Region-Based Compilation: An Introduction and Motivation

Richard E. Hank Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801

B. Ramakrishna Rau

Hewlett Packard Laboratories
Palo Alto, CA 94303

Abstract

As the amount of instruction-level parallelism required to fully utilize VLIW and superscalar processors increases, compilers must perform increasingly more aggressive analysis, optimization, parallelization and scheduling on the input programs. Traditionally, compilers have been built assuming functions as the unit of compilation. In this framework, function boundaries tend to hide valuable optimization opportunities from the compiler. Function inlining may be applied to assemble strongly coupled functions into the same compilation unit at the cost of very large function bodies. This paper introduces a new technique, called region-based compilation, where the compiler is allowed to repartition the program into more desirable compilation units. Region-based compilation allows the compiler to control problem size while exposing inter-procedural optimization and code motion opportunities.

Keywords: *ILP compilation, region-based compilation, compilation time complexity, function inlining, code expansion*

1 Introduction

As the amount of instruction-level parallelism (ILP) required to fully utilize high-issue rate processors increases, so does the difficulty of designing the compiler. An implementation of an ILP compiler must tradeoff the use of aggressive ILP techniques and compiler performance in terms of compile time and memory utilization. In situations where the compile time and memory usage becomes too large, the aggressiveness of the applied transformations must be scaled back to avoid excessive compilation cost. Also, the implementation of ILP compilation techniques may require the use of certain simplifying constraints and heuristics to make the technique viable in a production environment. The implementation of trace scheduling within the Multiflow compiler provides an example [1]. As a result, a production quality implementation may not reflect the true potential of a technique.

In order to satisfy the need for more ILP, compilers increasingly resort to inlining to support inter-procedural optimization and scheduling [2][3][4]. However, inlining often results in excessively large function bodies that make aggressive global analysis and transformation techniques, such as global dataflow analysis and register allocation, ineffective and intractable. The root of this problem is the function-oriented framework assumed in conventional compilers. Traditionally, the compilation process has been built using the function as a compilation unit, because the function body provides a convenient way to partition the process of compiling a program. Unfortunately, the size and contents of a function may not be suitable for aggressive optimization. For this reason, the function-based partitioning of the program may not provide the most desirable compilation units to the compiler.

The purpose of this paper is to motivate the selection of the fundamental compilation unit by the compiler rather than the software designer. Essentially, the compiler is allowed to repartition the program into a new set of compilation units, called *regions*. These regions will replace functions as the fundamental unit to which all transformations will be applied. This approach was used in a more restricted context within the Multiflow compiler where scheduling and register allocation are applied to traces [1]. Under the region-based framework, each region may be compiled completely before compilation proceeds to the next region. In this sense, the fundamental mode of compilation has not been altered and all previously proposed function-oriented compiler transformations may be applied.

Such an approach to compilation has several potential advantages. First, the compiler is in complete control over the size and contents of the compilation unit. This is not true with functions. Second, the size of the compilation unit is typically smaller than functions reducing the importance of the algorithmic complexity of the applied ILP transformations. Finally, the use of profile information to select regions allows the compiler to select compilation units that more accurately reflect the dynamic behavior of the program and may allow the compiler to produce more optimal code.

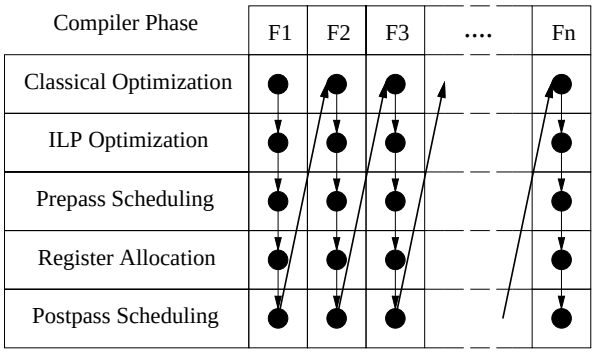

Figure 1: Block diagram of function-based compilation.

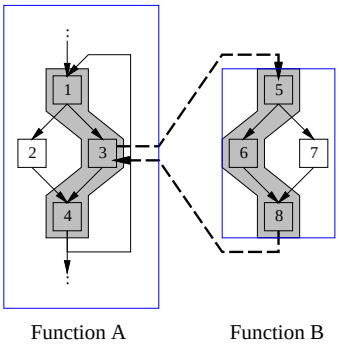

Figure 2: Example of an undesirable function-based partition.

The remainder of this paper is divided as follows. Section 2 discusses the characteristics of function-based compilation units. Section 3 discusses the benefits and drawbacks of applying aggressive inlining within an ILP compiler. Sections 4 and 5 discuss the implications of the region-based approach to ILP compilation and the research issues involved. Finally, Section 7 contains a summary.

2 Desirable Compilation Units

Traditionally, the compilation process has been built assuming functions as the unit of compilation. The function body provides a convenient way to break up the task of compiling a program, since each function is a self contained entity. Typically, the compiler processes each function of the program in turn, applying a phase ordered suite of transformations. Figure 1 illustrates the process of compiling a program of n functions. The partitioning of the program into functions is done by the software engineer usually to satisfy standard software engineering practices: modularity, reuse, and maintainability. For this reason, the function-based partition may not provide the most desirable compilation units to the compiler.

Consider the two functions shown in Figure 2. Blocks 1-4 of function A form a very frequently iterated loop. Within block 3 there is a subroutine call to function B. As a result, function B, which consists of blocks 5-8, is very frequently executed. The shaded portions represent the dynamic behavior of these two functions and indicate that blocks 2 and 7 are infrequently executed. While compiling function A, the scope of the compiler is limited to the contents of function A. The contents of function B are hidden. Likewise, the fact that function B is part of a cycle, is hidden from the compiler while compiling function B. In this case, the function-based compilation units are hiding potential optimization opportunities making this partitioning undesirable.

The type of compilation units desirable to an aggressive ILP compiler depends upon the techniques and transformations employed by the compiler. Conventional wisdom expects programs to spend most of their time in loops, since any program that executes for an appreciable amount of time must contain at least one cycle. This belief is supported by the large amount of active research being done to extract ILP from cyclic code [5][6][7][8][9][10][11][12][13][14][15]. Exposing more cycles to an aggressive ILP compiler increases the likelihood that these techniques may be applied to generate more efficient code.

By examining the dynamic distribution of cyclic and acyclic code within functions we can gain some insight as to the quality of the function-based partition of the program. Figure 3 contains the dynamic distribution of cyclic and acyclic code within the function bodies of several non-numeric programs. The large percentage of time spend in cyclic code within the programs validates the importance of cycles. The programs **lex**, **yacc**, and **023.eqntott**, which spend at least 95% of their execution time in cyclic code, fall in line with the philosophy that programs spend most of their time in loops. However, there are several programs that appear to spend an unexpectedly large percentage of their execution time in acyclic code. The programs **tbl**, **022.li**, and **perl** spend more than 50% of their time in intra-procedural acyclic code.

A large percentage of time spent in acyclic code implies that these programs contain cycles spanning function boundaries that are outside the scope of the compiler. These inter-procedural cycles are caused by the presence of subroutine calls within loop bodies, such as in Figure 2, and procedure call cycles, i.e., recursion. The function-based partition of the program hides the existence of these cycles, as well as other valuable optimization opportunities. By making these inter-procedurally coupled portions of the program simultaneously visible to the compiler, the potential for an aggressive compiler to expose more ILP is increased.

Function inlining is the only well known technique that will allow inter-procedurally coupled portions of the program to be assembled into the same compilation unit. However, within a function-based framework, any transformation applied during the compilation process must ensure that code expansion within the function body will not adversely affect the rest of the compilation process. Application of function inlining to the example in Figure 2

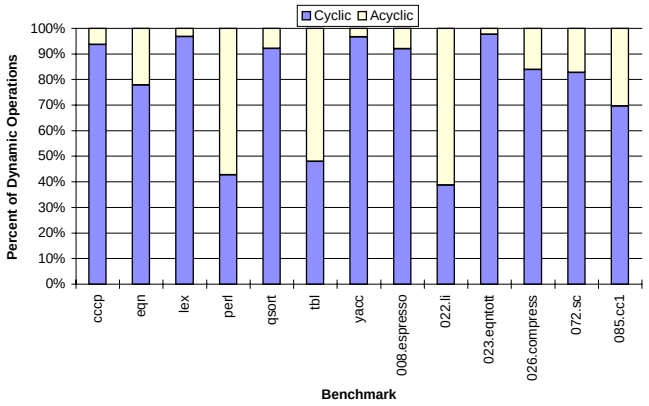

Figure 3: Intra-procedural distribution of dynamic acyclic and cyclic code.

successfully exposes the entire cycle to the compiler by placing the contents of function B into function A. However, function A now contains two basic blocks, 2 and 7, that are dynamically unimportant to the compilation of the exposed cycle. Thus, the effectiveness of the rest of the compilation process could be unnecessarily affected by the presence of these basic blocks.

Consider a situation were the compiler is allowed to repartition the program into a new set of compilation units, called *regions*. Where a *region* is defined as an arbitrary collection of basic blocks selected to be compiled as a unit. Under this framework, the compiler may select compilation units that are more representative of program behavior. Focusing the attention of the compiler on these regions as self-contained entities has several benefits. The compiler may more accurately determine the class of transformations applicable to particular region of the program. Also, individual regions are isolated from the code expansion effects in other regions and the surrounding function body. This allows the aggressive application of inline expansion under a region-based compilation framework to aid the formation of more desirable compilation units.

After inlining function B into A, the compiler may repartition the program and select the preferred region that consists of basic blocks 1, 3, 4, 5, 6, and 8. Blocks 2 and 7 will be placed in other regions and will no longer affect the compilation of this cycle. Compilation of this region as a self-contained entity has several implications to the compilation process that will be discussed in more detail in Sections 4 and 5.

3 Function Inlining

Traditionally, the goal of function inlining has been to eliminate the overhead of frequent subroutine calls [3][4]. Within the context of an ILP compiler, the goal of inlining is to increase the visibility of the compiler by exposing code that is hidden by subroutine calls. This benefits the compiler in several ways. Additional opportunities for the application of classical optimizations, such as, common

```
getline(s)
        register char *s;
{       register c;
        while((*s++=c=gtc())!='\n' && c!=EOF && c!=lefteq)
                if (s >= in+MAXLINE) {
                        error( !FATAL, "input line too long: %.20s\n", in);
                                in[MAXLINE] = '\0';
                                break;
        }
        if (c==lefteq)
                s--;
        *s++ = '\0';
        return(c);
}

gtc() {
loop:
        if (ip > ibuf)
                return(*--ip);
        lastchar = getc(curfile);
        if (lastchar=='\n')
                linect++;
        if (lastchar != EOF)
                return(lastchar);
        if (++ifile > svargc) {
                return(EOF);
        }
        fclose(curfile);
        linect = 1;
        if (openinfile() == 0)
                goto loop;
        return(EOF);
}
```

Figure 4: Source code for functions *getline* and *gtc*.

subexpression elimination, constant propagation, and loop invariant code motion are exposed [3]. Assembling larger compilation units may allow privatization of the code, improve variable aliasing information [2] and may subsume some inter-procedural analysis [4].

In addition, inlining frequent function calls tends to increase the amount of cyclic code visible to the compiler. This may increase the opportunities for application of techniques designed to extract ILP from cyclic code. A detailed example of the ILP benefits to be gained from inlining is provided in the next section. The negative effects of inlining within a function-based compilation framework are discussed in Section 3.2.

3.1 Benefits of Inlining - An Example

The function-based partitioning of the non-numeric program **eqn** provides an example of the potential ILP benefits of inlining. Intra-procedurally, **eqn** appears to have a large percentage of frequently executed code that does not occur within the body of a loop. Figure 3 shows that **eqn** appears to spend 22% of its execution time within acyclic code. This is the result of several inter-procedural cycles that are caused by the presence of subroutine calls within the bodies of frequently iterated loops.

One such inter-procedural cycle spans the two functions *getline* and *gtc*. The source code for these two functions is shown in Figure 4. The function *getline* contains a very frequently iterated loop which calls the function *gtc* once ev-

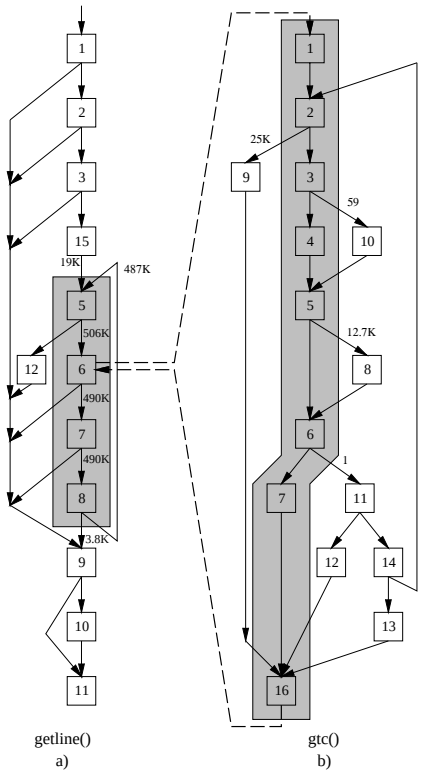

Figure 5: Control flow graphs for the functions a) *getline* and b) *gtc*.

ery iteration. Inlining function *gtc* into the call site within *getline* provides significant benefit beyond simply eliminating the overhead of the subroutine call. This is illustrated through the use of superblock optimization and scheduling techniques [13] as follows.

The control flow graph (CFG) for the function *getline* is shown in Figure 5a. The loop is composed of basic blocks 5, 6, 7, and 8. Basic block 6 contains a subroutine call to the function *gtc*. The CFG for *gtc* is shown in Figure 5b. The dotted lines indicate the implicit flow of control between these two functions.

Consider the application of superblock formation and optimization to the function *getline* as it appears in Figure 5a. Several superblocks will be formed. However, we are concerned primarily with the superblock loop generated from basic blocks 5, 6, 7, and 8. This is indicated by the shaded area in Figure 5a. These basic blocks correspond to the shaded portion of the *getline* source code in Figure 4. The contents of the resulting superblock after optimization is shown in Figure 6a. Scheduling this superblock loop for an 8-issue, fully uniform machine, yields the issue times shown to the right of Figure 6a. One iteration of this superblock loop requires four cycles. Applying superblock formation to the function *gtc* yields the superblock indicated by the shaded area in Figure 5b. The corresponding source code lines are shaded in Figure 4.

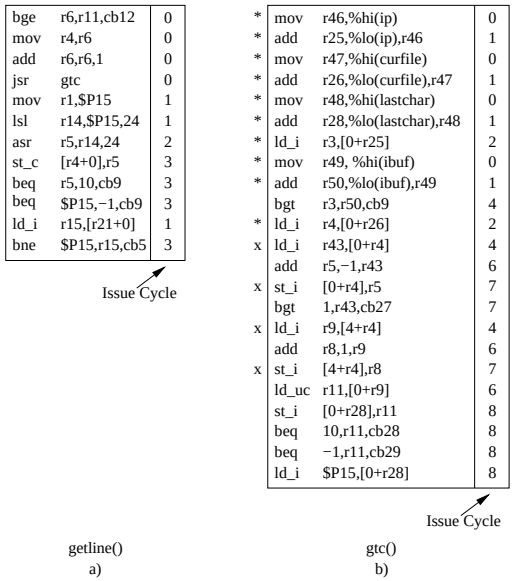

Figure 6: Superblock contents prior to inlining for a) *getline* and b) *gtc*.

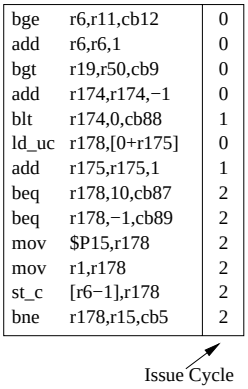

Figure 7: Superblock loop after inlining *gtc* into *getline*.

Again, the contents of the resulting superblock after optimization and scheduling for the same 8-issue, fully uniform machine is shown in Figure 6b. This superblock requires nine cycles to completely execute. Thus one loop iteration requires 13 cycles *not* including subroutine call overhead.

Consider the application of superblock formation and optimization to the function *getline* after the inline expansion of the function *gtc* into the call site in basic block 6 of *getline*. The loop in the function *getline* now contains all blocks from the function *gtc*. In this case, inlining has certainly increased the amount of code visible to the compiler, but it has also increased the amount of cyclic code visible to the compiler. The blocks inlined from *gtc* are now subject to loop-based optimization techniques, since their presence within the cycle is known to the compiler. Superblock formation yields a superblock that contains the blocks in both shaded areas of Figure 5.

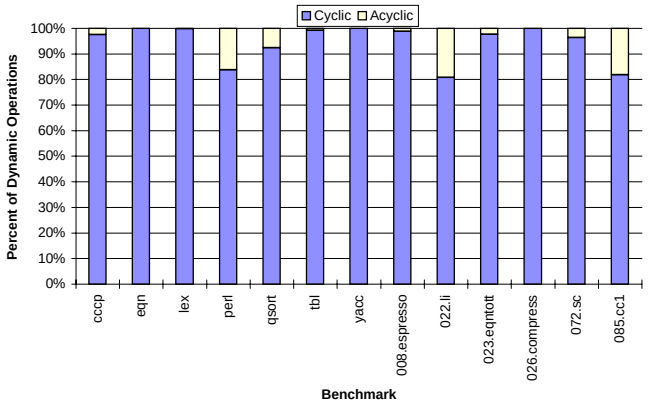

Figure 8: Distribution of dynamic acyclic and cyclic code after aggressive inlining.

Program	No. Oper	Inline No. Oper	Growth
cccp	5280	16259	3.1
eqn	3868	14077	3.6
lex	5501	11529	2.1
qsort	146	146	1.0
tbl	6375	28311	4.4
yacc	5154	13078	2.5
perl	44558	110298	2.5
008.espresso	27009	114818	4.3
022.li	8348	145099	17.4
023.eqntott	3695	13078	3.5
026.compress	1305	1428	1.1
072.sc	11433	20008	1.8
085.cc1	107414	554059	5.2
Average			4.0

Table 1: Static code growth due to inlining.

This superblock loop presents several optimization opportunities that were not available prior to inline expansion. Applying superblock optimizations to this loop results in the code shown in Figure 7. The loop contains 13 operations, one more than the original superblock loop despite the large amount of code added during inlining. The application of loop-based optimizations eliminates most of the operations from the superblock loop body. Application of loop invariant code elimination [16] allows the operations indicated by an (*) in Figure 6b to be removed. Also, the application of operation migration [13] allows the operations indicated by an (x) in Figure 6b to be hoisted outside the superblock loop body. These code optimizations would not be accomplished without compiling functions *getline* and *gtc* together. Scheduling this superblock for the same 8-issue, fully uniform machine actually produces schedule with length three cycles, one cycle shorter than the original superblock loop in *getline*.

Inlining the function *gtc* into *getline* results in a cycle that is four times shorter than in the non-inlined case. Detailed simulation of an 8-issue processor executing **eqn** and gathering statistics for this cycle shows that before inlining 7.24M cycles are spent in these two functions. After inlining, the loop requires only 1.9M cycles. The speedup is 3.8 which corresponds closely to the estimate of 4. Inline expansion has provided the compiler with many more ILP optimization opportunities than prior to inlining, yielding significant performance improvement.

3.2 Aggressive Inlining

The previous example illustrates the benefits of exposing hidden cycles to the compiler through the use of inlining. In order to form better compilation units, it is desirable to expose all of the frequently executed cycles that are hidden by the function-based partition of the program. This can be achieved by aggressively applying profile-based function inlining. Figure 8 shows how aggressive inlining affects the distribution of cyclic and acyclic code within the benchmarks shown in Figure 3.

For the most part, the desired result is achieved. The dynamic percentage of acyclic code for all but three of the programs has been reduced to less than 10%. This implies that most of the frequently executed cycles have been assembled and made visible to the compiler. The programs **022.li**, **085.cc1**, and **perl** still contain about 20% dynamic acyclic code. The principle reason for this is recursion. Although the inliner was allowed to inline self-recursive functions, this will not necessarily increase the amount of visible cyclic code. Inlining a self-recursive function into itself does not expose a cycle within the function body since the cycle is still hidden by the subroutine call. Inlining of recursive cycles does however serve to increase the scope of the compiler in the same way unrolling is applied to increase the scope of the compiler for iterative cycles.

Despite the obvious benefits of increasing the compilation scope in this way, inlining has several negative effects on the compiler's performance within the current function-based compilation framework. Inline expansion may increase register pressure to the point where the resulting spill code negates any benefit to be gained from the inlining [4]. More important, aggressive inline expansion can lead to excessive code expansion. The increase in function size will have adverse effects on compile time due to the algorithmic complexity of dataflow analysis, optimization, scheduling and register allocation.

The code expansion resulting from aggressively inlining benchmark programs is shown in Table 1. The code expansion ranges from 1.0 to 17.4 times the original code size, with an average increase of 4 times. The data presented in Figure 9 provides better insight into the effect inlining may have on compilation. Figure 9 contains histograms of the static function size weighted by the number of dynamic operations in each function. The function bodies within these programs tend to be rather small. Prior to inlining all 13 programs spent 80% of their execution time in functions with fewer than 250 operations[1]. Whereas after inlining there is a noticeable shift to the right. After inlining, over

[1]These and all subsequent operation counts are before the application of any aggressive optimization other than inlining. Since we are interested in assembling program units for compilation, this measure of size has merit.

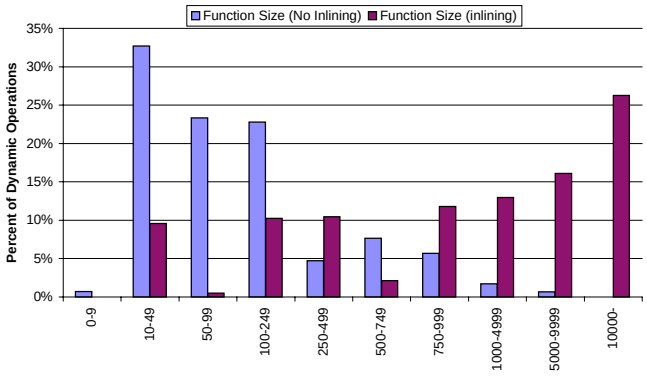

Figure 9: Histogram of function size before and after inlining.

50% of the program's execution time is spent in functions with more than 1000 operations. Inlining has succeeded in assembling the inter-procedurally coupled portions of the programs together. However, the areas of the program that should be subject to the most aggressive ILP techniques are now located within the largest function bodies. The tractability of applying aggressive ILP compilation techniques under these conditions is questionable. In this situation, the attention of the compiler must be focused such that the important regions of the program can be aggressively optimized without the compile time being affected by the size of the surrounding function body.

4 Region-Based Compilation Units

The compilation difficulties that arise as a result of inlining and other ILP techniques are a product of the compilation framework. Function-based compilation units are not representative of the behavior of the program and applied transformations are restricted to prevent excessive code expansion that may adversely affect compilation time. By allowing the compiler to repartition the program into regions, the compiler is provided a more desirable compilation unit that has the potential to result in better quality code. In addition, since each region is compiled as a self-contained entity, the compilation process for a region is not affected by code expansion within the surrounding function body.

This section will present a profile-based algorithm for region selection and discuss the application of this algorithm to aggressively inlined functions.

4.1 Region Formation

The goal of the region formation process is to provide the best possible compilation unit for aggressive optimization. The properties of the selected compilation unit should be such that aggressive optimization is both feasible and beneficial. In order to ensure that compilation time and memory usage fall within certain constraints, region formation should consider factors such as the number of operations, the number of virtual registers, the number of memory dependences, etc. In order to make aggressive optimization beneficial, region formation should consider factors that affect the quality of the generated code. These factors include the dynamic program behavior, the presence of optimization hazards [17], the control flow structure, etc. A possible region formation algorithm is presented next that takes into account dynamic program behavior.

This region formation algorithm is a generalization of the profile-based trace selection algorithm used in the IMPACT compiler [18]. The principle difference being that the algorithm is permitted to expand the region along more than one path of control. The use of profile information for region formation provides the compiler with an accurate indication of the interaction between basic blocks in the program. This results in compilation units that are more representative of the dynamic behavior of the program than the original functions.

The region formation process begins by selecting a seed block, s, which is the most frequently executed block not yet in a region. From this seed block, the scope of the region is expanded by selecting a path of desirable successors from s. For this discussion, desirability is based solely upon execution frequency. There are no other restrictions placed on the region formation process, however factors other than execution frequency should be considered in the future. Thus, a desirable successor of a block x is a block y that is likely to be executed once the flow of control enters block x. The control flow transition from block x to block y is considered likely if the weight of the control flow arc from x to y, $W(x \rightarrow y)$, is at least $(T \times 100)\%$ of the weight of block x, $W(x)$. Furthermore, to prevent the inclusion of irrelevant blocks to the region, the execution frequency of y must be at least $(T_s \times 100)\%$ of the execution frequency of s, where the values T and T_s are threshold values defined by the compiler. Therefore, y is a desirable successor of x if it satisfies the following equation.

$$Succ(x,y) = \left(\frac{\mathcal{W}(x \rightarrow y)}{\mathcal{W}(x)} \geq T\right) \&\& \left(\frac{\mathcal{W}(y)}{\mathcal{W}(s)} \geq T_s\right) \qquad (1)$$

Once the most desirable successor of s has been added to the region, the most desirable successor of that block is selected. This process continues until the successor path can no longer be extended. A path of most desirable predecessors from s is added next. The conditions under which block y is a desirable predecessor of a block x are analogous to the successor case. The resulting path forms the seed path of the region. The region is further extended by selecting all possible desirable successors from every block in the region. Each selected block is then added to the region and the process continues until no more desirable successors are found. This has the effect of adding all of the desirable paths that extend out from the seed path. The algorithm is summarized in Figure 10.

Consider the application of this algorithm to the inlining example from Section 3.1. Figure 5 represents the CFG after inlining *gtc* into *getline* if the dashed arcs are replaced by solid arcs. For this example, the assumed value of the

```
/* most frequent block not in a region */
seed = Seed(B)

/* select path of desirable successors */
x = seed
y = most frequent successor of x
while ( y ∋ R && Succ(x, y) ) {
      R = R ∪ {y}
      x = y
      y = most frequent successor of x
}

/* select path of desirable predecessors */
x = seed
y = most frequent predecessor of x
while ( y ∋ R && Pred(x, y) ) {
      R = R ∪ {y}
      x = y
      y = most frequent predecessor of x
}

/* select desirable successors of all blocks */
stack = R
while ( stack ≠ ∅ ) {
      x = Pop(stack)
      for each successor of x, y ∋ R {
            if ( Succ(x, y) ) {
                  R = R ∪ {y}
                  Push(stack,y)
}}}
```

Figure 10: An algorithm for region formation.

thresholds T and T_s is 0.10. Region formation begins by selecting the most frequent block not yet in a region. The loop header, block 5 of *getline*, will be selected as the seed block. The region selection process will then select block 6 in *getline*, blocks 1-7 and 16 in *gtc*, and finally blocks 7 and 8 in *getline*, in that order, as desirable successors because their execution frequency is very close to that of block 5. The third step of region formation is to select desirable predecessors of the seed block. In this instance, the execution frequency of block 15 (19K) in *getline* is much less than that of block 5 (506K), since block 5 is the header of a frequently iterated loop. Thus no predecessors are selected. If, in fact, the preheader and header blocks of a loop have similar execution frequencies, the region selection process would grow the region outside the loop, since the loop tends to infrequently iterate. The last step of region selection is to select desirable successors of all blocks currently in the region. The contents of the region before this step are indicated by the shaded area of Figure 5. In this example, the profile weights indicate that the dynamic behavior of these functions is extremely biased towards the currently selected path. Thus, there are no successors of the blocks currently in the region that satisfy the desirable successor condition, Equation 1. The region is now complete and is representative of the dynamic behavior of this area of the program.

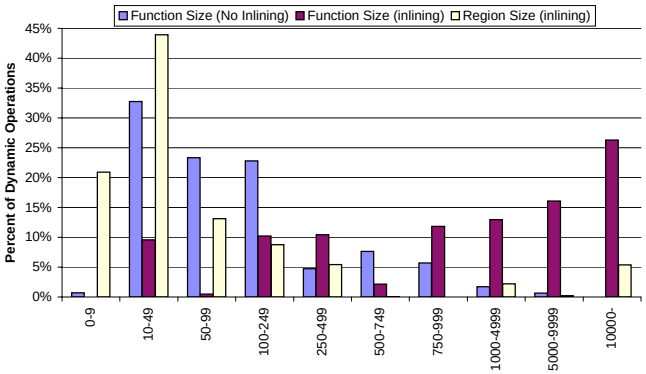

Figure 11: Histogram comparing function size before and after inlining with regions selected on inlined code.

4.2 Problem Size Control

One problem with function-based compilation units is that function size is potentially unbounded, especially if aggressive inlining is employed. The compiler engineer must deal with the time and memory complexity of algorithms in the presence of unbounded problem size. Allowing the compiler to select region-based compilation units, places the compiler in complete control of the problem size. Since the problem space of the compiler is now localized to a region, the size of the function body or the code expansion in other regions has no effect on the compilation of the current region. Reducing the problem space has the advantage of reducing the importance of the time complexity and memory complexity of the optimization, scheduling and register allocation algorithms used by the compiler. This simplifies the task of a compiler engineer developing a production quality ILP compiler.

The region formation algorithm presented in Section 4.1 used profile weight as the sole criterion for region formation. No upper bound on region size was imposed, in order to determine how well the profile information naturally controlled the size of the selected regions. Figure 11 adds a histogram of the selected regions to the function size histograms shown in Figure 9. For all 13 programs, 85% of the execution time was contained in regions with fewer than 250 operations. The large percentage (20%) of regions that contain less than 10 operations results from the fact that many of these integer programs are dominated by small loop bodies. Small cyclic regions are not a significant problem for optimization and scheduling, because the amount of ILP within a loop is essentially limited only by the trip count. There were also some regions formed with more than 10000 operations. These large regions were formed within **085.cc1** after aggressive inlining and they can be easily prevented by placing an upper bound on the allowable region size. Overall, the presented region selection algorithm is successful in controlling the problem size even in the presence of aggressive inlining.

Presenting the region size characteristics of all 13 programs within the same histogram hides some information.

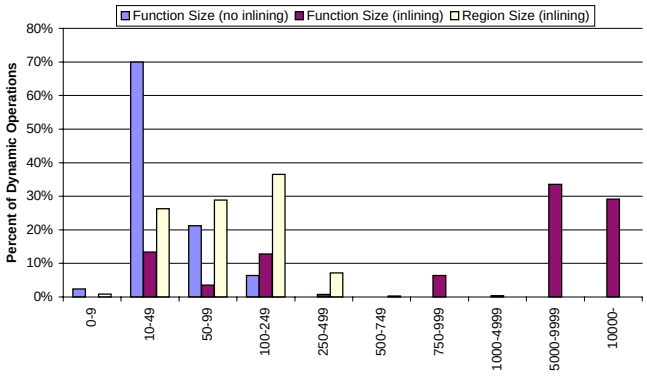

Figure 12: Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for 022.li.

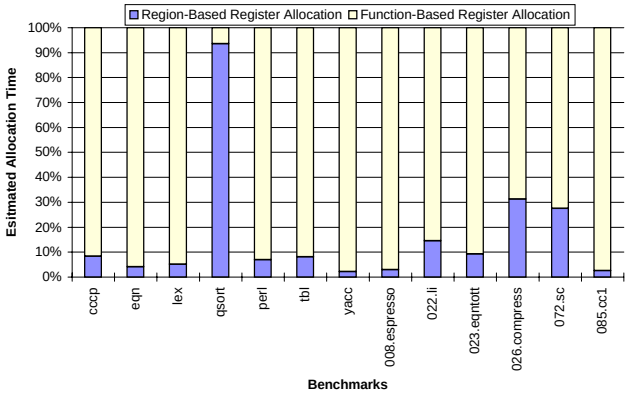

Figure 13: Estimated register allocation times for function-based and region-based compilation units.

The comparison of region size and function size for the program **022.li** provides some interesting insights[2], see Figure 12. Notice that all functions in **022.li** contain fewer than 250 operations prior to inlining; actually there are no functions containing more than 175 operations. After inlining, however, 50% of the execution time of **022.li** is shifted into functions containing more than 5000 operations. Inlining has increased the scope of the compiler to an extreme in this case. After performing region selection, the selected compilation units have two beneficial characteristics. First, the problem size is much smaller than the function bodies resulting from inlining. Second, the histogram indicates that the selected regions are larger than the original function bodies, which were extremely small. Thus, region formation properly enlarged the compilation scope when needed. The result is a shifting of the compilation scope into more desirable compilation units rather than the drastic increase seen at the function level.

In order to better illustrate the benefits of controlling problem size, consider the effect of problem size on a global graph-coloring register allocator [19][20]. The register allocation process generally consists of three steps: interference graph construction, register assignment or graph coloring, and spill code insertion. The computational complexity of the interference graph construction and graph coloring steps is $O(n^2)$, where n is the number of virtual registers in the compilation unit. The amount of time and memory required for global register allocation is heavily dependent upon the size of the compilation unit in terms of virtual registers. The region-based partition may reduce the amount of time and memory required by reducing the number of virtual registers visible to the register allocator. A technique proposed by Gupta *et al.* [21], with a similar goal, reduces the memory requirements of register allocation by using clique separators to avoid constructing the entire interference graph for a function.

By characterizing the time complexity of the global register allocator within the IMPACT compiler, a function was obtained that estimates register allocation time as a function of virtual register count. Figure 13 contains the register allocation time required for all regions selected on the aggressively inlined programs relative to the amount of time required to register allocate the large function bodies. Note that the time to register allocate the region-based compilation units is, for most of the programs, less than 10% of the time required to allocate the large functions[3]. The register allocation times for **qsort** are almost identical due to the small size of the program. The region-based partition will have similar effects for other compilation phases that use algorithms of nonlinear complexity.

4.3 Regions Spanning Multiple Functions

The intra-procedural distribution of cyclic vs. acyclic code in Figure 3 implies a significant amount of inter-procedural coupling between program functions. Figure 8 shows that through aggressive inlining, most of the inter-procedural transitions have been transformed into intra-procedural transitions. By examining the selected regions, we can determine the degree of inter-procedural coupling, as indicated by the dynamic profile information. Figure 14 shows the fraction of regions that span each given number of function bodies. For each region selected, the number of original functions represented by the blocks within the region are counted. These regions are then weighted by dynamic execution frequency to produce the distribution.

For example, the region selected from **eqn** within Section 4.1 contains blocks from two functions, namely *getline* and *gtc*. This region accounts for 30% of the execution time within that program. Thus, Figure 14 shows that for **eqn** more than 30% of the execution time spans two functions. As selected, this region contains 32 of the 9600 operations in the function body that contains it.

The distributions in Figure 14 indicate the depth of the inter-procedural coupling within these programs. Several of the programs have a great deal of inter-procedural cou-

[2] Unfortunately space does not permit discussing such a comparison for all 13 programs.

[3] The quality of register allocation is practically identical for these benchmarks with the function-based approach and the region-based approach.

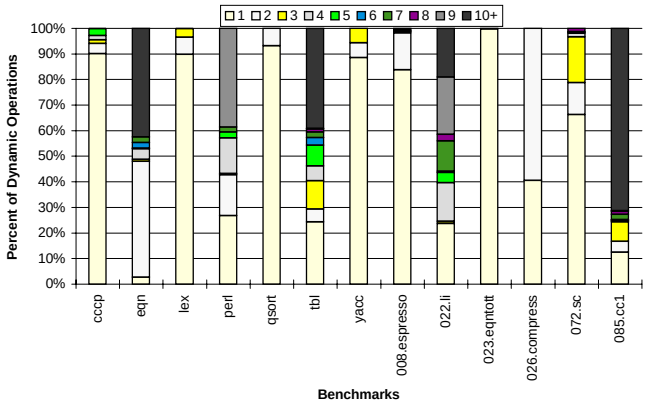

Figure 14: Dynamic distribution of the number of functions spanned by selected regions.

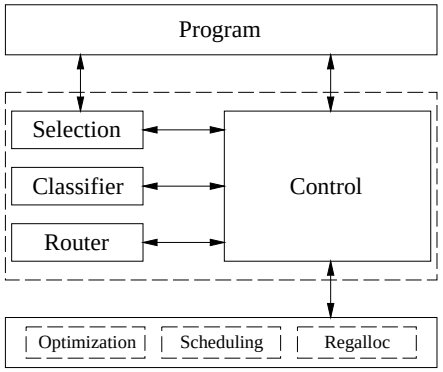

Figure 15: Block diagram of a region-based compiler.

pling. The programs **eqn** and **022.li** spend more than 40% of their execution time in regions spanning nine or more functions. For **022.li** most of these regions are small, containing fewer than 250 operations. The program **085.cc1** spends more than 70% of its execution time in regions spanning 10 or more function. Within **perl**, through inlining and region selection, a cyclic region is formed that spans nine functions and represents 25% of the execution time of the program. The importance of this region shows that it is desirable to assemble inter-procedurally coupled blocks in the program into the same compilation unit. This region contains only 230 operations, whereas the function it is contained in contains approximately 15,000 operations after inlining. Under a region-based framework, the compiler is able to isolate and perform aggressive optimization on this portion of the program without being affected by the large number of operations that actually reside within the function body. Aggressive ILP compilation of such large functions would be extremely expensive in terms of compilation time and memory.

5 Research Opportunities

A region-based compiler has several characteristics that distinguish it from a more traditional function-based compiler. Figure 15 contains a block diagram of a region-based compiler. The upper block represents the program being compiled, the bottom block represents the suite of available transformations, and the center block the core of the compiler. A region-based compiler has three more capabilities than a function-based compiler: region selection, classification, routing. There are a number of research issues raised by the application of each of these tasks that must be addressed in the design of a region-based ILP compiler.

Region Selection. The region-based compiler begins by repartitioning the program into regions. The region selector may select one region at a time or it may select all regions a priori, before the compilation process begins on any region. The results in this paper were generated using a region formation algorithm that considered only profile information. The importance of the additional factors discussed in Section 4.1 will be investigated.

Region Compilation. Once the region is selected, the compiler determines the best compilation path based upon the region's characteristics. The region characteristics that may be relevant to the compiler include topology, content, and previously applied transformations. A function-based compiler does not require this capability, because within such a compiler, a suite of transformations are applied to the function in a rigid phase order. The phase ordered application of transformations over a function implies that each phase of compilation is applied to every basic block in the function before the next phase begins.

For example, global optimization is typically applied to an entire function prior to scheduling. In a similar manner, the region-based compiler applies a transformation over the entire region before applying a subsequent transformation. However, a region has a scope that differs from the function it resides in. Thus, basic blocks in different regions may be in two completely different phases of the compilation process.

The fact that basic blocks in different regions may be at different phases in the compilation process provides a region-based compiler with a potential advantage over a function-based compiler. Compensation code generated while applying a transformation to a region may be pushed into unprocessed regions. Consider the application of global optimization, followed by scheduling, to a region. Any resulting compensation code may be pushed into a neighboring region where it will be subject to optimizations applied when that region is processed. This is not the case for a function-based compiler, since the entire function is optimized prior to scheduling and reapplication of the optimizer after scheduling may destroy the schedule.

Region Boundary Conditions. Separate compilation of a program using a traditional function-based compiler is facilitated by the fact that the boundary conditions of a function are fixed. The variables live across the single entry point and single exit point of a function are well defined by the parameter passing convention. However, a region is an arbitrary partition of the program's con-

trol flow graph. There may be any number of variables live across each region's entry and exit points. This live variable information and any other required information, such as memory dependence information, may change as a result of transformations applied to the current region.

Separate compilation of regions also requires the compiler to maintain register allocation and scheduling information at the region boundary points to ensure that regions can be reconciled. This capability has been implemented within the Multiflow compiler, which applies a combined scheduling and register allocation technique on individual traces. For correctness, the scheduler must take into account information on processor resources and register bindings at trace boundary points [1]. The register binding information is communicated by means of a *value-location mapping* data structure [22]. In general, a region-based compiler must maintain, update, and propagate all of this boundary information to a degree that guarantees correctness and allows efficient optimization.

Partial Inlining. Rather than perform aggressive inlining a priori, the inline expansion decisions could be made during the region selection process. The region formation algorithm could be allowed to cross function boundaries and grow regions inter-procedurally. Once the region is selected, only the desired portions of the called functions need be inlined rather than the entire function body. Recall the **eqn** example from Section 3.1. Assume an inter-procedural region selection algorithm selected the shaded basic blocks in Figure 5 to comprise the region, only the shaded blocks in the function *gtc* need be inlined. This reduces the code expansion that results from aggressive inlining prior to region formation. The region-based compilation framework may provide a directed method for performing partial inlining.

Self-Recursion. The results from Figure 8 show that self-recursion can prevent important cycles from being exposed to the compiler in some programs. Inlining self-recursive functions does not increase the amount of visible cyclic code since the cycle is still hidden by a subroutine call. All that is achieved is an increase in the amount of code visible to the compiler. A technique that can transform a general self-recursive function into iterative cyclic code has definite merit. Such a transformation would make the inter-procedural cycle visible to the compiler and should further reduce code expansion.

6 Related Work

In addition to the previously mentioned work done in the Multiflow compiler, Mahadevan and Ramakrishnan have proposed a global scheduling technique that operates over regions within a function [23]. In this approach, a *region* is a single entry sub-graph of a function control flow graph. The region-based compilation technique is more general than this technique in two important aspects: First, the definition of region for our technique places no restrictions on the subset of control flow nodes that make up a region. In addition, the technique described in this paper applies to the entire compilation process.

The term *region* has been used before in several contexts different from that of region-based compilation. For example, the *region scheduling* approach proposed by Gupta and Soffa uses an extended Program Dependence Graph representation [24] to support global code scheduling [25], allowing code motion between regions consisting of control-equivalent program statements. Although this technique provides a vehicle for efficient global code scheduling, the compilation unit remains an entire function.

7 Summary

The traditional function-based framework for compilation is not suitable for an aggressive ILP compiler. The function-based partitioning presents the compiler with compilation units which hide valuable optimization opportunities. By utilizing profile information to repartition the program into regions, the compiler may apply aggressive inlining to overcome the deficiencies of the function-based partitioning. The resulting compilation units allow aggressive optimization in the presence of large function bodies and have the potential to result in more efficient code.

In addition to the on going region-based compilation work in the IMPACT compiler group, HP Labs is actively investigating the area. HP Labs is developing an ILP research compiler, called Elcor, that supports region-based compilation. Both compilers are being used to investigate region-based program analysis, optimization, scheduling, and register allocation.

Acknowledgments

This paper and the underlying research have benefited from discussions with Santosh Abraham, Sadun Anik, Richard Johnson, Vinod Kathail, Scott Mahlke, and Mike Schlansker at HP Labs. The authors would also like to thank John Gyllenhaal, Grant Haab, all the members of the IMPACT research group, and the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly.

This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Intel Corporation, Advanced Micro Devices, Hewlett-Packard, SUN Microsystems, and AT&T GIS.

References

[1] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donell, and J. C. Ruttenberg, "The Multiflow trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, January 1993.

[2] R. Allen and S. Johnson, "Compiling C for vectorization, parallelization, and inline expansion," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 241–249, June 1988.

[3] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling realistic C programs," in *Proceedings of the ACM SIGPLAN 1989 Conference on*

Programming Language Design and Implementation, pp. 246–257, June 1989.

[4] J. W. Davidson and A. M. Holler, "Subprogram inlining: A study of its effects on program execution time," *IEEE Transactions on Software Engineering*, vol. 18, pp. 89–101, February 1992.

[5] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183–198, October 1981.

[6] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.

[7] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.

[8] K. Ebcioglu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 69–79, December 1987.

[9] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 308–317, June 1988.

[10] K. Ebcioglu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," in *Languages and Compilers for Parallel Computing*, pp. 213–229, 1989.

[11] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," in *Proceedings of Supercomputing '90*, November 1990.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.

[13] 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 technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[14] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 63–74, December 1994.

[15] M. Schlansker, V. Kathail, and S. Anik, "Height reduction of control recurrences for ILP processors," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 40–51, December 1994.

[16] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[17] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu, "Superblock formation using static program analysis," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.

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

[19] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pp. 98–105, June 1982.

[20] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 501–536, October 1990.

[21] R. Gupta, M. L. Soffa, and D. Ombres, "Efficient register allocation via coloring using clique separators," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 370–386, May 1994.

[22] S. Freudenberger and J. Ruttenberg, "Phase ordering of register allocation and instruction scheduling," in *Code Generation - Concepts, Tools, Techniques*, pp. 146–170, May 1992.

[23] U. Mahadevan and S. Ramakrishnan, "Instruction scheduling over regions: A framework for scheduling across basic blocks," in *Proceedings of the 5th International Conference on Compiler Construction*, pp. 419–434, April 1994.

[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, July 1987.

[25] R. Gupta and M. L. Soffa, "Region scheduling: An approach for detecting and redistributing parallelism," *IEEE Transactions on Software Engineering*, vol. 16, pp. 421–431, April 1990.