Function Outlining and Partial Inlining

Peng Zhao
IBM Toronto Laboratory
Markham, ON, Canada

José Nelson Amaral*
Department of Computing Science,
University of Alberta, Edmonton, Canada

Abstract

Frequently invoked large functions are common in non-numeric applications. These large functions present challenges to modern compilers not only because they require more time and resources at compilation time, but also because they may prevent optimizations such as function inlining. However, usually it is the case that large portions of the code in a hot function f_{host} are executed much less frequently than f_{host} itself. Partial inlining is a natural solution to the problems caused by including cold code segments that are seldom executed into hot functions that are frequently invoked. When applying partial inlining, a compiler outlines cold statements from a hot function f_{host}. After outlining, f_{host} becomes smaller and thus can be easily inlined. This paper presents a framework for function outlining and partial inlining that includes several innovations: (1) an abstract-syntax-tree-based analysis and transformation to form cold regions for outlining; (2) a set of ¤exible heuristics to control the aggressiveness of function outlining; (3) several possible function outlining strategies; (4) alias agent*, a new technique that overcomes negative side-effects of function outlining. With the proper strategy, partial inlining improves performance by up to 5.75%. A performance study also suggests that partial inlining is not effective on enabling more aggressive inlining. The performance improvement from partial inlining actually comes from better code placement and better code generation.*

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* [2, 3, 16].

There are many examples of large but infrequently executed code in hot functions [7, 8]. For instance, only 8.1% of the code in the BSD version of the TCP network protocol implementation is hot[8]. Function outlining is a technique that splits a region into a new, independent function f_{out} and replaces the region with a function call to f_{out}. Outlining has obvious performance potential because it might enable more inlining and improve code locality. 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. This cost must be taken into consideration when deciding to apply the outlining transformation. This paper makes the following contributions:

- An abstract-syntax-tree-based analysis and a set of heuristics to form and identify cold regions in hot functions. A performance study indicates that this region formation method is successful.

- Unlike previous outlining work, the outlining analysis presented in this paper happens very early in the backend. Early optimizations may negatively impact existing downstream optimizations. A major negative impact of outlining is on the alias analysis of the new parameters passed to the outlined functions. A novel technique, *explicit variable disambiguation*, prevents the spilling variables in hot paths.

- A performance study of two orthogonal outlining strategies (collective VS. independent splitting and splitting with VS. without explicit disambiguation) reveals that choosing the correct strategy is crucial. While combining independent splitting and alias

*This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada.

agent improves performance, other strategy combinations may signi£cantly deteriorate performance.

- Partial inlining improves benchmarks in the SPEC2000 integer suite by up to 5.75%. These improvements are due to better code placement and better code generation. The effect of partial inlining on aggressive inlining is much less pronounced than was expected by many in the optimizing compiler community.

Section 2 introduces WHIRL, the intermediate representation used in the Open Research Compiler (ORC). The Section 3 presents the design and implementation of outlining and partial inlining. Section 4 discusses the experimental study. Related work is discussed in Section 5.

2 WHIRL Trees

Outlining on very-high WHIRL bene£ts from high-level, structured, control-¤ow constructs — such as *if*, *loop* and *switch*. In contrast at lower level control ¤ow is represented by ¤at constructs —- such as conditional branches and *goto*s. Thus, high-level outlining can identify cold code segments in a single pass through the WHIRL tree. The contrived function shown in Figure 1, HOTPU, illustrates the WHIRL-tree representation. Statements are annotated with their execution frequency obtained from runtime pro£ling. Assume that HOTPU is frequently invoked. The shaded code segments or nodes are the cold parts of HOTPU. In very high WHIRL, three control-¤ow constructs may lead to cold code in a hot function:

if* statement.** An ***if node in a WHIRL tree has two children: a ***then*** block and an ***else*** block. These nodes are annotated with execution frequency. 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, the CG_n node is executed and the program jumps to the A_n node that contains the action code for case n. If the switch expression is not equal to any of the enumerated cases, the program jumps to the A_d node that contains the default action through the DG node. Feedback information indicates the execution frequency of each case. Often large switch statements have skewed execution frequency distribution [18]. 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. Usually a hot early return implies that the rest of the function is cold. In Figure 1, there are three early returns at lines 12, 15 and 18 that correspond to nodes $return0$, $return1$ and $return2$ in the WHIRL tree. However, only $return2$ at line 18 is hot.

2.1 Region

In this paper, a region is a sequence of code in the program that is guarded by a high-level control-¤ow construct such as *if* and loop statements (see Figure 1). 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.

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 non-executed *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:

$$SC(er,\ R) \quad = \quad \{s \mid A = NCA(er, s) \wedge Pos(s, A) > Pos(er, A)\}$$

where $NCA(er, s)$ is the *nearest common ancestor* of er and s, and $Pos(s, A)$ is the position of s, or one of its ancestors, in A. For a formal de£nition refer to [15]. 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 to split cold and hot code into separated regions; *candidate identi£cation* identi£es regions for which outlining is bene£cial; *function splitting* generates a new function from a candidate region and replaces the region with a call to the new function.

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. The splitting of switch statements is described in [18].

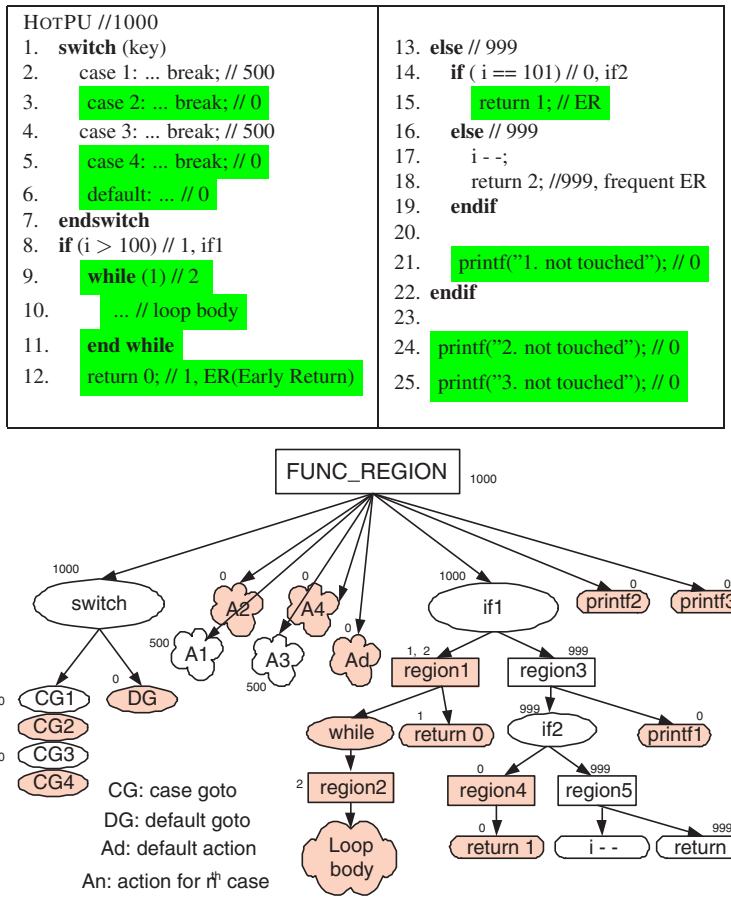

Figure 1. Example source code & WHIRL tree

HANDLEER (S_{er})
1. $ReturnFreq \leftarrow ReturnFreq + GetFreq(S_{er})$
2. **if** $\left(\frac{ReturnFreq}{GetFreq(HostFunc)} \leqslant ERThreshold\right)$
3. **return**
4. $S_{loop} \leftarrow S_{er}$
5. $CurParent \leftarrow GetParent(S_{er})$
6. **while** $(CurParent \neq ROOT)$
7. **if** ($CurParent$ is a loop construct)
8. $S_{loop} \leftarrow CurParent$
9. $CurParent \leftarrow GetParent(CurParent)$
10. $CurNode \leftarrow S_{loop}$
11. **while** $(CurNode \neq ROOT)$
12. $CurParent \leftarrow GetParent(CurNode)$
13. **if** ($CurParent$ is a region)
14. EXTRACTCODE$\left(SC(CurNode, CurParent)\right)$
15. $CurNode \leftarrow CurParent$

Figure 2. Handling early returns

3.1 Handling Frequent Early Returns

The algorithm HANDLEER, shown in Figure 2, handles early returns. HANDLEER is called when an early return statement S_{er} is encountered during the depth-first traversal 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, code from different levels 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).

3.2 Candidates for Outlining

Every region R is annotated with $(freq_R, size_R)$. The $size_R$ of a region is the number of WHIRL nodes in that region and $freq_R$ is its frequency. Next the compiler identi£es cold regions that are suitable for outlining.

3.2.1 Hazardous regions for outlining

Some regions are not outlined to prevent performance degradation or to preserve program correctness. Outlining is avoided in two following situations. 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, defeating our purpose of reducing function sizes. Outlining also avoids 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 dif£cult 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 Selective Outlining

The optimal outlining problem can be reduced from a 0-1 knapsack problem, which is NP-hard [15]. Therefore heuristics to £nd sub-optimal solutions in reasonable time are required.

An effective greedy algorithm for the *knapsack problem* is used to estimate the bene£ts of splitting a region R out of its host function F. This algorithm uses the $freq\_ratio$ and $size\_ratio$ to model the pro£t and cost for outlining a region R.

$$size\_ratio_R = \frac{size_R}{size_F} \quad (1)$$

$$freq\_ratio_R = \frac{freq_R}{freq_F} \quad (2)$$

$$benefit_R = \frac{size\_ratio_R}{freq\_ratio_R} \quad (3)$$

Essentially, $size\_ratio$ and $freq\_ratio$ are the contribution of the cold regions to the total size ($size_F$) and execution frequency ($freq_F$) of the host function. Therefore, the *bene£t* heuristics favors large cold regions. Intuitively, the less frequent a region and the larger its size, the more bene£cial to split it out of the host function. A region is outlined if its outlining bene£t, $benefit_R$, exceeds a carefully chosen *Bene£tThreshold*.

To avoid a situation in which the patch code is larger than the outlined region, there is a threshold for the size of a region R to be outlined: $size_R > SizeThreshold$.

3.3 Function Splitting

The splitting transformation consists of generating a new function f_{out} from the region R_{out} to be outlined and patching the data ¤ow and control ¤ow of both f_{host} and f_{out} so that the program's semantics is left intact. Function splitting is described in [17].

3.4 Performance Tuning

Proper thresholds for outlining bene£t and cold region size are important. After experimentation with a large set of thresholds the values of 1000 for the bene£t threshold and 10 for the cold region sizes were selected. This section describes some important performance tuning, based on different strategies, for the outlining optimization.

3.4.1 Independent VS. Collective outlining

Regions to be outlined may be scattered throughout f_{host}. The *independent outlining* strategy splits each region into separate functions. The *collective strategy* moves all the outlining candidates to a single combined region and splits it into a single function. Some handshaking work is required for both the host function and the split function. At the beginning of f_{out}, the control ¤ow is dispatched to the correct sub-region according to a *Flag* parameter. On the host function, each outlining candidate is replaced by an initialization of *Flag* and a jump to the patch code. Therefore the collective strategy generates a single new call site in the host function. The changes to the CFG introduced by the collective strategy may be troublesome for downstream compiler analysis. On the other hand the collective strategy may produce smaller patch code because all the sub-regions now share the same patch code. But this also means that there would be some unnecessary parameter passing for each speci£c call to the new function. Section 4 shows that independent outlining performs better than collective outlining.

3.4.2 Explicit Variable Disambiguation

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. A consequence of imprecise alias information is that a variable that was kept in a register must now be spilled. This constitutes a performance hazard in ORC 2.1 because the memory spills is often placed in a hot path. Our solution is to introduce an *explicit variable disambiguation* technique to eliminate this serious side-effect. Each variable v whose address is passed to f_{out} has a clone v' 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. Both copies occur in the same cold basic block that contains an invocation to f_{out}.

3.4.3 Partial Inlining

In ORC, function outlining occurs before the IPO phase, which includes function inlining optimization. Function outlining analysis is conducted on each function and regions are outlined accordingly. After all the functions are processed, the outlining phase ends and all the source £les are passed to the IPO phase. Partial inlining is achieved by enabling both function outlining and inlining.

4 Results

An experimental investigation on SPEC2000int benchmarks reveals that:

- Outlining reduces the size of hot functions by up to 97% and incurs less than 0.21% increase in run-time function calls.
- Explicit variable disambiguation combined with independent splitting is the best outlining strategy. Explicit disambiguation is crucial to avoid performance degradation caused by memory spills in hot paths.
- Surprisingly, function outlining has less effect on enabling aggressive inlining than previously expected. Performance improvement from function outlining alone ranges from -0.62% to 4.1%. When partial inlining is enabled, performance increases range from -0.85% to 5.75%.

4.1 Experiment Con£guration

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.[1] All the pro£ling information is obtained by using standard SPEC2000 training data set and the reported run-time data is from the standard reference data set. 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.

4.2 Function Outlining Performance

Figure 3 shows the performance changes caused by the four strategies described in Table 1. Combining explicit variable disambiguation with independent splitting, *D-I*, usually outperforms the other strategies. Noticeable performance improvements are observed on `gap`(1.1%), `gcc`(1.2%), and `perlbmk`(4.1%). When local variables are not disambiguated from their clones passed as parameters (*N-I* and *N-C*) signi£cant performance degradation occurs. Inspection of binaries indicates that the imprecision of the ORC 2.1 alias analysis results in many additional memory spills in hot paths. Collective splitting (*N-C* and *D-C*) also degrade performance because of the adverse effects of sharing patch code.

4.3 Outlining Statistics

Table 2 presents static measurements for *D-I*. The £rst row shows that function outlining occurs in many places in `gap`, `gcc`, `perlbmk` and `vortex`. Small benchmarks such as `bzip2`, `gzip` and `mcf`, have fewer regions split. *If* statements are the major source of cold regions in hot functions. The *Function Size Reduction* row shows that outlining reduces function sizes drastically (up to 97%). When the patching code is larger than the split code, outlining enlarges functions. The number of parameters needed for the outlined functions ranges from 2 to 19.

Function outlining increases the number of run-time function calls by at most 0.21%. This indicates that (1) the heuristics successfully avoid outlining hot regions and (2) the SPEC2000int training data set is representative of the reference data set.

[1] We don't include `eon` because our compiler cannot compile it successfully.

Short	Explanation
D-I	Explicit variable disambiguation and independent splitting.
N-I	No explicit variable disambiguation and independent splitting.
D-C	Explicit variable disambiguation and collective splitting.
N-C	No explicit variable disambiguation and collective splitting.

Table 1. Strategy Combinations

Benchmarks		bzip2	crafty	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr
Number of Regions Split		5	59	192	388	1	3	36	415	4	410	61
Control	if	5	54	186	354	1	3	36	374	3	398	61
Flow	switch	0	5	0	12	0	0	0	33	0	7	0
Construct	early exit	0	0	6	22	0	0	0	8	1	5	0
Function	min (%)	-14	1	-63	-37	35	7	-5	-23	3	-35	26
Size	max(%)	17	1	97	84	35	15	26	89	23	71	55
Reduction	avge(%)	1.5	9.8	19.7	15.6	34.5	11.0	16.1	21.6	9.6	18.2	7.0
Number of	min	2	2	2	2	2	3	2	2	3	2	2
Parameters	max	2	6	8	19	2	5	6	16	5	18	9
Passed	average	2.0	4.8	5.5	8.5	2.0	4.3	3.7	9.8	3.8	7.3	4.1

Table 2. Statistics of Outlining

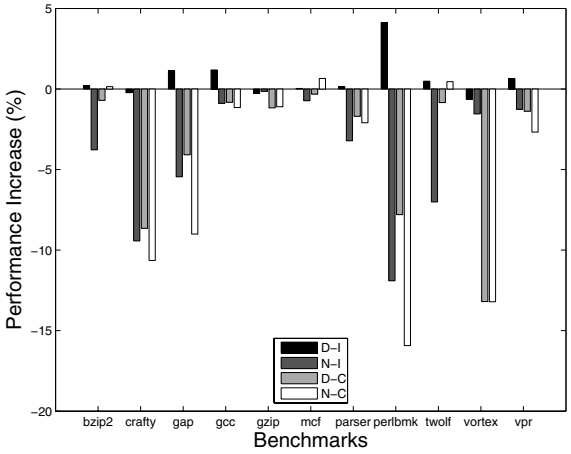

Figure 3. D-I is the only outlining strategy that does not degrade performance.

4.4 Partial Inlining

D-I is also the best strategy for partial inlining. Figure 4 shows the performance improvements due to *D-I* partial inlining.[2] `perlbmk` and `gap` improved by 5.75% and 3.90% because of partial inlining. `vpr` and `parser` have minor performance degradation (0.86% and 0.51%).

Surprisingly, very few additional call sites are inlined due to function outlining. The only benchmarks that have extra call sites inlined are: `gap` (10), `gzip` (5), `parser` (3), and `perlbmk` (5). These call sites contribute a very small percentage of the run-time function calls (less than 0.6%).

There are two major reasons that impede more aggressive inlining in the SPEC2000 integer benchmaks. More aggressive inlining is prevented by hot functions that cannot be inlined because they are too large even after outlining. To make matters worse, because few of them are leaf functions in the call graph, they often absorb other functions during function inlining and become even larger. Moreover, benchmarks that tend to bene£t from partial inlining are often large benchmarks, such as `perlbmk` and `gap`, where run-time function calls are distributed among many call sites and there is no dominating call sites. Thus, a small increase in the

[2]Partial inlining uncovered a bug elsewhere in ORC that prevents us from obtaining results for `gcc` and `vortex`.

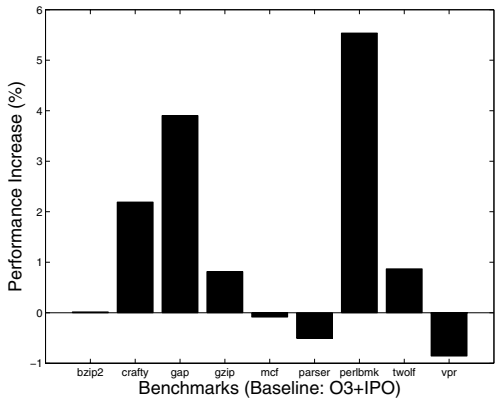

Figure 4. Outlining followed by inlining improves performance.

number of inlined sites is unlikely to yield signi£cant changes in performance.

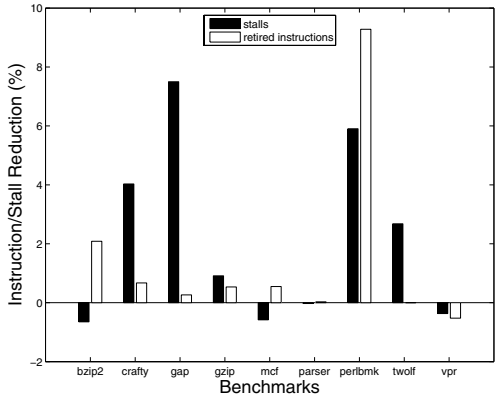

Figure 5. Effects on Stalls and Instructions

Thus, where is the the performance improvement of partial inlining coming from? First, function outlining segregates regions with heterogeneous execution frequency into separate functions and improves code placement and cache ef£ciency. Second, our outlining includes better switch optimization and explicit memory disambiguation, which might help the compiler to do a better job in other optimizations, such as code scheduling and register allocation. Figure 5 shows the changes in the number of processor stalls and retired instructions when partial inlining is enabled. There is a positive correlation between the number of retired instructions and/or processor stalls and improvements from partial inlining. For example, the processor stalls and retired instructions in `perlbmk` are reduced by about 5.9% and 9.3%, respectively. In other benchmarks, process stalls are signi£cantly reduced in `crafty`, `gap`, and `twolf`. `bzip2` also shows reduction in retired instructions (around 2%).

5 Related Work

The design and implementation of the outlining relates to previous work as follows.

Function Splitting. In their famous code positioning work[10], Pettis and Hansen separate the frequently executed code and the infrequently executed code in a program unit to optimize code layout. In their work transitions between hot and cold code is achieved by explicit jump instructions. Their control ¤ow patching method breaks the address integrity of a function and is dif£cult to implement and maintain in high-level optimizations. Castelluccia *et al.* and Mosberger *et al.* use the idea of outlining to increase the code density of network protocol code[1, 7]. They only handle `if` statements — we found that `switch` statements and early returns are also important. Muth *et al.* proposed to implement partial inlining in a link-time optimizer called ALTO [8]. Their outlining and partial inlining occur too late to bene£t from other code transformations. Way *et al.* experimented with partial inlining in early phases of a compiler backend [12, 13, 14]. Their inlining is an enabling technique to build inter-procedural regions and reduce optimization costs. They achieved less than 1% performance improvement from partial inlining.

Region Formation Algorithm. Hank's intra-procedural region formation method [4] is a generalization of the runtime feedback-based trace selection algorithm implemented in the IMPACT compiler [9]. The CFG-based region formation in the work of Suganuma *et al.* tries to identify cold regions in a hot function [11].

Semantics Preserving in Splitting. Komondoor *et al.* use function splitting to abstract repetitive code segments into a new function so that the program becomes easier to understand and maintain [5, 6]. Their splitting candidates are limited to single-entry regions while our splitting framework can handle side-entries to a region.

6 Conclusion

Our function outlining features a novel region formation approach that takes full advantage of high-level control-¤ow constructs, a set of heuristics to control the aggressiveness of outlining, and a solid patching method to maintain the correct semantics of the program. Our experimental study shows that both function outlining

and partial inlining improve the performance of some benchmarks in the SPEC2000 integer suite.

This paper focuses on outlining cold code in hot functions. Therefore it cannot handle cold functions that contain hot loops, or `heavy` functions [16]. When a heavy function is large, the ORC cannot inline the hot call sites in the loop because of the inlining budget. Function outlining is still a possible solution. However, there might be serious performance degradation if the current patching methods are used to outline hot regions out of a cold function. For instance, when variables are modi£ed in the hot region, their addresses would be passed to the new function. Thus all the related variable accesses would require one extra memory dereference. Therefore, instead of passing variable addresses, the new function should be a nested function, which can access variables in its host function directly.

References

[1] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating ef£cient protocol code from an abstract speci£cation. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 60–72, 1996.

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

[3] 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.

[4] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: An introduction and motivation. In *28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 158–168, Dec 1995.

[5] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Principles of Programming Languages (POPL)*, pages 155–169, Boston, MA, Jan 2000.

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

[7] D. Mosberger, L. Peterson, and S. O'Malley. Protocol latency: MIPS and reality. Technical report, TR-95-02, Department of Computer Science, University of Arizona, 1995.

[8] R. Muth and S. Debray. Partial inlining. Technical report, Department of Computer Science, University of Arizona, 1997.

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

[10] K. Pettis and R. C. Hansen. Pro£le guided code positioning. In *Programming Language Design and Implementation (PLDI)*, pages 16–27, 1990.

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

[12] T. Way. *Procedure restructuring for ambitious optimization*. PhD thesis, University of Delaware, May 2002.

[13] T. Way, B. Breech, and L. L. Pollock. Region formation analysis with demand-driven inlining for region-based optimization. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 24–36, 2000.

[14] 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.

[15] P. Zhao. *Code and Data Outlining*. PhD thesis, University of Alberta, Department of Computing Science, Edmonton, Alberta, Canada, April 2005.

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

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

[18] Peng Zhao and J. N. Amaral. Feedback-directed switch-case statement optimization. In *4th Workshop on Compile and Runtime Techniques for Parallel Computing*, Oslo, Norway, June 2005.