Functions Splitting

Peng Zhao and José Nelson Amaral

Department of Computing Science, University of Alberta, Edmonton, Canada
{pengzhao, amaral}@cs.ualberta.ca

Abstract. Large functions appear frequently in today's applications. Large functions present challenges to modern compilers not only because they increase the compilation cost including time and resources but also they usually degrade the quality of the final executables. To reduce these negative impacts, we propose to use runtime feedback information as a guide to split (or outline) the infrequently touched code out of a hot function (called function outlining). In this paper, We first introduce function inlining and the existent impediments agains aggressive inlining, which motivate our function outlining framework. Then we focus on our solution on how to split a part of a function out of it while maintaining correct semantics.

1 Introduction

1.1 Function Inlining

Function inlining is a very important optimization technique that replaces a function call with the body of the function [1, 3–6, 9, 13, 8, 7]. One direct advantage of inlining is that it eliminates the overhead resulting from function calls. The savings are especially pronounced for applications where only a few call sites are responsible for the bulk of the function invocations because inlining those call sites significantly reduces the function invocation overhead. For example, `mcf` (one of the SPEC2000 benchmarks) contains 34 call sites. Among these call sites, there are 5 that are executed more than 10 million times and 4 call sites that are executed more than 1 million times in the standard SPEC2000 training execution. These 9 call sites account for 99.85% of all the function invocations in `mcf`. Our experiments show that inlining the 15 most frequent call sites can reduce the running time of `mcf` by more than 8%[15].

Inlining also produces an augmented context for static analysis. This larger analysis scope creates opportunities for other optimizations. As the body of the program unit is now available at the call site, conservative assumptions that the compiler would previously make about the call site are no longer required.

Another advantage of inlining is the improvement of cache efficiency. From the point of view of the data cache (D-cache), once the callee is inlined, the caller's variables that are referenced by the callee do not need to be passed as parameters. Thus, a variable that previously had separate representations in the caller and in the callee can now be reduced to a single memory location or even promoted to a register. This storage consolidation reduces the data access footprint of the application and improves the use of the memory hierarchy. A similar advantage also exists for the instruction cache (I-cache). After inlining, closely related segments of code are placed together, reducing chances of instruction cache conflicts.

However, inlining has negative effects. One problem with inlining is the growth of the code, also known as *code bloat* in compiler lingo. Because a procedure may be called from multiple call sites, It is often impossible to eliminate a procedure after inlining a single call site. Thus, the final executable file must contain several copies of the procedure: one is the original one (if it is not eliminated as a dead function), the others are the inlined copies.

With the growth of program units because of inlining, the compilation time and the memory space consumption may become intolerable because some of the algorithms used for static analysis have super-linear complexity.

Besides the time and memory resource costs, inlining might also have the adverse effect of increasing the execution time of the application. After inlining, the register pressure may become a limitation because the caller now contains more code, more variables, and more intermediate values. This additional storage requirement may not fit in the register set available in the machine. Thus, inlining may increase the number of register spills[1], resulting in a larger number of load and store instructions executed at runtime.

Moreover, because the caller becomes larger after inlining, the possibility that it will interfere with other procedures in the cache is higher. This interference may cause deterioration of the I-cache efficiency.

Based on the above discussion of the benefits and drawbacks of inlining, we can develop some intuitive criteria to decide which call sites are good candidates for profitable inlining. The benefits of inlining (elimination of function call overhead, enabling of more optimization opportunities, improved cache efficiency) depend on the execution frequency of the call site. The more frequently a call site is invoked, the more promising the inlining of the site is. If the call site is invoked only a couple of hundred times in a long execution, inlining it unlikely to produce any improvement.

On the other hand, the negative effects of inlining relate to the size of the caller and the size of the callee. Larger functions tend to have worse cache behavior and higher register pressure. Inlining large callees results in more serious code bloat, and, probably, performance degradation due to additional memory spills or conflict cache misses.

Thus, we have two basic guidelines for inlining. First, the call site must be very frequent, and, second, neither the callee nor the caller should be too large. However, as we will see in the next section, sometimes we would like to inline a large callee.

[1] In compiler lingo, ***spill*** refers to the temporary movement of a variable from a register to a memory location.

1.2 The thwarted inlining

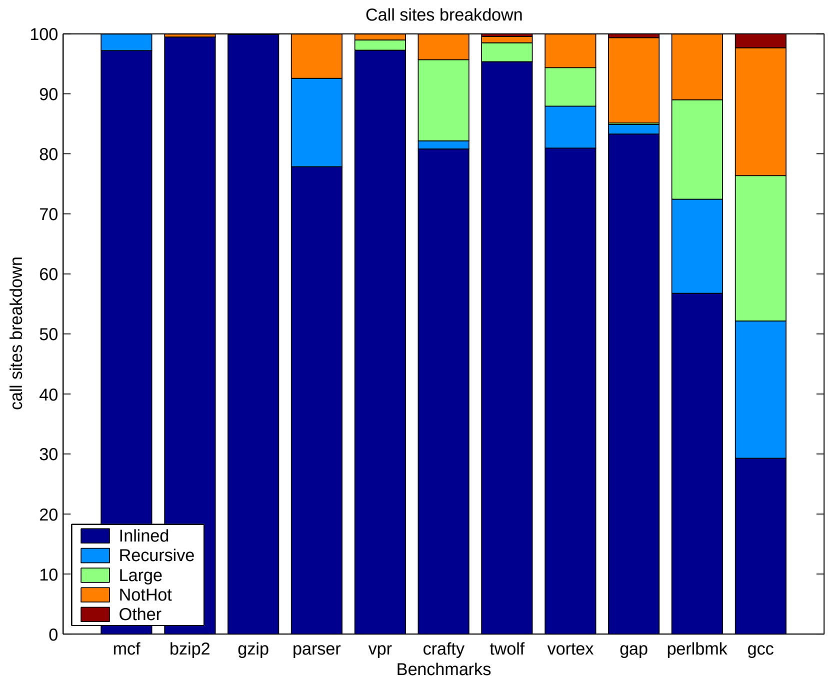


Fig. 1. Dynamic Function Calls Breakdown

We designed a new enhanced inlining criteria for the Open Research Compiler (ORC) [15]. Figure 1 is a study of function calls in the SPECint benchmark suite after our enhancement is applied. We divided the dynamic function calls into five different categories:

Inlined Funcion invocations that can be eliminated by inlining with our enhanced inlining technique.

NotHot Function invocations of call sites that are not frequently invoked. There would be no benefit from inlining these call sites.

Recursive ORC does not inline call sites that are in a cycle in the call graph.

Large Call sites that are hot[2] but cannot be inlined because either the callee, the caller or their combination is too large. This situation occurs quite often in `gcc`, `perlbmk`, `crafty` and `gap` benchmarks from the SPEC2000 suite.

[2] The temperature is a function of the size of the procedure and of the frequency of invocation of the callee. Typically a hot procedure is called often.

Other Function calls that cannot be eliminated by inlining due to some other special reasons. For example, the actual parameters to the call sites do not match the formal parameters of the callee. As Figure 1 shows, these call sites are very rare.

With our enhanced inlining framework, we were able to eliminate most of the dynamic function calls for small benchmarks such as `mcf`, `bzip2` and `gzip`. However we only eliminated about 30% of the dynamic function invocations for `gcc` and 57% for `perlbmk`. Examining the graph in Figure 1, to obtain further benefits from inlining we need to address inlining in these large benchmarks. The categories that are the most promising are the recursive function calls and call sites with large callers or callees.

Figure 1 shows that for some large benchmarks (`parser`, `perlbmk` and `gcc`) a significant portion of the function invocations that are not inlined are recursive functions. We plan to study the depth of the recursions to decide if inlining of recursive functions is promising. If a recursive function is invoked often but its recursion is shallow, limited inlining should be beneficial (the analogous intra-procedural code transformation is loop unrolling).

In order to harvest the benefits from inlining without incurring high costs on code growth and compilation time, we should inline only the portions of the procedures that are actually hot. Thus this research investigates outlining-enabled inlining (*i.e.* partial inlining).

1.3 Why Outlining?

Outlining, the opposite of inlining, splits a program unit into two or more smaller ones. The basic outlining rule in our research is to separate frequently executed statements ("hot" region) from rarely executed statements ("cold" region).[3]

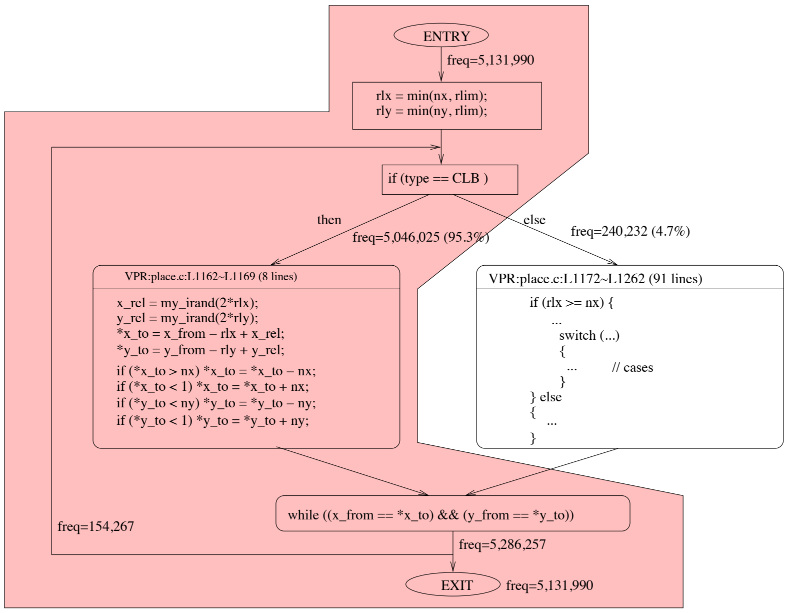
I will use the program unit *find_to* from the `vpr` benchmark[4] to illustrate the concept of outlining and its potential benefits. I start by describing the `vpr` program so that we can understand when outlining opportunities occur.

`vpr` (Versatile Placement and Routing) is a computer-aided design (CAD) tool for integrated circuit design. `vpr` performs placement and routing in FPGAs (Field-Programmable Gate Array). `vpr` placement allocates each circuit function to a circuit logic block (CLB) or I/O pad so that closely related logic functions are close to each other. This placement is essential to minimize the wiring overhead and maximize the circuit speed.

In the `vpr` placement component, the function *find_to* is invoked very frequently and thus is a good candidate for inlining to eliminate the function call overhead. The only caller of *find_to* in the entire program is a function called *try_swap*. *try_swap* randomly choses a logic unit in the FPGA and calls *find_to* to pick a place where the logic unit could be swapped to.

[3] Region is a concept broadly used in compilers. In this work, I often split (*i.e.* outline) a part of code out of a program unit and call the part of code to be outlined an **outlined region**.

[4] One of the benchmarks in the SPECint benchmark suite.



```
rlx = min(nx, rlim);
rly = min(ny, rlim);
```

```
if (type == CLB )
```

```
x_rel = my_irand(2*rlx);
y_rel = my_irand(2*rly);
*x_to = x_from − rlx + x_rel;
*y_to = y_from − rly + y_rel;
if (*x_to > nx) *x_to = *x_to − nx;
if (*x_to < 1) *x_to = *x_to + nx;
if (*y_to < ny) *y_to = *y_to − ny;
if (*y_to < 1) *y_to = *y_to + ny;
```

```
if (rlx >= nx) {
    ...
      switch (...)
      {
        ...        // cases
      }
} else
{
    ...
}
```

```
while ((x_from == *x_to) && (y_from == *y_to))
```

Fig. 2. Annotated CFG of PU *find_to* in `vpr`

Most of the logic units of an FPGA are circuit logic blocks (CLB). For instance, in a typical `vpr` input, the number of CLBs is 8383, while the total number of I/O pads is only 144. Thus, in such a run of `vpr`, 98.3% of the units randomly chosen by *try_swap* are CLBs. However, situations that happen infrequently are not necessarily simpler. In *find_to*, 91 lines of C code are required to handle the infrequent I/O pad case, while only 8 lines of C code are executed when the function is called to handle the popular CLB case.

Figure 2 is a simplified control flow graph (CFG) of the function *find_to* in the `vpr` benchmark. Important edges in the CFG are annotated with the respective frequency. The program unit *find_to* is invoked 5,131,990 times in the standard SPEC2000 training execution. The shadowed area in the CFG of Figure 2 is the hot region of the function *find_to*. Approximately, there are only 14 lines of C code (all the shadowed code in the figure) that are frequently executed. The cold portion (the unshaded code) only accounts for 4.7% of the total frequency, though it is a large else-block with 91 lines of C code. Moreover, the then-block only calls *my_irand* two times while the else-block calls *my_irand* 17 times. Because the function *my_irand* is very small, ORC inlines it at any call site independent of the invocation frequency. After *find_to* absorbs all the *my_irand* call site, the code size weight of the else block (the infrequently executed code) becomes more pronounced.

Although $find\_to$ is the callee of only one call site (which is the top 14th most frequently invoked call site) in the entire program, it cannot be inlined because it has become very large after the inlining of several invocations of my_irand.

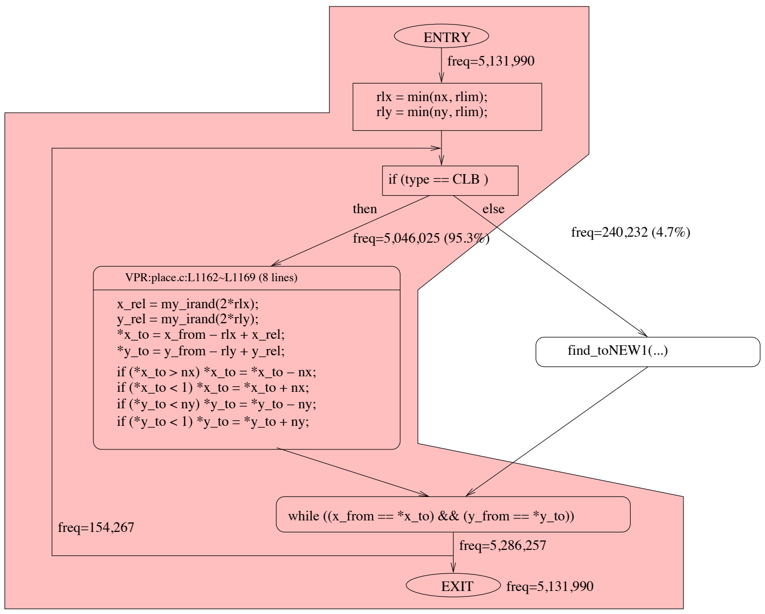

ENTRY
freq=5,131,990
rlx = min(nx, rlim);
rly = min(ny, rlim);
if (type == CLB)
then
else
freq=5,046,025 (95.3%)
freq=240,232 (4.7%)
VPR:place.c:L1162~L1169 (8 lines)
x_rel = my_irand(2*rlx);
y_rel = my_irand(2*rly);
*x_to = x_from – rlx + x_rel;
*y_to = y_from – rly + y_rel;
if (*x_to > nx) *x_to = *x_to – nx;
if (*x_to < 1) *x_to = *x_to + nx;
if (*y_to < ny) *y_to = *y_to – ny;
if (*y_to < 1) *y_to = *y_to + ny;
find_toNEW1(...)
while ((x_from == *x_to) && (y_from == *y_to))
freq=154,267
freq=5,286,257
EXIT
freq=5,131,990

Fig. 3. Annotated CFG of outlined $find\_to$ in `vpr`

Figure 3 shows the result of outlining: the comparatively infrequent code is split from the original PU to generate a new function $find\_toNEW1$. The original cold code is then replaced by a function call to the new function. Now, the function $find\_to$ becomes much smaller than the original one and hence can be inlined under the current ORC inlining heuristics.

There are many similar examples of large but infrequently executed code in an otherwise hot function. For example, the sqrt() library function in Digital Unix 3.2 on DEC Alpha workstations contains a 'hot' spot of two basic blocks containing 69 instructions out of fourteen basic blocks of 205 instructions[11]. 'Hot' spot covers only 8.1% of the code in the BSD version of the TCP network protocol implementation[11]. Mosberger *et al.* reported system software that contains up to 50% error handling code[10]. As the profitability for inlining a program unit is often dependent on the size of the program unit, a program unit may contain a very small 'hot' spot that is profitable to be inlined, but the inlining is prevented by the size of the cold portion.

The advantages of outlining are two-fold:

Enabling Inlining. A hot program unit whose code region was outlined might become small enough to enable its inlining. This size reduction applies to both caller and callee: either a large caller or a large callee increases the negative effects of inlining. As we demonstrated in our inlining study, even after adaptive inlining and cycle density analysis are added, there are still very frequent call sites that cannot be inlined because the caller, the callee, or both together, are just too large [15]. The inlining criteria in most current compilers simply decide to not inline large callees or large callers because of their negative effects on executable size, compilation time, and performance. I propose to use outlining (*i.e.* function splitting) to split a large program unit into smaller ones so that more aggressive inlining is possible.

Improving Locality. The second advantage of outlining is instruction cache efficiency and instruction fetch bandwidth. Without outlining, infrequently and frequently executed statements are mixed together. They will interfere with each other. Cold statements might pollute the cache by evicting hot statements from the cache. Moreover, modern superscalar and VLIW architectures demand high instruction bandwidth of the memory hierarchy because a sufficient number of useful instructions must be fetched into the cache to allow the functional units in the processors to be fully utilized. Some researchers found that the bottleneck of instruction bandwidth keeps almost 70% of the CPU cycles idle[10]. Mixing the cold statements with hot statements definitely contributes to this problem: when a hot statement is executed, some cold statements are also fetched into the cache. But the cold statements might never be executed at all, thus reducing the effective instruction fetch bandwidth.

The negative performance impact of outlining is that extra function calls are introduced: the control flow between the region and the other parts of the program unit now is transformed into function calls. This cost must be taken into consideration when applying the outlining transformation.

2 Function Splitting

In this appendix I report the progress that I have made on the implementation of outlining. I have finished the work on inlining tuning, implementation feasibility analysis, benchmark study and analysis for outlining opportunities, region identification by analyzing structured control flow constructs, and function splitting implementation.

I briefly introduce our inlining tuning experience in section 2.1. In section 2.2 we discuss about our decision on where we should insert our outlining phase in ORC. I discuss hazardous program units that should not be outlined in section 2.3. Section 2.4 presents the details of function splitting.

2.1 Inlining tuning

Partial inlining on high level intermediate representation is motivated by our inlining tuning experience in ORC [15]. In our inlining tuning work, we introduced two new heuristics to enhance the ORC inlining decisions: *adaptation* and *cycle_density*. ORC uses a *temperature* heuristics to calculate the potential benefit of inlining a call site. With *adaptation* we are allowed to vary the temperature threshold so that more aggressive inlining is applied to small benchmarks. The motivation for this heuristic is the vulnerability of large benchmarks to the negative effects of inlining. With *cycle_density* we prevent the inlining of procedures that have a high temperature in spite of being called infrequently.

Our empirical study of inlining suggests two major reasons that prevent some very frequently invoked call sites from being inlined: recursive function calls and large function body of the caller or the callee. My proposal to implement outlining to enable more aggressive inlining (*i.e.* partial inlining) is motivated by this observation.

2.2 Where should outlining occurs

Because I want to use outlining to enable more aggressive inlining, the outlining should occur before the inlining analysis. Looking back at the Figure **??**, IPO (including inlining analysis) starts from IPL. Thus, naturally, outlining must be implemented at the beginning of IPL phase. Some important implementation decisions are shaped by the characteristics of IPL phase.

IPL is an early phase that just follows the front end of the compiler and few transformations have been performed before IPL. At the beginning of IPL, the intermediate representation is very high level WHIRL tree which contains high level control flow structures. One good example for the high level control flow structure is the $SWITCH$ WHIRL node which represents the switch-case statement in high level programming languages.

We can do the outlining optimization either on CFG or WHIRL tree. I have tried to perform outlining transformation on CFG, but this turns out to be very difficult and we gave up this effort. In the CFG-based optimization, we need to build CFG from very high level WHIRL tree; annotate the CFG with profiling information; do outlining optimization on the CFG; and finally we have to translate (also called emit) the CFG back to very high level WHIRL tree, which is expected by the following components in the compiler. Moreover, the CFG construction is complicated by that fact that the CFG needs to be converted back to high level WHIRL tree. This is because WHIRL tree in this stage contains a lot of high level control flow constructs (*e.g.* loop and switch) which are essential for other middle-end optimization (*e.g.* nest loop optimizations and switch-case statement optimizations). To ensure the WHIRL tree retains the high level control flow constructs after emission, quite some information needs to be annotated in the CFG. This complicates the whole process. Without decent emission, all the later optimization components are negatively impacted. If the intermediate representation is not what an optimization component expects, the optimization opportunity can be lost or, even worse, the compiler will refuse to work. Thus, it is very difficult to seamlessly integrate the CFG-based outlining with existent optimizations in ORC.

On the contrary, as we will show in this work, very high level WHIRL tree has enough information for outlining analysis and direct WHIRL tree manipulation is more straightforward, concise and effective.

2.3 Hazardous program units for outlining

Like in inlining, we try to avoid outlining for some program units out of performance or semantic correctness consideration.

Trivial functions I don't consider trivial functions in our outlining analysis. Trivial functions are those program units that is infrequently invoked and do not contain loops with large trip count. The outlining analysis does not consider trivial functions because the performance impact of these functions are negligible.

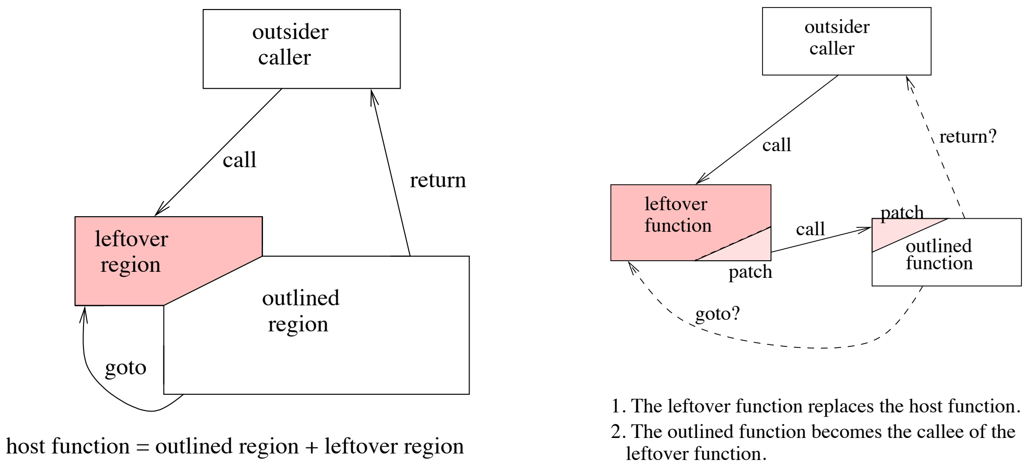
Small regions I hope to use outlining to reduce the size of program unit that contains frequently executed code. However, outlining small regions might achieve the contrary effect.
When splitting a chunk of code out of the original program unit, often we need to pass the accessed variables in the code chunk as parameters to the newly generated program unit. If the to-be-split chunk of code contains *return* statement or *goto* statement which jumps to a label in the intact code of the original program unit, the compiler will have to insert a special code stub in the original program unit to *return* from the program unit or *goto* to the intended label. The parameter passing code and special code stubs are called "patches" in the outlining analysis. It is possible that the size of the introduced "patches" exceeds the to-be-split region. This kind of outlining should be strictly avoided because we would fail to reduce the size of the original program unit.

Regions with escaped *alloca*-allocated memory *Alloca* allocates memory space in the stack frame of the caller. The memory allocated in the stack will be freed automatically when the caller returns. *Alloca* is usually used for local variables in a program unit. Thus the programmers can dynamically allocate memory without worrying about when to release them.
When a program unit uses *alloca* to allocate memory in a region and accesses the allocated memory out of the region, the region should not be outlined. This is because the newly generated program unit would allocate a memory block with *alloca* and pass this block to the original program unit. It would difficult to maintain the original semantics of the program because the memory allocated in the new program unit would be automatically freed at its exit and would be no longer valid in the original program unit. Thus the operations on the *alloca*-allocated memory are actually accessing invalid memory.

2.4 Implementation of function splitting in ORC

In this discussion of outlining I adopt the following terminology: f_{host} (***host function***) is the original program unit in which a region is selected for outlining;

R_{out} (***outlined region***) is the region within f_{host} that is selected for outlining and $R_{leftover}$ (***leftover region***) is f_{host} excluding R_{out} (Figure 4.a); f_{caller} (***outsider caller***) is a function that calls f_{host}; f_{out} (***outlined function***) is the new function that is generated by the outlining process to contain R_{out} and $f_{leftover}$ (***leftover function***) is the original program unit after R_{out} is split out of f_{host}. After outlining (Figure 4.b), f_{out} becomes the callee of $f_{leftover}$ and f_{host} is replaced by $f_{leftover}$ in the call chain ($f_{leftover}$ inherits all the original resources (including the function name, patched WHIRL tree and symbol table).



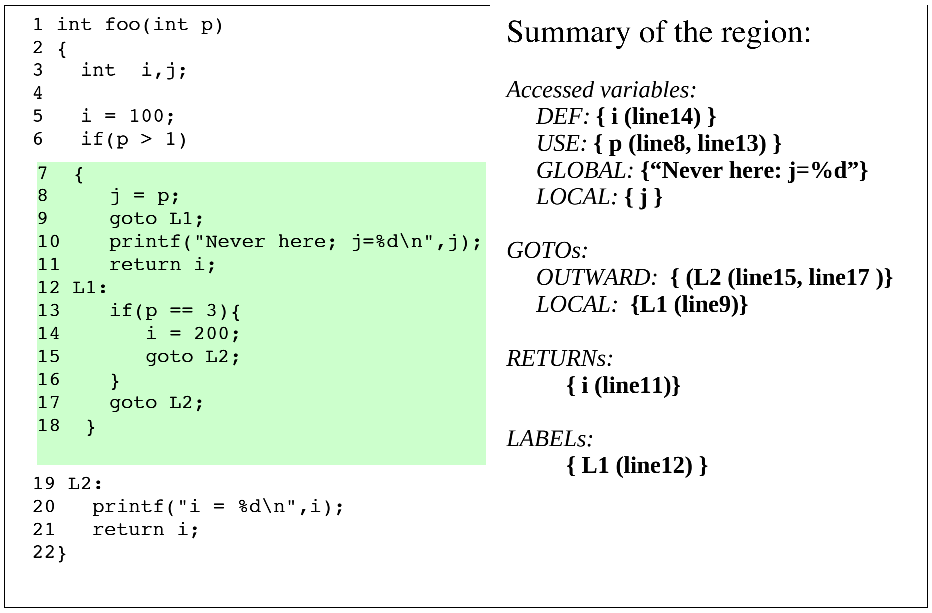
outsider caller
call
return
leftover region
outlined region
goto
host function = outlined region + leftover region
outsider caller
call
return?
leftover function
call
patch
patch
outlined function
goto?
1. The leftover function replaces the host function.
2. The outlined function becomes the callee of the leftover function.

(a) Before outlining (b) After outlining

Fig. 4. Outlining transformation

The outlining transformation consists of four major phases: *region identification* selects R_{out} regions to be split out of the original program unit f_{host}. *Summarizing* collects information that is needed by function splitting. *Callee generation* generates the f_{out} program unit for the outlined region R_{out}. *Caller patching* eliminates the split code and inserts compensation code in f_{host} to conserve the correct semantics. I will use the function foo in Figure 5 as an example of an f_{host} program unit to demonstrate the outlining process.

Naive region identification algorithm Currently the analysis identifies regions based only on structured control flow constructs. Examining the annotated WHIRL tree, this analysis can detect the infrequently executed *then* block and *else* block in hot functions. Let's assume that the identification algorithm locates the shadowed code in Figure 5 as the region to be split (*i.e.* R_{out}).



```
1 int foo(int p)
2 {
3   int  i,j;
4
5   i = 100;
6   if(p > 1)
7  {
8     j = p;
9     goto L1;
10    printf("Never here; j=%d\n",j);
11    return i;
12 L1:
13    if(p == 3){
14       i = 200;
15       goto L2;
16    }
17    goto L2;
18  }
19 L2:
20   printf("i = %d\n",i);
21   return i;
22}
```

Summary of the region:

Accessed variables:
DEF: **{ i (line14) }**
USE: **{ p (line8, line13) }**
GLOBAL: **{“Never here: j=%d”}**
LOCAL: **{ j }**

GOTOs:
OUTWARD: **{ (L2 (line15, line17)}**
LOCAL: **{L1 (line9)}**

RETURNs:
{ i (line11)}

LABELs:
{ L1 (line12) }

Fig. 5. Function foo before function splitting

Summarizing The summary information of a region is used in the outlining transformation. This step goes through the WHIRL representation of the region and collect information including:

Variable access information There are two kinds of variable accesses: *use* and *definition*. A variable's use reads the value of the variable and a variable's definition writes a new value to the variable. For example, in the statement $a = x + y$, variable x and y are used and variable a is defined. The variable access information is used to determine the variables that are needed to be passed to the new program unit.[5]

There is no need to collect access information for global variables because they can be seen by all the functions, including the newly generated program unit.

The right column of Figure 5 lists the variables. In this case, the first parameter (a constant string) to the $printf$ function call is a global symbol and is accessible by all the functions, it does not need to be passed as a parameter to f_{out}. Note that the variable j is never accessed outside of R_{out}. Thus, j

[5] The study of which variables contain values of interest for the execution of the program is known as *liveness analysis* in compilers. A variable is *live* if it contains a value that might be used anywhere in the program in the future. If an analysis can prove that the value stored in a variable cannot be used in the future, then the variable is *dead*.

can be converted to a local variable in f_{out} to avoid the overhead of passing it as parameter.

Local label information We collect all the labels that appear in R_{out}. This is because labels are also symbols in a function. After the outlining transformation, the labels in f_{host} are converted to labels in f_{out}. In Figure 5, there is one local label $L1$ in R_{out}.

***Goto* and *return* information** We divide *goto* statements in R_{out} into two categories: outward *goto*s and intra-regional *goto*s. Outward *goto*s are the *goto* statements that jumps to a label in $R_{leftover}$. A *goto* statement in R_{out} that jumps to a label also in R_{out} is an intra-regional *goto*s.
It is easy to tell whether a *goto* statement is outward or intra-regional by checking whether the destination of the *goto* falls in the local label list.
Return statement information is used to insert compensation code in f_{host} and f_{out} so that when *return* is executed, f_{out} returns to f_{caller} (*i.e.* the caller of the f_{host}).

These information is also listed in Figure 5 shows the *goto*s, *return*s and *label*s found in the R_{out} region of foo.

Outline the region This step involves generating f_{out} based on R_{out}. Figure 6 shows $f_{leftover}$ (*i.e.* the original foo after outlining) and Figure 7 shows the new function f_{out} :$fooNEW1$. Function splitting needs to perform the following tasks:

Construction of f_{out} and its symbol table. The compiler needs to build a valid WHIRL tree and its symbol table for f_{out}. This involves building an empty WHIRL tree first and then cloning the WHIRL tree of R_{out} into the empty WHIRL tree. The generated WHIRL tree is still not valid because both f_{host} and f_{out} need to be repaired to interact properly. Also, the symbol table for f_{out} is initialized.

Patching the variable accessing. Because R_{out} is only a part of the original program unit, all the variables accessed in R_{out} are within the scope of f_{host}. Thus, when R_{out} is transformed into f_{out}, the scope of the variable accesses must be modified to access the correct memory location.
In very high level WHIRL representation, variable accesses fall into four categories:

- Load (LOAD) a variable
- Store (STORE) a variable
- Load address (LDA) of a variable (*i.e.* address taking)
- Indirect load and store (ILOAD and ISTORE)

In the WHIRL representation, ILOAD and ISTORE never directly access a variable in the program. Instead, their operand is a LOAD statement or another ILOAD statement. Therefore, we only need to take care of LOAD, STORE and LDA cases.
Currently, if a local variable v is accessed in both R_{out} and $R_{leftover}$, we need to pass v as a parameter to f_{out}. In Figure 5, we only need to pass

```
int foo(int p)
{
  int  i,j;
  int ReturnFlag, ReturnValue;
  i = 100;
  if(p > 1)
  {
     fooNEW1(&ReturnFlag, &ReturnValue, &i, p);

     if (ReturnFlag != 0) {
         if (ReturnFlag == 1)
            return (ReturnValue);
         else
            COMPGOTO(ReturnFlag-2,{L2} );
     }
   }
 L2:
   printf("i = %d\n",i);
   return i;
 }
```

Fig. 6. After function splitting (the original $foo()$)

```
void
fooNEW1 (int *ReturnFlag, int *ReturnValue, int *iNEW, int pNEW)
{
     int  j;
     *ReturnFlag = 0;
     j = pNEW;
     goto L1NEW;
     printf("Never here; j = %d\n", j);
     *ReturnValue = *iNEW;         // return I;
     *ReturnFlag = 1
     return;
L1NEW:
     if( pNEW == 3 ){
         *iNEW = 200;              // i = 200;
         *ReturnFlag = ( 0 + 2 ); // goto L2
         return;
     }
     *ReturnFlag = ( 0 + 2 );     // goto L2
     return;
 }
}
```

Fig. 7. After function splitting (the new PU $fooNEW1()$)

the variables in the DEF and USE groups. A variable can be passed by its value or by its address, depending on whether it is ever defined in R_{out}. If the variable is ever defined in R_{out}, we place it in the DEF group and pass its address to f_{out}. Otherwise it is in the USE group and its value is passed. This parameter passing scheme can be improved by promoting shared variables to global variables. After such promotion, there would be no need to pass these variables as parameter. However, we still do not have a clear understanding of the impact of such promotions on performance. For example, how many global variables will be introduced? What is the impact of these additional global variables on the register allocator? Answering these questions will require further empirical and analytical study of the compiler. Because the original WHIRL tree accesses the variables directly, some variable access nodes in the WHIRL tree of the region must be patched according to the parameter passing scheme used for f_{out}. Table 1 lists the data access patching rules. In this table, $\boldsymbol{a}$ is the variable in f_{host}, $\boldsymbol{A}$ is the parameter to f_{out}, "–" represents an invalid situation that should not appear in a correct transformation. The binding column tells how the variable a should be binded to the formal parameter A. The first row is the original variable access statement in the region. The second and the third rows list the transformation needed when a variable is passed by is value and by its address, respectively.

	passing method	binding	LOAD $\boldsymbol{a}$	STORE $\boldsymbol{a}$	LDA $\boldsymbol{a}$
USE	value ($\boldsymbol{a}$)	$\boldsymbol{A} = \boldsymbol{a}$	LOAD $\boldsymbol{A}$	–	–
DEF	address (&$\boldsymbol{a}$)	$\boldsymbol{A} = \&\boldsymbol{a}$	ILOAD $\boldsymbol{A}$	ISTORE $\boldsymbol{A}$	LOAD $\boldsymbol{A}$

Table 1. Variable patching rule

ORC puts some variables or intermediate results in pseudo registers. One important feature of a pseudo register is that the variable stored in such a register is not aliased to anything. In the register allocation phase, a pseudo register can be promoted to a physical registers. Pseudo registers that fail to be mapped to real registers are demoted back to memory variables. The very high level WHIRL tree contains referenced to pseudo registers. A pseudo register does not have an address and thus cannot be passed by its address. Therefore, if a pseudo register falls into the DEF group of R_{out} and is also accessed in $R_{leftover}$, we cannot simply pass its address to f_{out}. We have to demote the pseudo register to a memory variable and pass the memory variable by reference. This kind of demoting probably have negative impact on performance. I plan to investigate such impact to decide if it should be included in outlining heuristics for a region.

Building local labels and local *goto* statements The compiler converts the labels in R_{out} to local labels in f_{out}. Essentially, it generates a new label in the local symbol table of f_{out} for each label in R_{out} and modifies every intra-regional *goto* statement to jump to the respective new label. In Figure 7, the original label $L1$ in foo is changed to a local label $L1NEW$ in function $fooNEW1$, and the original intra-regional *goto* statement is modified to point to label $L1NEW$ (Line 7 and 12 in Figure 7).

Function exit handling Function f_{out} returns to $f_{leftover}$ in one of three ways:[6]

1. The control naturally falls through the new function and returns to $R_{leftover}$. This kind of returning implies that the next operation executed upon the return is the WHIRL node next to R_{out} in the WHIRL tree of f_{host}.
2. A *return* statement is executed. Originally, the *return* statement returns from f_{host} to its caller f_{caller}. After the outlining, f_{out} becomes the callee of $f_{leftover}$. Therefore, we need to develop a mechanism that returns from f_{out} to f_{caller}.
3. An outward *goto* statement is executed. An outward *goto* means that the control needs to be directed from f_{out} to the specified label in $f_{leftover}$.

Though it is an optimization option, currently architecture-specific stack manipulation techniques are not used to achieve inter-procedural *goto* and multiple level *return*. Instead, I decided to implement these transfers of control in a more traditional way by creating two new symbols (variables) in the original program unit: $ReturnFlag$ and $ReturnValue$ (Line 4 in Figure 6). The addresses of these symbols are both passed to f_{out} as parameters. The integer variable $ReturnFlag$ is set by f_{out} as a flag to specify the action that should be taken by $f_{leftover}$ upon the return of f_{out}.

$ReturnFlag$	Action
0	fall through
1	return $ReturnValue$
≥ 2	computed goto (*ReturnFlag*-2, JUMPTABLE)

Table 2. Semantics of $ReturnFlag$ on the return of the new PU

Table 2 shows $f_{leftover}$'s action according to $ReturnFlag$ on f_{out}'s return. When $ReturnFlag$ is 0, the next instruction to R_{out} is executed. When $ReturnFlag$ equals 1, the $f_{leftover}$ returns to its caller immediately (*i.e.* f_{out} returns to its caller's caller).

In $f_{leftover}$, we build one jump table for each region to be split. For a region R_{out}, its jump table contains the destination labels of all its outward *goto*s. When $ReturnFlag$ is greater than 1 on f_{out}'s return, $(ReturnFlag - 2)$ is an index to R_{out}'s jump table Thus we can use a computed goto statement to direct the control to the proper label.

For example, the region in function foo contains only two outward *goto*s and both of them point to label $L2$. Thus, the jump table contains only one

[6] A program unit can also return by the *exit*() function call, but it will exit the application and we do not need to worry about it.

element $L2$. In the new function (Line 15 and 18 in Figure 7), the original *goto* $L2$ is replaced with a statement that stores $(0+2)$ into the $ReturnFlag$ (0 is the index to the jump table and 2 is a constant that skips over the other two return methods). In the original program unit, we get the proper jump table index (0) by $(ReturnFlag - 2)$. Thus, the control is directed to the first label in the jump table: $L2$ (Line 13 and 14 in Figure 6).
The second new symbol $ReturnValue$ is used to handle regions with $return$ statement. $ReturnValue$ is a variable of the return type of the original program unit. When f_{out} needs to return a value, the new function doesn't return the value directly to $f_{leftover}$. Instead, it saves the return value in $ReturnValue$ and set the $ReturnFlag$ with 1. When $ReturnFlag$ is 1 on the f_{out}'s return, the $f_{leftover}$ should directly return the value stored in variable $ReturnValue$ to f_{caller} (Line 12 in Figure 6).

Put f_{out} into the compiler control. The compiler compiles only one program unit at a time. After outlining, f_{out} has to be placed in the background so that the compiler can proceed with the compilation of $f_{leftover}$. ORC maintains a list of program units to be compiled. Thus, a control block of f_{out} (including its WHIRL tree and symbol table information) is inserted into this list, waiting for its turn to be compiled.

3 Related Work

Function outlining has been presented by several authors. In their famous code positioning work[12], Pettis and Hansen separate the frequently executed code and the infrequently executed code in a program unit. The transition between the frequently executed code and the cold code is achieved by explicit jump instructions (if the two parts are located too far away from each other, code stub that relays jumping need to be inserted). The motivation for their function splitting is to make the primary function (hot code) as small as possible so that important related code can co-exist in the instruction cache or be placed in the same memory page. Similar function splitting approaches appear in the work of Mosberger *et al.* and Castelluccia *et al.* [10, 2]. However, Mosberger and Castelluccia only tried to use function splitting to improve the code density of network protocol code. None of these approaches to function splitting have the goal of enabling aggressive inlining.

Muth and Debray proposed to implement partial inlining in a link-time optimizer called ALTO [11]. They generate a new program unit to hold all the split code. Moreover, once the control enters the newly generated code (the cold code), it will not return even if it touches the hot code again. This implies that usually the new program unit has to clone both the cold code and all the frequently invoked code that can be reached from the cold code, worsening the code bloat problem. Moreover, duplicating all the code that can be reached by the cold WHIRL tree would more likely increase the parameter passing overhead.

In the work of Muth *et al.* , the performance improvement from outlining is barely seen. The most important reason is probably that outlining occurs at the link time (on object files) or after the link time (on binary executables), which are too late in their frameworks. Very few optimizations occur after linking. Thus placing partial inlining at or after link-time cannot increase the scope of some powerful optimizations that have occurred in earlier phases. Moreover, all the above work have dealt exclusively with hot functions.

My work is different from their work in the following aspects:

– My outlining occurs in the very beginning of the middle-end optimization. This enables aggressive inlining and will benefit all the middle-end and back-end optimizations.
– I have different outlining strategies for hot functions (the functions that are invoked frequently) and heavy functions (the functions that are infrequently invoked but contains loops with large trip count). We emphasize heavy functions because our inlining tuning experiences show that heavy callers also prevent aggressive inlining.
– Building a high-level outlining framework, I use tree-based analysis and a series of pre-processing to ensure that the analysis finds beneficial regions for outlining. Thus, the engineering problem is also new compared to the existent work.

Way *et al.* made some initial experiments on partial inlining that occurs on high level intermediate representation[14]. They viewed inlining as an enabling technique to build inter-procedural region, which is considered to be important for taming the compilation optimization cost. As a by-product of their CFG-based inter-procedural formation work, they manually find the infrequently invoked portion of a program and then partially clone the cold code into a new function (also manually). Therefore the cold code can be replaced with a call to the new function. Their partial inlining concepts are most similar with ours. However, the two most important components of partial inlining are done by hand in their work, which is almost impossible in real life and less convincing for its feasibility. Instead, we propose tree-based partial inlining and some enabling pre-processing techniques to make partial inlining possible in a product-strenth compiler.

4 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, 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).

References

1. Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive inlining. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 134–145, May 1997.
2. C. Castelluccia, Walid Dabbous, and Sean O'Malley. Generating efficient protocol code from an abstract specification. In *ACM SIGCOMM*, pages 60–72, 1996.
3. Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software - Practice and Experience*, 22(5):349–369, 1992.
4. 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.
5. Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Software - Practice and Experience (SPE)*, 18(8):775–790, 1989.
6. Jack W. Davidson and Anne M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering (TSE)*, 18(2):89–102, 1992.
7. David Detlefs and Ole Agesen. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming (ECOOP)*, pages 258–278, June 1999.
8. Rainer Leupers and Peter Marwedel. Function inlining under code size constraints for embedded processors. In *International Conference on Computer-Aided Design (ICCAD)*, pages 253–256, Nov 1999.
9. Wen mei W. Hwu and P. P. Chang. Inline function expansion for compiling realistic C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 246–257, 1989.
10. 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.
11. Robert Muth and Saumra Debray. Partial inlining. Technical report, Dept. of Computer Science, Univ. of Arizona, U.S.A., 1997.
12. Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, 1990.
13. Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. An empirical study of method inlining for a Java just-in-time compiler. In *2nd Java Virtual Machine Research and Technology Symposium (JVM '02)*, pages 91–104, Aug 2002.
14. Thomas Way. *Procedure Restructuring for Ambitious Optimization.* PhD thesis, University of Delaware, May 2002.
15. 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.