Region Formation Analysis with Demand-driven Inlining for Region-based Optimization

Tom Way, Ben Breech and Lori Pollock
Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716
{way,breech,pollock}@cis.udel.edu

Abstract

Region-based compilation repartitions a program into more desirable compilation units for optimization and scheduling, particularly beneficial for ILP architectures. With region-based compilation, the compiler can control problem size and complexity by controlling region size and contents, expose interprocedural scheduling and optimization opportunities without interprocedural analysis or large function bodies, and create compilation units for program analysis that more accurately reflect the dynamic behavior of the program. This paper presents a region formation algorithm that eliminates the high compile-time memory costs due to an aggressive inlining prepass. Individual subregions are inlined in a demand-driven way during interprocedural region formation. Our experimental results on a subset of the SPEC benchmarks demonstrate a significant reduction in compile-time memory requirements with comparable runtime performance.

1. Introduction

For abstraction, software maintenance ease, and a number of other reasons, large applications are written in a modular fashion. Compiler writers have addressed the problem of modularity by developing techniques that range from aggressive procedure inlining to procedure cloning to fast, flow and context insensitive interprocedural analysis to more sophisticated flow and context sensitive interprocedural analysis. While evidence shows that especially with many small functions, whole program analysis can significantly improve the precision of program analysis and increase opportunities for optimization, most of these methods suffer from poor scalability in either or both compilation-time memory and time requirements for large programs. All have been applied to the original program partitioned by the programmer into functions created with abstraction and good program design in mind.

A very different approach that has been developed for ILP compilers is region-based compilation [14, 15, 19], in which the compiler repartitions the program into a new, more *desirable*, set of compilation units prior to program analysis and optimization. Hank, Hwu, and Rau [15] show solid experimental evidence of the potential impact of giving the compiler control of the compilation unit size and contents. The compiler can successfully repartition the code into compilation units that will enable more aggressive optimizations to be performed. The use of profile information for region formation allows the compiler to create regions that more accurately reflect the dynamic behavior of the program. Hank, Hwu, and Rau [15] found that the size of each of the resulting compilation units is typically smaller than functions, which has the effect of reducing the impact of the quadratic algorithmic complexity of the applied optimizing transformations. Aggressive optimizations can be performed in the presence of large function bodies. In particular, by creating new regions, the compiler has more freedom to move code between procedures than approaches based on interprocedural analysis. Hank's implementation [14] included classical optimizations like global common subexpression elimination, dead code removal, and code motion, but particularly found register allocation and scheduling to be enhanced by this approach.

Region-based compilation as proposed by Hank et al. and as implemented in the IMPACT [5] compiler is accomplished by performing an aggressive inlining pass, followed by a partitioning phase that forms new regions based on a heuristic, bundles regions to look like functions, and passes these compiler-created functions to the unchanged optimization phases. While this approach can achieve some degree of scalability during program analysis and optimization by allowing the compiler to control the size of regions, the region formation phase itself remains quite unscalable.

The region formation phase works on a form of the whole program which has been aggressively inlined, in order to be able to form regions that cross function bound-

aries. We refer to regions that consist of instructions from more than one function of the original program as *interprocedural regions*, and regions with instructions all from the same original function as *local regions*. In the experiments by Hank et al. [15], approximately half of their benchmarks spent more than 50% of their time in interprocedural regions. Some programs spent more than 40% of their execution time in regions spanning 9 or more functions.

This paper addresses the problem of enabling region-based compilation for large programs, where the compile-time space and time costs must be conserved throughout all phases of compilation. We first describe the issues involved in performing the region formation process in tandem with inlining in order to avoid the separate unscalable step of aggressive inlining. We present an algorithm for region identification that incorporates demand-driven inlining as interprocedural regions are being identified and formed. This improved region formation algorithm recursively analyzes function bodies as inlinable callsites are encountered during region identification.

While the original region-based compilation framework requires that the entire aggressively inlined program be in memory at once during region formation, our approach requires at most enough memory for the functions along the longest static acyclic call chain at any given time during region formation. We have created a prototype of our demand-driven inlining and region formation algorithm by extending the region-based compilation framework in Trimaran. Our experimental comparison of the memory requirements for seven benchmarks indicates that, in the worst case, from 2% to 55%, with an average of 23%, of the aggressively inlined program size lies along the longest static acyclic call chain. Using our prototype implementation and the Trimaran simulation of an 8-processor HPL-PD EPIC machine, we verified that our method produces comparable execution times and processor utilization to those of the aggressive inlining framework, while significantly reducing compile-time memory requirements.

2. Region-based Compilation

Hank et al. [14, 15] proposed the region-based compilation framework as a solution to the problem of exposing interprocedural scheduling and optimization opportunities without the cost of very large function bodies created through inlining, or the expense and complexity of sophisticated interprocedural analysis and code motion. This region-based compilation framework is embellished in the IMPACT [5] and Trimaran compilers [21]. Limited forms of region-based compilation were used in the Multiflow [20] and Cydrome [10] compilers. While they have shown it to be especially beneficial in an ILP compiler, region-based compilation also can be useful for achieving both interprocedural scope and scalability in program analysis.

Regions are formed on the flattened form of the program after aggressive inlining. A call is *inlinable* if it is not recursive, not indirect via a function pointer, and not a precompiled library function, or if an arbitrarily set limit for code growth is not yet reached. Any remaining *uninlinable* callsites are treated as hazards, which end a region along that path. In this framework, a *region* is a subgraph of the control flow graph (CFG) of the program.

Under Hank et al.'s framework, regions are encapsulated in such a way that the optimizer can be invoked with a scope that is limited to a given region. Thus, the quality of the generated code depends upon the ability of the compiler to efficiently transform individual regions in isolation. Hank et al. use a profile-sensitive region formation process that is a generalization of the profile-based trace selection algorithm used in the IMPACT compiler [5].

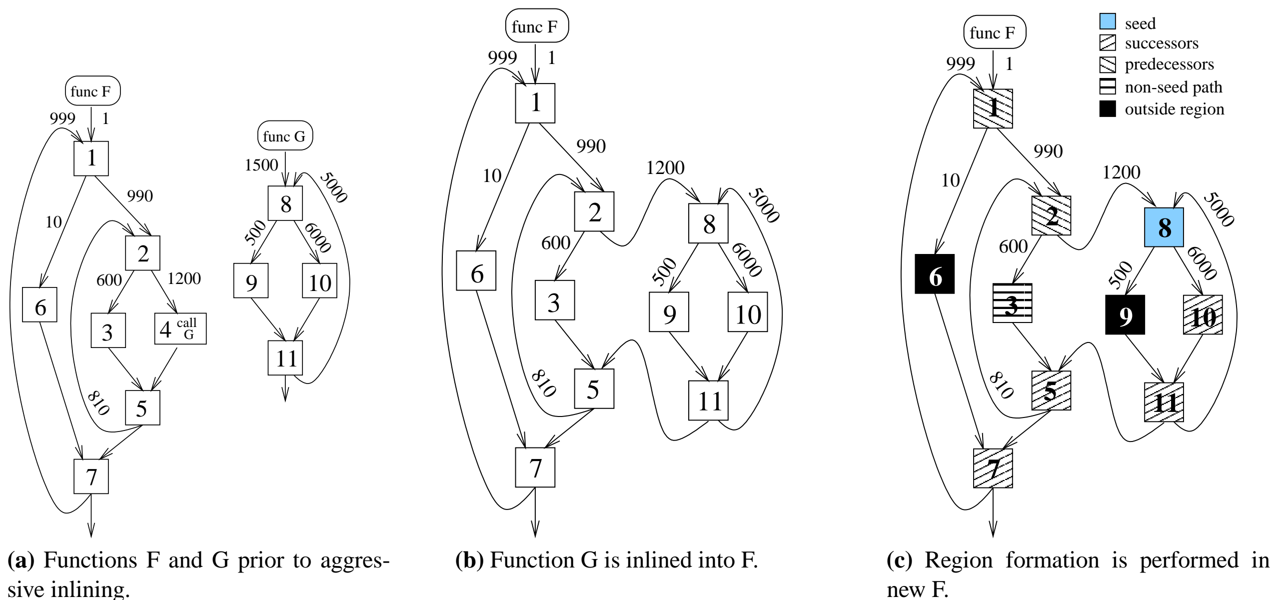
In practice, each region is encapsulated in a single-entry/single-exit CFG by adding dummy prologue and epilogue blocks and boundary condition blocks that convey variable liveness at the region exit points. Side entrances into regions can be removed by tail duplication, similar to superblock formation [17]. This encapsulation achieves the effect of making the region appear to the rest of the compiler as a function. After compilation and optimization, a region is re-integrated into the containing function by updating changes in data flow conditions (i.e. variable liveness), entry and exit points and constraints on register allocation. Code is generated from the re-integrated function.

The profile-sensitive region formation algorithm [15] is comprised of the following steps, performed between aggressive inlining and region encapsulation. These steps are performed until all blocks in the program have been included in some region. Figure 1 shows a simple example of two functions, (a) before inlining, (b) after inlining, and (c) after region formation. The results of each step of the region formation algorithm are shown by different fill patterns in the blocks in Figure 1c.

Step 1: Seed Selection - From among all basic blocks in the program not yet included in a region, select the block with the highest execution frequency. In this simplified example, this is block 8.

Step 2: Region Expansion to Successors - A path of *desirable* successors is selected, starting at the seed block. This region expansion is guided by heuristics which halt the growth under a set of conditions such as [14]:

- a procedure call is reached,
- a minimum acceptable execution frequency for a successor block is not met



(a) Functions F and G prior to aggressive inlining. **(b)** Function G is inlined into F. **(c)** Region formation is performed in new F.

Figure 1. Example of the steps in *Region_aggr* region formation

(e.g., at least 50% of the frequency of both its immediate predecessor in the region and that of the seed block, which in this simplified example is why block 6 is not selected in this step), or

- a region size threshold (e.g., 200 basic blocks) is exceeded. The successors selected in the example in this step are blocks 10, 11, 5 and 7.

Step 3: Region Expansion to Predecessors - A path of frequently executed predecessors to the seed block is chosen analogous to the selection of desirable successors. The resulting path after this step is the seed path of the region. In this case, blocks 2 and then 1 are added as predecessors.

Step 4: Region Expansion from All Blocks in the Seed Path - By selecting as above the desirable successors of *all* blocks currently in the region, the region is grown along multiple control flow paths. In this example, block 3 is added to the region. The result of this step is a path-sensitive region. Blocks not yet in a region (blocks 6 and 9) are used to form additional regions.

To summarize, three regions are formed in the example. The largest region consists of blocks 1, 2, 3, 5, 7, 8, 10, and 11. The remaining blocks 6 and 9 form single block regions. Note that original block 4 was replaced by the inlined function G.

The promise of region-based compilation is that more desirable compilation units than the original functions are provided to the compiler. Utilizing profile information and aggressive inlining, these units are grouped together to expose more optimization opportunities, increase available instruction-level parallelism, and improve generated code quality. By controlling the size of the compilation unit, region-based compilation can reduce compilation time and memory requirements for the optimizer.

Limitations of the current region-based framework include the inherent unscalability, the potential for excessive code growth and unnecessary inlining due to an aggressive approach, and the well-known training-data effect of profile-guided compilation. Placing an upper bound on the amount of *a priori* inlining performed, while reducing code growth, may produce lower optimization opportunities and quality output code.

In the remainder of this paper, we refer to the approach of aggressive inlining followed by intraprocedural profile-sensitive region formation (Hank et al.) as *Region_aggr*. Our new approach with demand-driven inlining is called *Region_demand*. In *Region_aggr*, a *hazard* is a procedure call which stops further expansion of the current region. In *Region_demand* a procedure call is not considered a *hazard*, unless the procedure is non-inlinable (e.g. recursive).

3. Issues in Forming Interprocedural Regions

Interprocedural regions are key to the power of region-based compilation to uncover optimizations missed due to

procedure boundaries [15]. When aggressive inlining is not performed prior to region formation, a number of issues need to be addressed by the interprocedural algorithm for region formation:

1. *How can the region identification process be changed to make region-sensitive inlining decisions and identify interprocedural regions as it encounters callsites, and when should the actual inlining be performed?* When a callsite is encountered as a most frequent successor or predecessor of a block in the current region, the path selection process needs to determine whether the call is inlinable (not all are), and if so, continue forming the current region inside the callee's code. In this way, inlining is driven by the *demand* placed at callsites as regions are being formed, and interprocedural regions are identified by having the region formation process cross function boundaries, creating an interprocedural algorithm. The callee's body could be inlined on demand just before beginning region formation inside the callee, but by not inlining *a priori*, we can accomplish the effect of partial inlining guided by the region formation inside the callee.

2. *How does the region identification process handle multiple callsites to the same function?* While region formation on flattened code will analyze a distinct code segment for each callsite in the original code that has been inlined, region formation without prior inlining analyzes the same body of the function called for each callsite to that function encountered during region formation. Depending on the surrounding context, a callee could be partitioned into different regions for different callsites. Thus, the interprocedural algorithm needs to maintain separate information for each inlinable callsite.

3. *How do we ensure that demand-driven inlining during region identification does not create the same or larger compile-time memory requirements as Region_aggr?* Performing demand-driven inlining can lead to similar large memory requirements as *Region_aggr* if the order that regions are formed and inlining is performed is not carefully considered. In particular, as a callsite is encountered, the region formation algorithm begins to form regions in the body of the callee. Thus, the amount of code expansion and data structures used is affected by the handling of the worklist of blocks for partitioning as the region formation crosses different functions.

4. *In the context of demand-driven inlining during region formation, how do functions which are not inlined at every callsite get processed?* While a function's code is partitioned into regions on demand from sites where it is called, there can be callsites that are not expanded, and thus, the function needs to be partitioned into (local only) regions in isolation of a calling context.

5. *What is the most appropriate way to allow the compiler to control the amount of total code growth created by Region_demand?* A limit on the memory requirements for *Region_aggr* is achieved by putting a limit on how large the program can grow in total size during the aggressive inlining pass, since the amount of memory needed during analysis is proportional to this size. So, what are reasonable ways to put limits on compile-time memory requirements used by *Region_demand*? Total code growth is not a reasonable measure for *Region_demand* since each region will be analyzed and optimized separately and demand-driven inlining ensures that the complete program will never be in memory at once.

6. *Can there be any potential added memory savings through region identification as a separate pass from code generation in this approach?* Is it possible to eliminate any duplicate regions identified in the same function with respect to different callsites, because the functions are not actually inlined first?

4. Region Formation with Demand-driven Inlining

Interprocedural regions are key to the power of region-based compilation to uncover optimizations missed due to procedure boundaries [15]. In this section, we describe the *Region_demand* approach to region formation, with the goal of uncovering interprocedural regions on demand. We assume that dynamic profiling information is available for each function in the program similar to profile-sensitive *Region_aggr*. Without loss of generality, we assume that each function is represented as a control flow graph (CFG) with a single entry block and a single exit block. Thus, in *Region_aggr*, the CFG for the aggressively inlined program would have no edges leading to/from the middle of each inlined function's control flow subgraph from/to the rest of the program's CFG representation.

4.1. A Classification of Regions of a Function

We have previously defined interprocedural and local regions in general. We now define a classification of regions with respect to individual functions and callsites where they are invoked in order to motivate and explain the algorithm. Figure 2 illustrates each of the different cases. For each function f, each callsite c with a call to f has a single *entry* region associated with f, $entry_{f,c}$ which is the region that

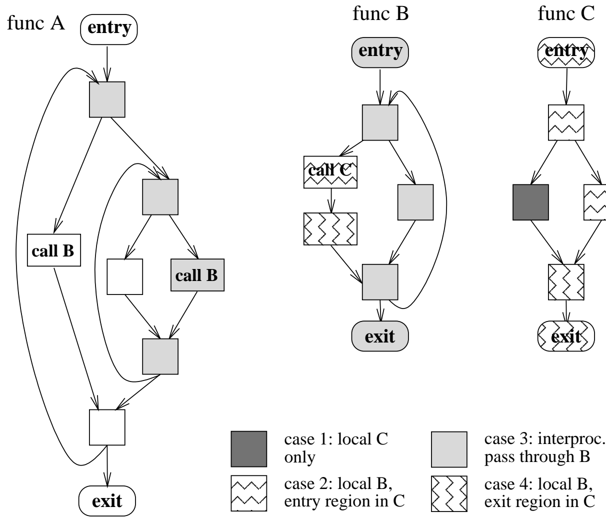


Figure 2. Illustration of region classification for individual functions

contains the entry block of f. At the one callsite in A to function B in the figure, the entry region associated with B contains not only the entry block in B but a path that passes through to the exit of B, and contains the exit of B also. At the callsite in B to function C, the entry region associated with C contains the entry block in C and only two other blocks in C.

Similarly, each callsite c to function f has a single exit region, $exit_{f,c}$. As is the case for the one callsite in A to B, $entry_{f,c}$ and $exit_{f,c}$ could in fact be the same region because the region follows a path that passes through from entry to exit; in this case, we say that this region is a *pass-through* interprocedural region of f at callsite c. All remaining regions containing blocks in f are called $local_{f,c}$, as they do not involve blocks from the caller of f. It should be noted that f may not be partitioned into the same regions for every callsite to f, since region formation within f is based on the context surrounding the callsite to f.

A region R in $local_{f,c}$ can be one of four forms:

case (1) local region: R could start in f and involve only blocks within f with no interprocedural regions involving functions called from f. R is completely local to f in this case. In Figure 2, case (1) exemplifies a local region of C.

case (2) entry region: R could start in f and involve blocks in functions called either directly or indirectly by f, and end in one of these called functions. In Figure 2, case (2) is an entry region for C because it starts in another function that eventually calls C, but the region ends within C. It is a local region to B because it starts in B.

case (3) pass-through region: R could start in f, pass through one or more functions called either directly or indirectly by f, and end back in f. In the figure, case (3) is a pass-through region of B as it contains both the entry and exit of B.

case (4) exit region: R could start in a function called by f (directly or indirectly) and end back in f. In the figure, case (4) is an exit region of C because it starts in C but ends back in the caller, B.

4.2. An Interprocedural Algorithm for Region Formation

Figure 3 presents the interprocedural algorithm for the *Region_demand* approach, *FormRegions*. This algorithm has the same basic structure as Hank's profile-sensitive region formation algorithm [14] for forming regions intraprocedurally. Hank's algorithm is extended in several important ways in order to form interprocedural regions without aggressive inlining, and also enable partial inlining. First, when a callsite is encountered as a region is being grown, *FormRegions* recursively calls itself to continue to grow the current region in the callee in the context of the caller, but without inlining at that time. Second, in order to minimize the size of the data structures maintained at any given time during region formation, all regions within a called function will be identified before *FormRegions* returns to region formation in the caller. Third, to enable formation of interprocedural regions through this recursive approach, *FormRegions* operates mainly on regions rather than individual basic blocks.

FormRegions begins with a worklist `B` of all blocks in the current function f for which it is forming regions. Successor and predecessor blocks are added to the current region only if they are desirable as defined in section 2; `Desirable(x,y)` plays this role. Non-callsite blocks are appended to the region as before. When a callsite c is reached in the analyzed code, the recursive call to *FormRegions* forms regions local to the callee, say g, and then *FormRegions* returns with the entry and exit regions of g.

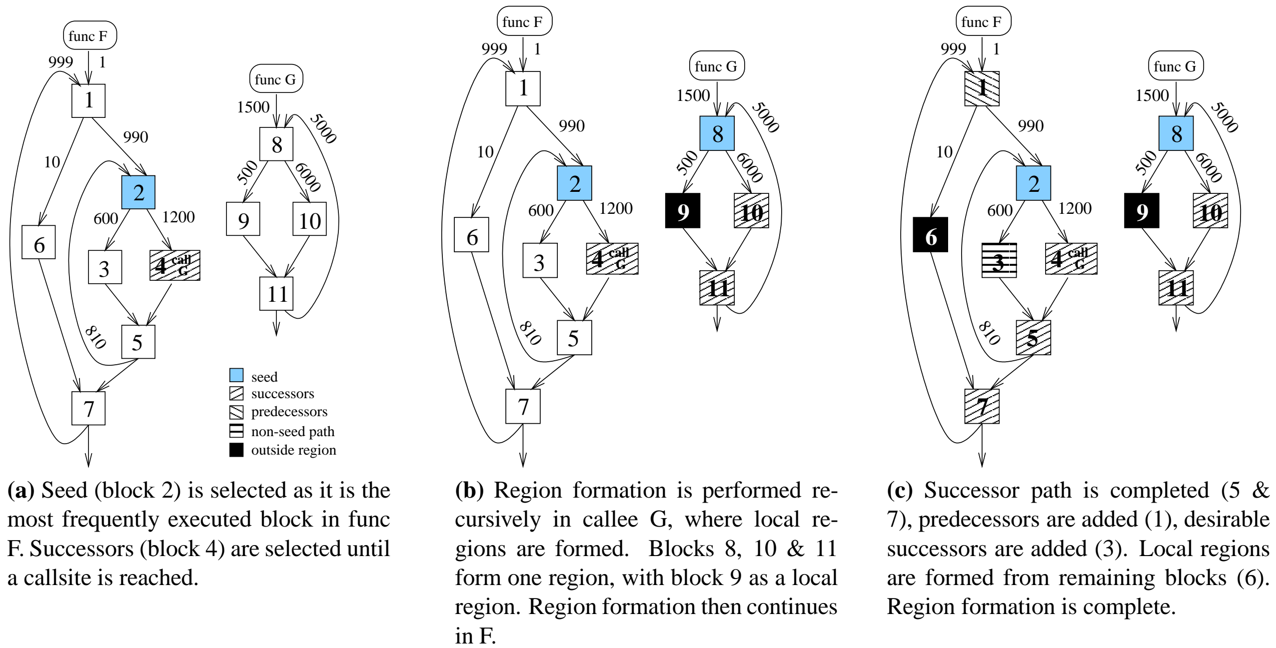
If there was not a pass-through region of g, then $entry_{g,c}$ is concatenated with the region `R` currently being formed in f when the callsite was encountered (which completes that interprocedural region), and this merged region is added to the local list `Rlist` of completed regions in f. Next, a new region `R` is begun, consisting solely of $exit_{g,c}$.

If there is a pass-through region for g, then this pass-through region is added to `R`, but `R` is not necessarily complete at this point.

Region formation continues in f by adding blocks to `R`. Once all blocks on function f's worklist `B` are exhausted,

```
function FormRegions(f, isolated, entryR, exitR) {
B = all blocks in func f
Rlist = ∅
while (blocks remain in B) {
  R = Seed(B)
  seed = last block in R

  // Add successors to the region
  x = seed
  y = most frequent successor of x
  while (y ∉ R && Desirable(x,y)) {
    if (y is func call && y is inlinable) {
      FormRegions(callee(y), 0, entryR, exitR)
      if (entryR ≠ exitR) {
        R = R ∪ entryR
        Rlist = Rlist ∪ R
        R = ∅
      }
      S = exitR
    }
    else
      S = {y}
    R = R ∪ S
    x = y
    B = B – {y}
    y = most frequent successor of x
  }

  // Add predecessors to region, analogous to adding
  // successors - code omitted for space limitations

  // Add desirable successors to seed path
  stack = R
  while (stack ≠ ∅) {
    x = Pop(stack)
    foreach successor of x, y ∈ B {
      if (Desirable(x,y))
        if (y is func call && y is inlinable) {
          FormRegions(callee(y), 0, entryR, exitR)
          if (entryR ≠ exitR) {
            R = R ∪ entryR
            Rlist = Rlist ∪ R
            R = ∅
          }
          S = exitR
        }
        else {
          S = {y}
          Push(stack,y)
          B = B – {y}
        }
        R = R ∪ S
  } }
  // Copy tail & add region to Rlist
  B = B ∪ TailDuplication(R)
  Rlist = Rlist ∪ R
 }
 // Remove entry & exit regions from list
 // generate code for regions local to f
 entryR = region in Rlist with entry of f
 exitR = region in Rlist with exit of f
 if (not isolated)
   Rlist = Rlist - (entryR ∪ exitR)
 CodeGen(Rlist)
}

function Seed(B) {
  s = block with maximum weight in B
  B = B – s
  if (s is func call) {
    FormRegions(callee(s), 0, entryR, exitR)
    if (entryR ≠ exitR) {
      R = R ∪ entryR
      Rlist = Rlist ∪ R
    }
    S = exitR
  }
  else
    S = {s}
  return S
}

function CodeGen(Rlist) {
  foreach region R ∈ Rlist
    optimize R
  generate code for Rlist
}

Main() {
  FormRegions(main, 1, entryR, exitR)
  foreach func f ≠ main
    if (not all callsites to f were inlined)
      FormRegions(f, 1, entryR, exitR)
}
```

Figure 3. Interprocedural algorithm for region formation with demand-driven inlining



(a) Seed (block 2) is selected as it is the most frequently executed block in func F. Successors (block 4) are selected until a callsite is reached.

(b) Region formation is performed recursively in callee G, where local regions are formed. Blocks 8, 10 & 11 form one region, with block 9 as a local region. Region formation then continues in F.

(c) Successor path is completed (5 & 7), predecessors are added (1), desirable successors are added (3). Local regions are formed from remaining blocks (6). Region formation is complete.

Figure 4. Example of *Region_demand*

the return parameters `entryR` and `exitR` are assigned the regions in f that contain the entry and exit blocks, respectively. The local regions with respect to f (all regions except the entry and exit regions of f) are optimized and code is generated for them, prior to returning the entry and exit regions.

The main steps of *FormRegions* are illustrated for a single callsite by the interprocedural CFGs in Figure 4. For clarity, the same fill patterns are used to differentiate the steps of the *Region_demand* algorithm in this figure as were used to describe the *Region_aggr* algorithm in Figure 1. In this example, a pass-through region of G exists, is returned to F by *FormRegions* as both `entryR` and `exitR`, and is appended to the currently forming region `R`.

Functions that are not inlined at every callsite, not inlined at all, or are potential function aliases, are identified after the region formation that began with the main program is complete. The parameter to *FormRegions* called `isolated` is used to indicate when no interprocedural regions will exist with respect to the current function, namely for forming regions in these functions and the main program.

4.3. Analysis of Space Requirements

At any given time during execution of the *FormRegions* algorithm, the amount of required memory includes the data structures and code associated with each function along the current call chain being analyzed by the recursive region formation. The function code and data structures for a function need not be in memory until a callsite to that function is encountered during the region formation. After region formation is complete for that callsite, these data structures are no longer needed. Thus, since we do not analyze bodies of functions at callsites where inlining is prohibited by recursion, the maximum memory requirement for this algorithm is proportional to the sum of the sizes of the functions along the longest acyclic path in the call graph starting at the root of the call graph.

Total code growth created by the *Region_demand* approach is based on the amount of inlining performed during region formation. Total code growth can be limited in the same way that it is for full inlining, by preventing inlining when a certain predetermined percentage of the original code size is reached. However, to limit the amount of compile-time space requirements, we propose a heuristic that limits how far the region formation will recur in the callgraph, along a specific call chain. By enforcing this limit, we force some callsites to be considered as uninlinable during region formation. Like the *Region_aggr* approach, this just has the effect of preventing interprocedural regions from being formed over those callsites. With this limit, the *Region_demand* approach can result in different callsites being inlined than in the *Region_aggr* approach.

			Trimaran Pcode expressions		
Benchmark	Lines of source code	No. of funcs	Total Prog. size	Max. func. size	Avg. func. size
008.espresso	14850	361	12503	513	34.6
023.eqntott	3628	62	2160	251	34.8
026.compress	1503	16	595	222	37.1
130.li	7597	357	6019	232	16.8
132.ijpeg	29259	473	12666	246	26.8
bmm	106	7	87	13	10.9
paraffins	388	10	209	76	20.9

Table 1. Size and function characteristics of selected benchmarks

5. Experimental Evaluation

In this section, we experimentally compare the *Region_demand* and *Region_aggr* approaches in terms of compile-time memory use, interprocedural regions, and runtime performance of the generated code.

5.1. Methodology

Our experiments have been performed on the Trimaran compiler system [21]. The Trimaran System is an integrated compilation and performance monitoring infrastructure developed through the combined efforts of the Compiler and Architecture Research Group at Hewlett Packard, the IMPACT Group at the University of Illinois and the ReaCT-ILP Laboratory at New York University. While Trimaran is especially useful for ILP research, it also provides a rich compilation framework in which optimizations and analysis modules can be easily added, deleted or bypassed. With *Region_aggr* as an existing component, Trimaran was a natural choice for our experimental work. We have used seven C benchmarks in our study, five of which are from SPEC and two are included in the Trimaran distribution. Table 1 describes the benchmarks in terms most relevant to our study. The numbers of source code lines and function definitions for each benchmark are listed along with three measurements based on number of *Pcode expressions*. *Pcode* is a high-level intermediate code used in Trimaran.

5.2. Results

Table 2 compares the compile-time memory requirements for the *Region_aggr* approach versus the *FormRegions* algorithm of *Region_demand*. Values are expressed

Benchmark	*Region aggr* Size after inline	*Region_demand* Static call chain Avg	Static call chain Max	Function size Avg	Function size Max	Memory require. Avg	Memory require. Worst
008.espresso	13871	5	11	38	919	678	1101
023.eqntott	2465	3	7	44	591	246	540
026.compress	691	2	5	43	310	284	383
130.li	6598	22	35	19	601	349	666
132.ijpeg	13856	8	14	29	501	77	237
bmm	103	2	2	14	26	24	36
paraffins	308	2	3	26	76	32	98

Table 2. Comparison of memory requirements during region formation

Benchmark	*Region_aggr* Average Reg. Size	*Region_aggr* Total Regions	*Region_demand* Average Reg. Size	*Region_demand* Total Regions
008.espresso	11.74	1787	11.87	1774
023.eqntott	6.86	436	6.55	476
026.compress	8.35	117	9.25	102
130.li	7.63	801	7.79	793
132.ijpeg	11.95	3575	11.89	1791
bmm	9.65	20	8.43	21
paraffins	6.33	55	6.08	49

Table 3. Comparison of Formed Regions: *Region_aggr* **versus** *Region_demand*

in terms of the number of *Pcode* expressions. For *Region_aggr*, the compile-time memory requirements are proportional to the size of the code after aggressive inlining is performed. For *Region_demand*, the lengths of static acyclic call chains were measured at the source code level using the *cflow* utility. To measure the compile-time memory requirements, we calculated an average and maximum of the sum of measured function sizes (in *Pcode* expressions) along all call chains. The average value provides a good estimate of typical compile-time memory usage for purposes of comparison, while the maximum value shows the worst case.

On average, *Region_demand* uses 14% of the memory required by *Region_aggr* for region formation for the benchmarks we studied, over a range of 1% to 41%. Benchmarks with larger numbers of functions and function calls, and more and longer call chains, *008.espresso*, *023.eqntott*, *130.li* and *132.ijpeg*, benefited the most from *Region_demand*. Since *Region_aggr* keeps the entire aggressively inlined program in memory throughout region formation, even the worst case memory required by *Region_demand* compares favorably in terms of compile-time space requirements. For the worst case, *Region_demand* uses from 2% to 55% as much compile-time memory as *Region_aggr*, averaging about 23%. The three smaller benchmarks, *026.compress*, *bmm* and *paraffins*, tended not to gain as much from *Region_demand*. While still making thriftier use of memory than *Region_aggr*, these three contain too few functions and function calls to gain as much as larger benchmarks do from *Region_demand*.

Hank[14] has already experimentally shown the compile time benefits of the application of classical optimization within a region-based compilation framework versus a function-based compilation framework. In particular, he has shown how optimization time improvements are due to the reduction in problem size provided by the region partition, but more importantly achieved by the ability to apply more optimization time to more important regions and spend less optimization time on unimportant regions as indicated by profiling information. Thus, in our experiments, we focused on how the region partition that we produce using *Region_demand* differs from the regions produced by Hank's region formation, *Region_aggr*, in order to evaluate whether our region formation algorithm will provide similar optimization time improvements over the function-based framework, and if not, the causes for any potential discrepancies.

For each benchmark, Table 3 compares the results of *Region_aggr* and *Region_demand* in terms of characteristics of the regions being formed. The techniques result in very similar average region sizes and total numbers of regions. Slight variations in sizes and numbers of regions are attributed to differences in the order in which callsites are inlined. The aggressive inlining of *Region_aggr* favors inlining frequently executed, smaller functions over larger functions due to the limit it places on total code growth. Since the demand-driven inliner inlines as it is creating a region and reaches a callsite, it does not need to use a heuristic based on function size because it is inlining only what it needs for the interprocedural region. The demand-driven approach to inlining in *Region_demand* and the recursive nature of the algorithm leads to the bottom-up inlining of regions. The inlining is performed as the recursive calls to *FormRegions* return. The result of this demand-driven approach is that *Region_demand* creates comparable numbers of regions to *Region_aggr* while reducing compile-time memory requirements.

Table 4 reports the results of execution of the compiled benchmarks on a simulated HPL-PD EPIC computer with 8 functional units and an instruction width of 8. Generally, results for processor resource utilization and execution time were quite similar for *Region_aggr* and *Region_demand*. There are little or no differences in perfor-

Benchmark	Operations-per-cycle		Execution Speedup
	Region_aggr	*Region_demand*	
008.espresso	2.61	2.48	0.95
023.eqntott	3.67	3.64	0.99
026.compress	2.74	2.26	0.69
130.li	2.01	1.66	0.70
132.ijpeg	1.26	1.27	0.99
bmm	3.46	3.46	1.00
paraffins	2.07	2.06	0.99

Table 4. Comparison of processor utilization and execution time speedup for *Region_aggr* and *Region_demand*.

mance for *008.espresso*, *023.eqntott*, *132.ijpeg*, *bmm* and *paraffins*. The drop in performance and instruction throughput for *026.compress* and *130.li* is due to our naive heuristics for deciding whether to inline at a given callsite, and the way our current prototype system handles demand-driven inlining of indirect recursive function calls. In particular, with our current implementation, it is possible for the code limit to be reached before inlining is performed in some of the high execution frequency regions resulting in optimization loss. We plan to enhance our prototype to perform better detection of indirect recursion, and explore a variety of different heuristics for deciding whether to inline at a given callsite during demand-driven inlining.

6. Related Work

Profile-driven function inlining and cloning have been shown to have a large to moderate positive impact on performance of C codes depending on the use of profile information and heuristics [1, 3, 6, 16]. Others have had positive, but less dramatic speedups for C without the use of profiling [8]. A study of inlining on Fortran codes indicated that there exists potential for performance degradation due to such secondary effects as increased register pressure, and that inlining should be considered primarily when it can be shown to enable high-payoff optimizations [13]. All of these experimental studies demonstrated the adverse impact of increased function size due to inlining on compile-time and space requirements. One middle-of-the-road alternative to inlining and interprocedural analysis over the original program is procedure cloning [4, 7, 9, 13, 18, 22]; however, the problem of scalability for large programs remains for cloning.

Using profile-guided region partitioning followed by region-based optimization is related to profile-guided optimization analysis and transformation. Profile-guided data flow analysis seeks to compute more precise data flow analysis for hot paths [2], while profile-guided optimization trades more aggressive optimization [11, 12] along heavily executed paths for potentially increasing the execution time along less frequently executed paths. These techniques can focus on optimizing the same paths as profile-guided region-based optimization by crossing interprocedural boundaries. However, in profile-driven region-based compilation, code segments on hot paths are explicitly separated and optimized in isolation within the new region boundaries. In this way, classical data flow analysis and optimization techniques can be applied to the compiler-generated regions without requiring specialized data flow techniques.

7. Conclusions and Future Work

By developing an interprocedural algorithm for region formation, we have eliminated the need to flatten the code via inlining in order to repartition the original program for improved analysis and optimization. Our experiments demonstrate that significant improvements in compile-time memory usage are possible using a demand-driven approach to inlining, generally without negatively impacting performance.

The contribution of this research is the development of a region-based compilation framework that controls memory usage and the size of the compilation unit, enabling more scalable optimization of larger programs. Region-based compilation increases scheduling opportunities, which is vital for improving the performance of programs running on ILP architectures. Reducing memory requirements increases the ability of a region-based compiler to extend this visibility of instructions to the scheduler while enabling better optimization on larger programs.

We are currently extending our implementation to have the capability of investigating partial inlining as a naturally occurring side-effect of region formation. We also plan to explore the impact of more sophisticated region-formation heuristics on the amount of inlining performed and therefore the size and number of interprocedural regions produced.

Acknowledgments

We would like to thank David Kaeli and the anonymous referees for their helpful comments and suggestions for improvement of this paper.

References

[1] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. *ACM SIGPLAN Con-*

ference on Programming Language Design and Implementation, pages 241–249, 1988.

[2] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–84, June 1998.

[3] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–145, 1997.

[4] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

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

[6] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–369, May 1992.

[7] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, Feb. 1993.

[8] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software Practice and Experience*, 18(8):775–790, Aug. 1988.

[9] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 93–102, June 1995.

[10] J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. *Journal of Supercomputing*, 7:181–227, 1993.

[11] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial dead code elimination using predication. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 102–115, San Francisco, California, Nov. 1997.

[12] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial redudancy elimination using speculation. In *IEEE International Conference on Computer Languages*, pages 230–239, May 1998.

[13] M. W. Hall. *Managing Interprocedural Optimization*. Ph.D. thesis, Rice University, Apr. 1991.

[14] R. E. Hank. *Region-Based Compilation*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1996.

[15] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: Introduction, motivation, and initial experience. *International Journal of Parallel Programming*, 25(2):113–146, Apr. 1997.

[16] W. W. Hwu and P. P. Chang. Inline function expansion for compiling C programs. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–257, 1989.

[17] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective structure for VLIW and superscalar compilation. *Journal of Supercomputing*, 7:229–248, July 1993.

[18] R. L. Johnston. The dynamic incremental compiler of APL/3000. In *APL'79 Conference*, pages 82–87, 1979.

[19] H. Kim, K. Gopinath, and V. Kathail. Register allocation in hyper-block for EPIC processors. In *Parallel Computing '99*, pages 36–44, 1999.

[20] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1):51–142, Jan. 1993.

[21] A. Nene, S. Talla, B. Goldberg, and R. M. Rabbah. Trimaran - an infrastructure for compiler research in instruction-level parallelism - user manual, 1998. `http://www.trimaran.org`. New York University.

[22] T. Way and L. Pollock. Using path spectra to direct function cloning. In *Workshop on Profile and Feedback-Directed Compilation*, International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 40–47, Oct. 1998.