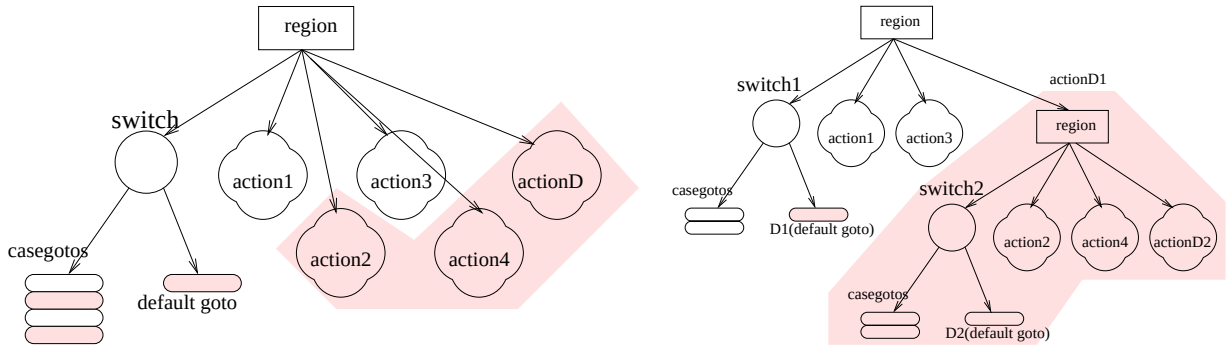
There are three phases in function outlining optimization: *region reorganization* transforms the WHIRL tree so that the cold code is in separated regions from hot code; *candidate identification* identifies regions for which outlining is beneficial; *function splitting* generates a new function from a candidate region and replaces the region with a call to the new function.

3.1 Region Reorganization

In biased if statements, the hot code and the cold code are well structured in two separate sub-regions. However, for *switch* statements and early returns, hot and cold codes are mixed with each other. A depth-first pass on the WHIRL tree reorganizes these codes into different regions for convenient splitting.

3.1.1 Partitioning a *Switch* Statement

Unlike ***if*** and loop constructs, the abstract syntax tree representation of a ***switch*** statement is not well structured. Consider the WHIRL representation of a 4-case ***switch*** shown in Figure 2(a). Assume that cases 1 and 3 are hot and the other cases are cold. Both the case selection and the case actions of the hot and cold cases are mixed together. *Switch-case partitioning* splits the ***switch*** statement into two nested ***switch*** statements, as shown in Figure 2(b). The parent ***switch*** contains only hot cases and their corresponding action code. The cold cases and their action code are placed into the nested ***switch*** statement. The advantage of this re-organization is three fold:



(a) Before Partition (b) After Partition

Figure 2: Partition a ***switch*** statement

(1) the cold ***switch*** statement is in an independent region and can be easily split into a separate function; (2) the case selection phase for the hot cases may be faster because now the cold cases are out of the way; (3) code layout can be improved by placing the cold ***switch*** away from the hot actions. The only disadvantage is that the case selection for cold cases is delayed. However, this will not hurt the performance as long as cases classified as cold by profiling feedback are indeed cold.

The depiction of ***switch*** splitting in Figure 2 is conceptual and assumes that the *default* case is cold and can be moved into the cold ***switch*** statement. When the *default* case is also hot, ***switch*** splitting should behave differently: the case selection must remain in the original ***switch*** statement and the cold case selection must be duplicated in the cold ***switch*** [28].

3.1.2 Handling Frequent Early Returns (ER)

The algorithm HandleER, shown in Figure 3, handles early returns. HandleER is called when an early return S_{er} is encountered during the depth-first traversing of the WHIRL tree. $ReturnFreq$ accumulates the execution frequency of early returns (step 1). Its value is reset to zero before the scan of a function starts, and is preserved between calls to HandleER. Unless S_{er} resides in a loop body, when the ratio between the accumulated frequency and the frequency of the host function reaches the $ERThreshold$, the code after S_{er} is cold. If S_{er} is inside a loop body S_{loop}, it

is possible that the code in $SC(S_{er}, S_{loop})$ is still hot and we avoid outlining it. We use an upward traversal from the early return S_{er} to find its uppermost loop ancestor S_{loop} (step 6-9). If there is no loop ancestor, S_{loop} is set to be S_{er} itself. The cold code resulted from frequent early return is $SC(S_{loop}, FUNC\_BODY)$. The cold code might spread into different levels of the WHIRL tree (*e.g.* the three *printf* statements in our example). To preserve program correctness, they cannot simply be put together in a single region. Instead, an upward traversal from S_{er} (step 11- 15) extracts the cold code of every region that it encounters into a new region (step 14).

```
HandleER (S_er)
1.  ReturnFreq ← ReturnFreq + GetFreq(S_er)
2.  if ( ReturnFreq / GetFreq(HostFunc) ⩽ ERThreshold )
3.     return
4.  S_loop ← S_er
5.  CurrentParent ← GetParent(S_er)
6.  while ( CurrentParent ≠ ROOT) // ROOT is the root of the WHIRL tree.
7.     if (CurrentParent is a loop construct)
8.        S_loop ← CurrentParent
9.     CurrentParent ← GetParent(CurrentParent)
10. CurrentNode ← S_loop
11. while (CurrentNode ≠ ROOT)
12.    CurrentParent ← GetParent(CurrentNode)
13.    if (CurrentParent is a region)
14.       call ExtractColdCodeIntoRegion( SC(CurrentNode, CurrentParent) )
15.    CurrentNode ← CurrentParent
```

Figure 3: Handling Early Exits.

3.2 Outlining Candidate Identification

After region reorganization, every cold code snippet is placed in an independent region and annotated with $(frequency, size)$. The *size* of a region is the number of WHIRL nodes in that region. Next the compiler identifies cold regions that are suitable for outlining.

3.2.1 Hazardous program units for outlining

Some program units are not outlined to prevent performance degradation or to preserve program correctness. Besides trivial functions that are rarely executed, the following are not outlined.

Small regions. Outlining replaces a region in a host function, f_{host}, with a function call to a

new function, f_{out}. Code patches are often required, before and after the call to f_{out}, to preserve correctness. If a region is too small, these patches might be larger than the outlined region. This kind of outlining is strictly avoided because it fails to reduce the size of the original program unit.

Regions with escaped *alloca*-allocated memory. *Alloca* allocates memory space in the stack frame of a function. This memory is automatically freed when the function returns. When a function uses *alloca* to allocate memory in a region and references the allocated memory outside of the region, the region should not be outlined. This is because f_{out} would allocate a memory block with *alloca* and pass this block to f_{host}. It would be difficult to maintain the original semantics of the program because the memory allocated in f_{out} would be automatically freed at its exit and would be no longer valid in f_{host}.

3.2.2 Optimal Outlining Problem is NP-hard

Cold regions are not always beneficial for outlining. The major benefit of outlining is the size reduction of the host function that enables more aggressive inlining and improves code layout. Splitting a segment of code out of a function has several costs. First, necessary code patches may eliminate the size reduction benefit. Second, because the original cold region in f_{host} is replaced by a function call to f_{out}, there is a performance penalty to execute the cold region. It is a hazard to outline hot code because it will result in many runtime function calls. Therefore, an *Optimal Outlining Problem* (OOP) can be formulated as a constrained optimization problem: f_{host} is formed by a set of regions. Assume precise frequency F_i and size S_i information for each region R_i in f_{host}, and a given budget of extra runtime calls K, find a set of regions that, when outlined, minimizes the size of f_{host} without exceeding K. The 0-1 knapsack problem [9] can be reduced to OOP. Given N items $\langle v_i, w_i \rangle$, each item with value $v_i > 0$ and weight $w_i > 0$. The 0-1 knapsack problem is the problem of finding a vector with N binary elements d_i that satisfy the following condition:

$$Maximize(\sum_{i=1}^{N}(d_i * v_i)) \text{ such\_that } \sum_{i=1}^{N}(d_i * w_i) \leqslant K \text{ and } d_i \in \{0,1\} \quad (1)$$

Given a knapsack with capacity K and N items in the 0-1 knapsack problem, we construct an OOP function as follows. Each item $\langle v_i, w_i \rangle$ represents a region with size v_i and outlining cost w_i. Let K be the extra runtime call budget. The conversion complexity is linear to the number of items N. Therefore, the 0-1 knapsack problem can be reduced to OOP in linear time. The 0-1 knapsack problem is a well-known NP-hard problem, therefore OOP is also NP-hard. Thus, a reasonable heuristic to find approximate solutions to OOP is necessary.

3.2.3 Engineering Approach to Selective Outlining

Given a region R_i in a function f_{host} with frequency F_i and size S_i, we define the frequency ratio of R_i and the size ratio of R_i as:

$$size\_ratio(R_i) = \frac{S_i}{size(f_{host})} \tag{2}$$

$$freq\_ratio(R_i) = \frac{F_i}{frequency(f_{host})} \tag{3}$$

We adopt a popular *knapsack problem* greedy algorithm to estimate the benefits of splitting a region R_i out of f_{host}. The greedy algorithm has tight time and space bounds and has been proved effective in many cases. The idea is to calculate the profit density for each region. In this framework the profit density of a region is called *benefit*:

$$benefit(R_i) = \frac{size\_ratio(R_i)}{freq\_ratio(R_i)} \tag{4}$$

The regions are then sorted in decreasing order of their *benefit* value. Regions are selected for outlining, equivalent to placing items into a knapsack, until the constraint in equation 1 is violated, *i.e.* the knapsack cannot hold any more items.

Essentially, $size\_ratio(R_i)$ and $freq\_ratio(R_i)$ estimate the contribution of R_i to the total size and execution frequency of f_{host}. Therefore, this heuristics favors large cold regions. Intuitively, larger regions that are not executed frequently should produce the most benefit from outlining and incur in the least runtime penalty.

Also, a $FreqRatioThreshold$ ensures that the number of invocations of f_{out} does not exceed a small percentage of the invocations of f_{host}:

$$F_i \ \leq \ FreqRatioThreshold \times frequency(f_{host}) \tag{5}$$

In our implementation, $FreqRatioThreshold = 0.01$.

3.3 Function Splitting

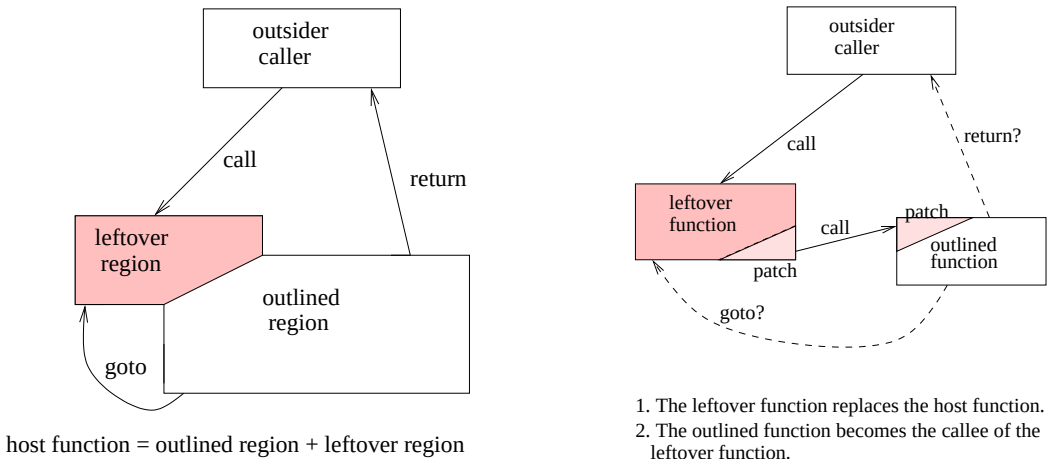
This section discusses the most important aspects of the function splitting process. A complete description is available in [28].

3.3.1 Splitting and Patching

A *region* R_{out} is selected to be outlined from a *host function* f_{host}. The *leftover code* remaining in f_{host} forms $R_{leftover}$. An *outside caller*, f_{caller}, is a function that contains a call to f_{host}. The newly generated *outlined function*, f_{out}, contains R_{out}. A *leftover function*, $f_{leftover}$, is a new version of f_{host} where R_{out} is replaced by a call to f_{out} plus code patches. After outlining (Figure 4.b), f_{caller} calls $f_{leftover}$, and $f_{leftover}$ calls f_{out}. $f_{leftover}$ inherits all the original resources of f_{host} including the function name, patched WHIRL tree and symbol table. To preserve program semantics, the following patches are required.

Patching Data Flow. To maintain data flow integrity, local variables accessed in R_{out} are passed as parameter to f_{out}. If all references to a variable x in R_{out} are *uses*, the value of x is passed to f_{out}. If x is ever *defined* in R_{out}, the address of x is passed as a parameter, and all references to x in f_{out} are changed correspondingly.

Patching Control Flow. Three kinds of inter-region control flow need to be handled with care: side-entrances, side-exits and *return* statements. If f_{host} enters R_{out} via several entries, then $f_{leftover}$ must inform, in each invocation of f_{out}, which entry should be taken. If R_{out} jumps to an arbitrary label in $R_{leftover}$, $f_{leftover}$ must be informed at which label execution should restart after returning from f_{out}. If R_{out} contains a *return* statement, $f_{leftover}$ must return to f_{caller}



(a) Before outlining (b) After outlining

Figure 4: Outlining transformation

immediately upon the return of f_{out}. To achieve this hand-shaking, we introduce two new local variables in $f_{leftover}$, $Flag$ and $ReturnValue$. The addresses of $Flag$ and $ReturnValue$ are both passed to f_{out}. Before invoking f_{out}, $Flag$ is initialized to tell which entry should be taken. At the return of f_{out}, $Flag$ specifies control flow dispatching as explained in Table 1. If $Flag$ equals 0, the $f_{leftover}$ returns to its caller immediately upon return from f_{out}. If $Flag$ is 1, the statement immediately following R_{out} is executed after f_{out} returns. Otherwise, the control should flow to a specific label in $f_{leftover}$. If $Flag \geq 2$, the value of $Flag$ is used to index a jump table containing the addresses for a computed *goto* statement.[2] $ReturnValue$ has the same type as the return type of $f_{leftover}$. When f_{out} needs to return a value, it saves the value to be returned in $ReturnValue$ and sets $Flag = 1$. When $Flag$ is 1 on f_{out}'s return, $f_{leftover}$ immediately returns the value stored in $ReturnValue$ to f_{caller}.

$Flag$	Action
0	return $Return\_Value$
1	fall through
≥ 2	computed goto ($Flag$ - 2, JUMPTABLE)

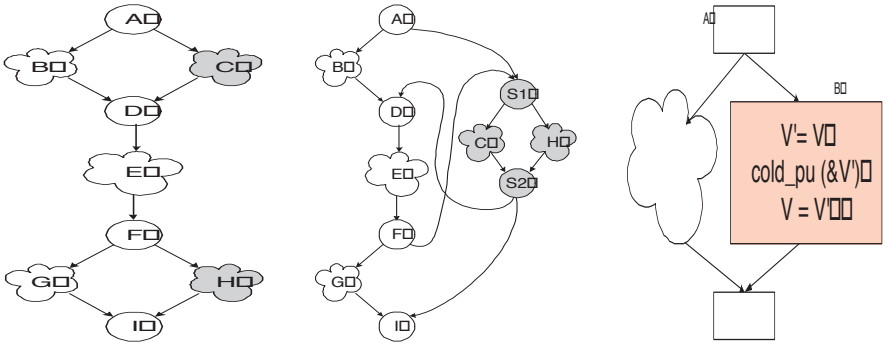
Table 1: Semantics of $Flag$ and $Return\_Value$ on the return of f_{out}

[2]To be precise, it is the value of $Flag - 2$ that is used to index the table.

3.3.2 Performance Tuning

This section describes important performance tuning for the outlining optimization. The motivation of outlining is to enable more aggressive inlining. However, as we have discussed, outlining itself has several negative impacts on performance. As section 4 shows, the impact of outlining strategy selection on performance is dramatic. This section, describes these strategies.

Independent outlining VS. Collective outlining. Regions to be outlined may be scattered throughout f_{host}. Two possible outlining strategies are *independent outlining* and *collective outlining*. In independent outlining, each cold region is split into a separate function as shown in Figure 5(a). In collective outlining a single f_{out} function contains all outlined regions (Figure 5(b)). In this case the $Flag$ parameter is used to dispatch control to the correct region whenever f_{out} is invoked. Each outlined region in $f_{leftover}$ is replaced with an assignment to $Flag$, a call site to f_{out} and control flow patching code after the call site. A drawback of collective outlining is a more complex CFG in $f_{leftover}$ which may be difficult for downstream compiler analysis.



(a) Independent (b) Collective (c) Alias Agent

Figure 5: Different Strategies (Shaded code is cold)

Alias Agent When the address of a variable x is passed as a parameter to f_{out}, imprecise alias analysis will conservatively assume that x can now be aliased to any other variable that f_{out} has access to. This conservative assumption may prevent downstream optimizations and result in serious performance penalty. For instance variables that were kept exclusively into registers might

be spilled after outlining is implemented because of imprecise alias information. This situation occurs often in ORC 2.1 and constitutes a performance hazard. Our solution is to introduce an *alias agent technique* to eliminate this serious side-effect. Each variable v whose address is passed to f_{out} has a corresponding alias agent v'. An alias agent is a new local variable introduced in f_{host}. Just before the invocation of f_{out}, the value of v is copied into v'. Then the address of v' is passed to f_{out}. Upon return from f_{out}, the value of v' is copied into v as shown in Figure 5(c). Both copies occur in the same cold basic block that contains an invocation to f_{out}. Without *alias agent*, we found that ORC often places memory spills in hot paths (*e.g.* into block A instead of B in Figure 5(c)), degrading runtime performance significantly.

4 Results

The results of our experimental investigation of outlining on SPEC2000 integer benchmarks may be summarized as follows:

- Outlining can reduce the size of hot functions by up to 97%. This impressive size reduction makes these hot functions more amenable to later inlining.
- Independent splitting significantly outperforms collective splitting. When combined with alias agents, independent splitting results in outlining with practically no performance penalty.
- Alias agents are crucial to minimize the performance penalty of outlining. Without alias agent, memory spills may be placed in hot paths, resulting in significant performance degradation (ranges from 0.6% to 11.3%).
- Although outlining increased retired instructions and function calls, sometimes it may slightly improve performance. In our experiments, outlining improves `perlbmk` by about 1.7%.

Experimental Framework: This research uses the Open Research Compiler 2.1 (ORC) as an experimental platform. ORC is an open-source compiler that evolved from the SGI's MIPSPro compiler, which implements a rich set of optimizations including Inter-Procedural Optimizations

(IPO) and complete program analysis support. ORC generates binaries for Intel's 64-bit Itanium processors.

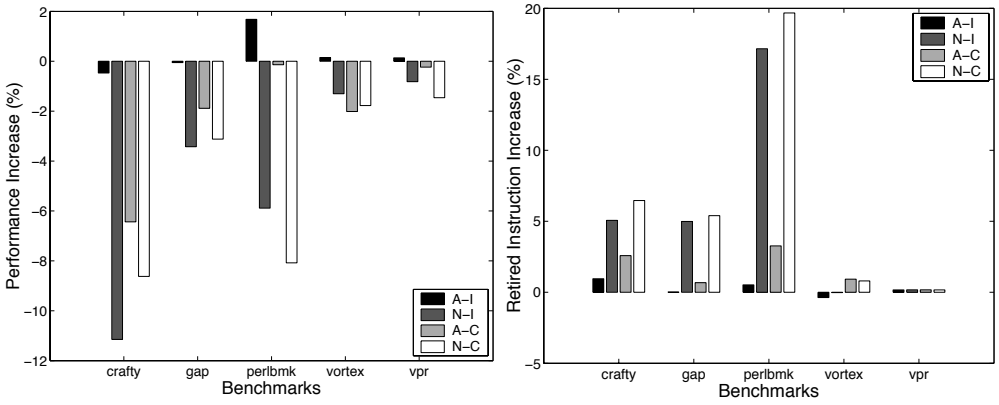
Experiment Configuration: Experimental results were obtained on an HP ZX6000 workstation with a 1.3GHz Itanium-2 processor, 1 GB of main memory, 32KB of L1 cache, 256KB of L2 Cache, and 1.5MB of on-die L3 cache. The operating system is Red Hat Linux 7.2 with a 2.4.18 kernel. This experimental study is based on SPEC2000 integer benchmarks that have large hot functions: `crafty`, `gap`, `perlbmk`, `vortex`, `vpr`. Time is measured by the Linux *time* command and micro-architectural benchmarking is obtained with *pfmon*. All reported run-times are the average of 5 consecutive identical runs.

4.1 Statistics for Outlining Transformation

Table 2 presents a study of outlining in each benchmark. In this framework, outlining occurs only on hot functions that cannot be inlined because of their large sizes. The first row of Table 2 contains a static count of the number of regions split. The *Control Flow Construct* rows show the frequency of each type of control flow constructs where cold regions are found. The major source of cold regions in `crafty`, `gap` and `vpr` are *if* statements. Benchmarks `perlbmk` and `vortex` have large switch-case statements in which a small portions of cases dominates the execution. Early returns are not frequent. The *Function Size Reduction* row shows that outlining can drastically reduce the size of large functions: size reduction ranged from 1% to 97%. Except for `crafty`, the average function size reduction is greater than 39%. The *Parameter Pass* row reports that the number of parameters required for outlined functions ranges from 2 to 18 in the benchmarks. Finally, the *Runtime Call Increase* reports that the increase in the number of runtime calls never exceeds 0.15%.

4.2 Performance Analysis

This section compares the runtime performance of the four outlining strategies listed in Table 3. The four strategies result from two orthogonal decisions: (i) whether the alias agent technique is used to prevent false aliases; (ii) whether collective or independent splitting is used.



(a) Runtime Performance.

(b) Retired Instructions.

(c) Processor Stalls.

(d) Instructions/Cycle.

Figure 6: Execution time and microarchitecure performance of outlining.

Benchmarks		crafty	gap	perlbmk	vortex	vpr
Number of Regions Split		18	4	12	16	5
Control Flow Construct	if	18	3	6	8	5
	switch	0	0	6	7	0
	early return	0	1	0	1	0
Function Size reduction	min-max(%)	1-24	7-97	40-66	8-71	24-69
	average(%)	11.6	39.4	49.8	36.9	46.4
Parameter Pass	min-max	2-5	3-6	2-16	2-18	5-9
	average	3.2	5.0	7.5	10.4	7.0
Runtime Call Increase (%)		0.15	$6.85 * 10^{-6}$	$9.50 * 10^{-3}$	0.05	0.08

Table 2: Statistics of Outlining

Short	Explanation
A-I	Alias agent and independent splitting.
N-I	No alias agent and independent splitting.
A-C	Alias agent and collective splitting.
N-C	No alias agent and collective splitting.

Table 3: Strategy Combinations

Figure 6(a) shows the performance changes caused by these different strategies. Strategy selection has significant impact on performance, here are some important observations.

- Independent splitting with alias agent (A-I) always out-performs the other versions. Considering that outlining introduces extra function calls and extra instructions, the fact that A-I delivers outlining with no practical performance penalty is impressive: the performance difference is negligible for `crafty`, `gap`, `vortex` and `vpr`. For `perlbmk` A-I outlining even improves performance by about 1.7% (see Section 4.3).

- Without the alias agent (N-I and N-C strategies) outlining significantly degrades performance (see results for `crafty`, `gap` and `perlbmk`). This is a consequence of ORC's imprecise interprocedural alias analysis.

- Collective splitting outlining results in higher performance penalty and thus should be avoided.

4.3 Micro-architecture Level Benchmarking

This section investigates how different outlining strategies change the micro-architectural behavior of the benchmarks and therefore the runtime performance.

Figure 6(b) shows the number of instructions executed.[3] The A-I strategy always executes the least instructions. This is evidence that the region selection algorithm is indeed selecting cold regions. If the alias agent technique is not used (*i.e.* N-I and N-C), many additional instructions are executed in `crafty`, `gap` and `perlbmk`. This increase is specially dramatic in `perlbmk` where it reaches 17.2% for N-I. Collective splitting executes from 0.7% to 3.3% more instructions than independent splitting.

Figure 6(c) presents a comparison of processor stalls under the four strategies. While the number of stalls always increases, A-I produces the least number of extra stalls (0.1% to 1.5%). Outlining without alias agent increases stalls in the range of 0.4% to 18.8%.

It may be surprising that although A-I outlining increases the number of retired instructions (by about 0.5%), and the number of processor stalls (by about 1.3%), it still improves the runtime performance (by about 1.7%) for `perlbmk`. A study of the average number of instructions per cycle (IPC), shown in Figure 6(d), offers an explanation. A-I outlining improves the IPC of `perlbmk` by 2.6%. Thus, it seems that the compiler does a better job, in later optimizations, of improving the processor's functional unit utilization. Although the improvements on IPC for `perlbmk` in the other strategies is even higher, the benefit is dwarfed by the the number of additional retired instructions and stalls in those strategies.

5 Related Work

There is previous work related to several aspects of the design and implementation of the outlining presented in this paper.

[3]The numbers reported reflect the number of instructions *retired*.

5.1 Function Splitting

Several researchers have studied function splitting [4, 14, 15, 18, 21, 22, 23, 24]. In their well-known code positioning paper, Pettis and Hansen split a function and put the hot and cold code far apart in the address space [18]. They don't generate new functions for the split code. Control transition between hot and cold code is by explicit jump instructions. When the two parts are located too far away from each other, code stubs are needed for relaying jumps. Their motivation is to reduce the size of the primary function, which contains the hot code, to allow important related code to co-exist in the instruction cache or to be placed in the same memory page. Their function splitting is only intended for code placement, therefore it is implemented at the link phase. Their control flow patching method breaks the address integrity of a function and is difficult to implement and maintain in high level optimizations.

Castelluccia *et al.* and Mosberger *et al.* use outlining to increase the code density of network protocol code[4, 14]. However, they only handle if statements. Our experimental work shows that *switch* statements and early returns are also important causes for unbalanced execution frequency in standard benchmarks. Thus our outlining framework includes outlining of these constructs.

Muth *et al.* proposed the implementation of partial inlining in a link-time optimizer called ALTO [15]. They generate a new program unit to hold all the split code. Once control is transfered to the program unit containing the cold code, it cannot return to the unit containing the hot code. As a consequence, the cold program unit has to clone any portion of the hot code that is reachable from the cold region, making the code bloat problem worse. In ORC, duplicating all the code that can be reached from the cold WHIRL tree would likely increase the parameter passing overhead. While we share Muth's motivation, we think that their outlining and partial inlining occur too late in the compilation process to allow other optimizations to benefit from partial inlining. Very few optimizations occur after linking. In contrast, our outlining occurs in the very beginning of the backend. Early outlining enables aggressive inlining and potentially benefits all the later optimizations.

Way *et al.* experimented with partial inlining on high level intermediate representation[22, 23, 24]. Their inlining is an enabling technique to build inter-procedural regions and reduce optimization costs. As a by-product of their CFG-based inter-procedural formation work, they manually find cold portions of a program. Then they manually partially clone the cold code into a new function and replace the cold code with a call to the new function. While the concepts that they study are similar to the ones presented in this paper, their hand manipulation of code is not suitable for commercial compilers. In contrast, our tree-based outlining and enabling pre-processing techniques make partial inlining suitable for production-strong compilers.

Suganuma *et al.* propose partial inlining for a Java just-in-time compiler. Their work takes advantage of the on-stack-replacement technique [11] supported by their JVM and therefore they do not need to worry about control flow patching during outlining.

5.2 Handling Unstructured Control Flows

Many approaches to handle unstructured control flow have been described [1, 2, 3, 8, 17, 19, 25, 26]. The common idea is to eliminate unstructured control flow, such as *goto*s and irreducible loops, to make programs more amenable to optimization. We do not need to eliminate unstructured control flow, instead we focus on unstructured flow that results in cold code inside a hot function. Our transformation only tries to group code with similar execution frequency together.

5.3 Region Formation Algorithm

Hank's intra-procedural region formation method [10] is a generalization of the IMPACT compiler runtime feedback-based trace selection algorithm [16]. Hank analyzes the CFG of a function to identify a hot region that includes the entry and exit of the function. Using the most frequent basic block as a seed for the hot region, Hank's algorithm first traverses upward and downward in the CFG to find a most desirable path as the seed path. During this seed path generation process, Hank determines the desirability according to the frequency relevance of the successor or predecessor basic blocks. Once the seed path is selected, the algorithm tries to use similar frequency heuristics to include more relevant basic blocks to generate the hot region. Hank's region formation occurs

after aggressive inlining. Very large functions generated by aggressive inlining may significantly increase compilation time. Hank's motivation is to repartition a large function into small regions to control the compilation time while exposing important optimization opportunities. Way *et al.* later contended that aggressive inlining itself might be expensive. Instead they propose an extension to Hank's algorithm that integrates inlining with inter-procedural region formation [22, 23].

The CFG-based region formation of Suganuma *et al.* tries to identify cold regions in a hot function [21]. They first select some seed basic blocks as *rare* or *non-rare* according to some predefined heuristic. Then this information is propagated along backward data flow until it converges. Then the CFG is traversed again to decide the regions and the transitions between them.

All these region formation methods, including ours, try to separate code segments with heterogeneous execution frequency. We take advantage of frequency-annotated high-level intermediate representation to implement our region formation. Our transformation is closer to the source code and done without CFG formation. We claim that our implementation is more straightforward and easier to debug. When CFG is used to form regions high-level control flow information is lost. For instance, the absence of this information might lead to collective outlining, which we found to be not efficient in Section 4.

5.4 Preservation of Semantics in Splitting

Komondoor *et al.* use function splitting to abstract repetitive code segments to a new function so that the program becomes easier to understand and maintain [12, 13]. We have a different goal: to separate the cold code from a hot function. However, their semantics-preserving methods handle problems that are similar to the ones that we met in our study: some statements, known as *exiting jumps* in their work, such as *return*s in the outlined region and *goto*s from the outlined region to the leftover region should simply not be included in the outlined function. Their splitting candidates are limited to single-entry regions while our splitting framework can handle side-entries to a region. Thus our control flow patching work can be seen as a superset of their techniques.

6 Conclusion

The abstract syntax tree-based function outlining described in this paper features (1) a novel region formation approach that takes full advantage of high level control flow constructs, (2) a set of outlining candidate identification heuristics and (3) a solid patching method to maintain the correct semantics of a program. With proper strategy, this function outlining can significantly reduce the size of large hot functions without performance penalty.

This encouraging result motivates the future study of the performance potential of client optimizations such as inlining the leftover hot function after outlining, *i.e.* partial inlining, and outlining-enhanced code placement. Given that we were able to implement outlining without performance penalty, we expect that these client optimizations should produce significant performance improvements.

7 Acknowledgements

We had a lot of help to perform this work. We thank the SGI team for making the code that originated ORC open source. We thank the ICRC and the ORC team in the Institute of Computing Technology, Chinese Academy of Sciences for building the ORC research infrastructure. Sincere thanks to Sun C. Chan and Shin-Ming Liu for their help and discussion. Thanks also go to people who answered a lot of questions in the ORC and Open64 mailing lists. This research is supported by the Natural Science and Engineering Research Council of Canada (NSERC) and by the Canadian Foundation for Innovation (CFI).

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pages 177–189, Austin, January 1983.

[2] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, March 1992.

[3] B. S. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[4] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 60–72, Palo Alto, CA, USA, 1996.

[5] J. W. Davidson and A. M. Holler. A model of subprogram inlining. Technical report, Computer Science Technical Report TR-89-04, Department of Computer Science, University of Virginia, July 1989.

[6] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software - Practice and Experience (SPE)*, 18(8):775–790, 1989.

[7] J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering (TSE)*, 18(2):89–102, 1992.

[8] A. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *International Conference on Computer Languages (ICCL)*, pages 229–240, May 1994.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[10] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: An introduction and motivation. In *the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, Dec 1995.

[11] U. Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, 1994.

[12] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL)*, pages 155–169, Boston, USA, Jan 2000.

[13] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *11th International Workshop on Program Comprehension (IWPC)*, pages 33–43, Portland, USA, May 2003.

[14] D. Mosberger, L. Peterson, and S. O'Malley. Protocol latency: Mips and reality. Technical report, TR-95-02, Dept. of Computer Science, Univ. of Arizona, 1995.

[15] R. Muth and S. Debray. Partial inlining. Technical report, Dept. of Computer Science, Univ. of Arizona, U. S. A., 1997.

[16] P. P. Chang and W. W. Hwu. Trace selection for compiling large C application programs to microcode. In *Proceedings of the 21st International Workshop on Microprogramming and Microarchitecture*, pages 188–198, Nov 1988.

[17] W. W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat, and exit statements. *Communications of the ACM*, 16(8):503–512, Aug 1973.

[18] K. Pettis and R. C. Hansen. Profile guided code positioning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, 1990.

[19] L. Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM*, 35(4):893–920, October 1988.

[20] SGI. Whirl intermediate language specification, 2000.

[21] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a java just-in-time compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 312–323, 2003.

[22] T. Way. *Procedure Restructuring for Ambitious Optimization.* PhD thesis, University of Delaware, May 2002.

[23] T. Way, B. Breech, and L. L. Pollock. Region formation analysis with demand-driven inlining for region-based optimization. In *The 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 24–36, 2000.

[24] T. Way and L. L. Pollock. A region-based partial inlining algorithm for an ilp optimizing compiler. In *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 552–556, 2002.

[25] M. H. Williams. Generating structured flow diagrams: the nature of unstructuredness. *The Computer Journal*, 20(1):45–50, 1977.

[26] M. H. Williams and H. L. Ossher. Conversion of unstructured flow diagrams to structured form. *The Computer Journal*, 21(2):161–167, 1978.

[27] P. Zhao and J. N. Amaral. To inline or not to inline, enhanced inlining decisions. In *16th Workshop on Languages and Compilers for Parallel Computing*, pages 405–419, College Station, TX, Oct 2003.

[28] P. Zhao and J. N. Amaral. Splitting functions. Technical Report TR04-18, Dept. of Computing Sciences, Univ. of Alberta, Edmonton, Canada, 2004.