REGION-BASED COMPILATION

BY

RICHARD EUGENE HANK

B.S., University of Illinois, 1990
M.S., University of Illinois, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

REGION-BASED COMPILATION

Richard Eugene Hank, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1996
Wen-mei W. Hwu, Advisor

The increasing amount of instruction-level parallelism (ILP) required to fully utilize high issue-rate processors has forced the compiler to perform more aggressive analysis, optimization, parallelization and scheduling on the input programs. Yet, the compiler designer must scale back the use of aggressive transformations in order to contain compile time and memory usage. The root of the problem lies in 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 provides a convenient partition of the program. However, the size and contents of a function may not provide the best environment for aggressive analysis and optimization. This dissertation presents a technique in which the compiler is allowed to repartition the program into more desirable compilation units, called *regions*. Placing the compiler in control of the size and contents of the compilation unit reduces the importance of the algorithmic complexity of the applied transformations, allowing more aggressive transformations to be applied while reducing compilation time.

The region concept has been traditionally applied within an ILP compiler only in the context of code scheduling. This dissertation proposes extending the concept of region partitioning to the entire compilation process. The implications of region-based compilation to the design of an ILP compiler will be assessed in the context of classical, ILP optimization and register allocation. A quantitative analysis is performed to determine the quality of the code produced by a region-based ILP compiler as compared to a function-based ILP compiler, as well as the compilation time and memory usage benefits afforded by region-based compilation units.

DEDICATION

To the most important people in my life: Anne, Charlie, Jeff, Lynda, and Richard.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Wen-mei W. Hwu, for his keen insight and guidance throughout my graduate studies. I look forward to applying in my future career the knowledge I have gained and I hope to have the honor and pleasure of working with him again.

During my tenure as a graduate student, I also had the honor of working with Dr. Bob Rau at Hewlett-Packard Laboratories. My many discussions with Bob and the other members of his research group: Santosh Abraham, Sadun Anik, Richard Johnson, Vinod Kathail, Scott Mahlke, and Mike Schlansker were invaluable in the development of this thesis.

This research would not have been possible without the support of the members of the IMPACT research group, both past and present. Members of the group were always willing to provide any assistance required in the form of research discussions, practice talks, reviewing papers, or software enhancements. Scott Mahlke provided valuable advice through innumerable conversations. The optimizers developed during his thesis research were an instrumental part of this research. Roger Bringmann was always willing to listen to my brainstorms as both my officemate and roommate. His research provided the scheduling environment used in this thesis. John Gyllenhaal also survived having me as an officemate and this thesis benefited from his insight. John is responsible for the profiling and simulation environment used in this thesis. Brian Deitrich was responsible for the function inliner used for many of these experiments. Dave Gallagher, Grant Haab, and Dan Lavery put up with my frequent sojourns into their office for no apparent reason. Sabrina Hwu provided much needed advice during my many years as a member of this group; she is truly an asset to the IMPACT group.

I would like to thank my parents, Charles, Lynda, and Richard, and my brother, Jeff for their love and encouragement during my many years in graduate school. I would also like to thank them for their emotional support when things were difficult.

Most importantly, I would like to thank Anne. During that last two years of my graduate studies Anne was there to help me celebrate the many good times and to pick me up during a few bad ones. Without her love and emotional support, this thesis would never have been written.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 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 trade off the use of aggressive ILP techniques and compiler performance in terms of compile time and memory utilization. In situations in which 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.

To satisfy the need for more ILP, compilers increasingly resort to inlining to support interprocedural 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.

This dissertation introduces region-based compilation. Under this framework, which is a generalization of current function-based compilation approaches, the fundamental compilation unit is selected 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 transformations in all phases of compilation will be applied. This approach was used in a more restricted context within the Multiflow compiler where combined scheduling and register allocation are applied to to individual 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.

Region-based compilation units provide a compilation environment that is completely under the control of the compiler. The compiler may determine the size and contents of the compilation units to be operated upon. Reducing the problem size 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 may simplify the task of engineering a production quality ILP compiler. The use of profile information to determine region contents allows the compiler to select compilation units that more accurately reflect the dynamic behavior of the program and are more efficiently optimized. As a result the compiler may produce better quality code.

This dissertation investigates the implications of region-based compilation on the implementation of an ILP compiler. A region-based compiler has been implemented within the IMPACT compiler framework to quantify the benefits of applying classical optimization and register allocation over regions. In addition, this implementation allows a direct comparison of the compile time and memory usage benefits of region-based compilation.

1.1 Contributions

The four major contributions of this dissertation are discussed below.

1. The designer of an ILP compiler must trade off the use of aggressive transformations required to expose the ILP necessary to utilize the processor with the need to contain compile time and memory usage. All transformations must be applied in such a way that any code expansion does not adversely affect later transformations. This thesis asserts that the problem lies in the current function-based compilation framework. The function-based partition of a program is designed with software engineering in mind, not compilation of the program. A quantitative analysis is done to assess the quality of the function-based partition and to assess the effect of aggressive code expanding transformations on compilation time and memory usage.

2. This thesis introduces region-based compilation, a generalization of function-based compilation, as an effective framework to contain compile time and memory usage during the application of aggressive transformation. Under this framework, the compiler repartitions the program into *regions*, which serve as the fundamental compilation unit for all phases of the compilation process. This thesis extends the region-based approach traditionally

used during code scheduling to encompass the entire compilation process and discusses the implications this has to compiler design. A quantitative analysis is done to investigate the ability of region-based compilation units to contain problem size in the presence of extensive code expansion.

3. Traditionally global classical optimizations [5] are applied to the entire function to maximize available optimization opportunities. Within a region-based compiler, the scope of optimization is constrained to the same region used for scheduling. Reducing the scope of the global optimizer has two effects. First, reducing the problem size reduces the time and memory required for application of the nonlinear algorithms employed. Second, reducing the scope will also affect the quality of the optimized code. This thesis investigates the techniques to allow separate optimization of regions, including encapsulation of the region in a manner equivalent to functions and the maintenance of dataflow information at region boundaries. A detailed analysis is done to assess the compile time and memory usage improvements afforded by region-based compilation, as well as optimization quality.

4. Similarly, graph coloring-based register allocation [6] is typically applied to the entire function to give the register allocator as much context as possible. Reducing the register allocation scope again reduces register allocation time since register allocation is $O(n^2)$, and the quality of the register allocation is possibly improved. Previous region-based register allocation techniques were restricted to syntactical [7] or trace-based regions [8]. This thesis investigates the requirements for separate register allocation of arbitrary regions. A detailed analysis is done to assess the compile time and memory usage benefits and the quality of the register allocation when applied to profiled-based regions.

1.2 Overview

This dissertation is composed of seven chapters. Chapter 2 presents an overview of the organization and operation of the IMPACT compiler. All compiler techniques discussed in this dissertation are implemented within the framework of the IMPACT compiler.

The current backend compilation environment and methodology are discussed in Chapter 3. Several deficiencies in the current function-based approach are presented. The use of inlining to overcome some of these deficiencies and the effect of aggressive inlining on compiler performance are investigated. Chapter 4 introduces region-based compilation as a generalization of the current function-based approach to compilation. Region-based compilation provides an effective framework to contain compile time and memory usage during the application of aggressive transformation. This chapter discusses the ramifications of this concept to the design of an ILP compiler.

Chapter 5 investigates the application of classical optimizations to region-based compilation units. Separate optimization of regions requires the compiler to maintain variable liveness conditions at region boundaries. A quantitative analysis is done to investigate the quality of region-based optimized code, as well as the compile time and memory usage benefits. Chapter 6 discusses the issues involved in separate register allocation of region-based compilation units. The boundary condition information required for separate register allocation of regions will be investigated. A quantitative analysis is done to investigate the quality of region-based register allocation, as well as the compile time and memory usage benefits. Finally, in Chapter 7, conclusions and directions for future research are given.

CHAPTER 2

OVERVIEW OF THE IMPACT COMPILER

The implications of region-based compilation are studied within the framework of the IMPACT compiler. The IMPACT compiler is a retargetable, optimizing C compiler being developed at the University of Illinois to investigate architectural and compilation techniques to support ILP processors. A block diagram of the IMPACT compiler is presented in Figure 2.1. The compiler is divided into two distinct parts based on the level of intermediate representation (IR) used. The first level IR, *Pcode*, is a parallel C code representation with loop constructs intact. In Pcode, memory dependence analysis [9], [10], loop-level transformations [11], and memory system optimizations [12], [13] are performed. In addition, statement level profiling and function inline expansion are performed at this level [14], [15], [16].

The second level of IR in the IMPACT compiler, referred to as *Lcode*, is a generalized register transfer language similar in structure to most load/store processor assembly instruction sets. Lcode is logically subdivided into two subcomponents, the machine independent IR, *Lcode*, and the machine specific IR, *Mcode*. The data structures for both the Lcode and Mcode are identical. The difference is that Mcode is broken down such that there is a one-to-one mapping between Mcode operations and the operations in the target processor's assembly language. Therefore, to convert Lcode to Mcode, the code generator breaks up Lcode operations into one or more operations which directly map to the target architecture. Lcode operations are broken up for a variety of reasons including limited addressing modes, limited opcode availability, ability to specify a literal operand, and field width of literal operands.

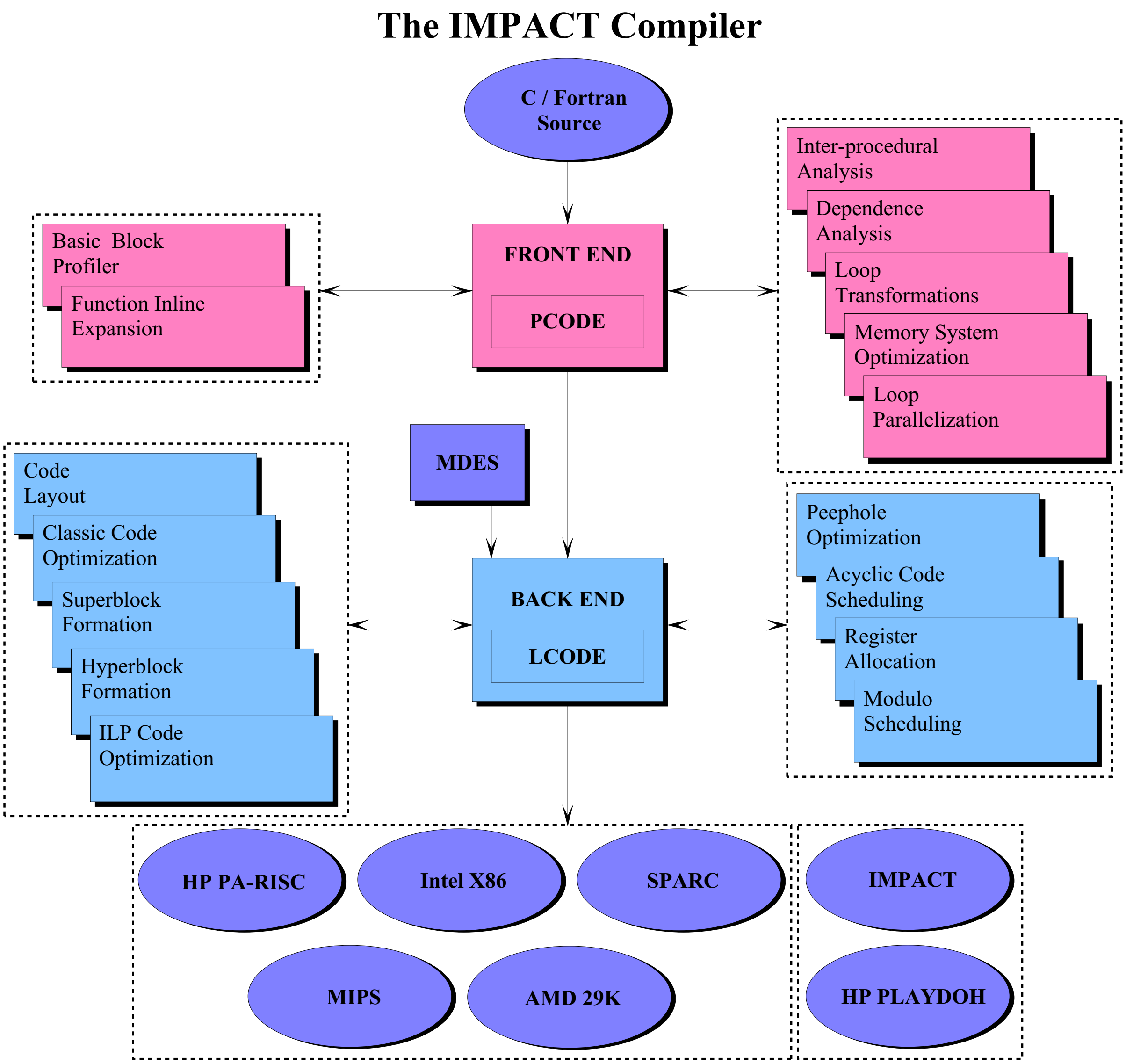


Figure 2.1 The IMPACT compiler.

The region-based compilation techniques, which are the focus of this thesis, are performed at the Lcode level. Support for region-based compilation, includes region formation using execution profile information, as well as region-based optimization and register allocation. The existing Lcode modules have been extended to operate on regions.

At the Lcode level, all machine-independent classic optimizations are applied [17]. These include constant propagation, forward copy propagation, backward copy propagation, common subexpression elimination, redundant load elimination, redundant store elimination, strength reduction, constant folding, constant combining, operation folding, operation cancellation, code reordering, dead code removal, jump optimization, unreachable code elimination, loop invariant code removal, loop global variable migration, loop induction variable strength reduction, loop induction variable elimination, and loop induction variable re-association. Additionally at the Lcode level, interprocedural safety analysis is performed [18]. This includes identifying safe instructions for speculation and function calls that do not modify memory (side-effect free). The effects of applying classical optimizations on a per-region basis will be discussed in Chapter 5.

After classical optimization, ILP transformations are applied. This involves the application of superblock code transformation and optimization [19]. When predicated execution support is available in the target architecture, hyperblock techniques [20], in addition to superblock techniques, are used as the underlying compilation structure. All superblock optimization techniques have also been extended to operate on hyperblocks. In addition, a suite of hyperblock-specific optimizations to further exploit predicated execution support is available.

After ILP transformation and optimization, code generation is performed to translate Lcode into the assembly language used by the target processor. Two of the most significant components of code generation are the instruction scheduler and register allocator, both of which are common modules shared by all code generators. Scheduling is performed via either acyclic global scheduling [18],[21] or software pipelining using modulo scheduling [22], [23]. Acyclic global scheduling is applied both before register allocation (prepass scheduling) and after register allocation (postpass scheduling) to generate an efficient schedule. For software pipelining, loops targeted for pipelining are identified at the Pcode level and marked for pipelining. These loops are then scheduled using software pipelining, and the remaining code is scheduled using the acyclic global scheduler. In addition to control speculation, both scheduling techniques are capable of exploiting architectural support for data speculation to achieve more aggressive schedules [10],[24],[25].

Graph coloring-based register allocation is utilized for all target architectures [26]. The register allocator employs execution profile information, if it is available, to make more intelligent decisions. The extensions to the register allocator to support region-based register allocation will be discussed in Chapter 6. For each target architecture, a suite of specially tailored peephole optimizations is performed. These peephole optimizations are designed to remove inefficiencies during Lcode to Mcode conversion and to take advantage of specialized opcodes available in the architecture.

A detailed machine description database, *Mdes*, for the target architecture is also available to all Lcode compilation modules [27]. The Mdes contains a large set of information to assist with optimization, scheduling, register allocation, and code generation. Information such as the number and type of available function units, size and width of register files, instruction latencies, instruction input/output constraints, addressing modes, and pipeline constraints is provided by the Mdes. The Mdes is queried by the optimization phases to make intelligent decisions regarding the applicability of transformations. The scheduler and register allocator rely more heavily on the Mdes to generate efficient as well as correct code.

Seven architectures are actively supported by the IMPACT compiler. These include the AMD 29K [28], MIPS R3000 [29], SPARC [30], HP PA-RISC, and Intel X86. The other two supported architectures, IMPACT and HPL Playdoh [31], are experimental ILP architectures.

These architectures provide an experimental framework for compiler and architecture research. The IMPACT architecture is a parameterized superscalar processor with an extended version of the HP PA-RISC instruction set. Varying levels of support for speculative execution and predicated execution are available in the IMPACT architecture. For this thesis, all experiments utilize the IMPACT architecture with varying parameters.

CHAPTER 3

CURRENT BACKEND COMPILATION

3.1 Compilation Environment

Traditionally, the compilation process has been built assuming functions as the unit of compilation. This environment is a direct result of the way programs are written. Software engineering and object-oriented programming practices encourage the software designer to break up a program in a modular manner that facilitates the reuse and maintainability of the source code [32]. The problem to be solved is partitioned into a number of distinct modules or functions each of which solves a portion of the problem as illustrated in Figure 3.1. Since each of these functions is a self-contained entity, the function-based partition of the program provides a convenient way to break up the task of compiling a program. Typically, the compiler processes each function of the program in turn, applying a phase ordered suite of transformations.

3.2 Compilation Methodology

Current compilers typically compile a program on a function-by-function basis. Each function is compiled completely before proceeding to the next function. Figure 3.2 illustrates the phases applied in the backend of the compiler during compilation of a function, $\mathbf{F}_x$, consisting of basic blocks, $\mathbf{BB}_1$ **-** $\mathbf{BB}_n$. This first phase involves application of classical optimizations [5]. These optimizations, such as common subexpression elimination and copy propagation, are applied locally to each basic block, but they are also applied globally across all basic blocks to

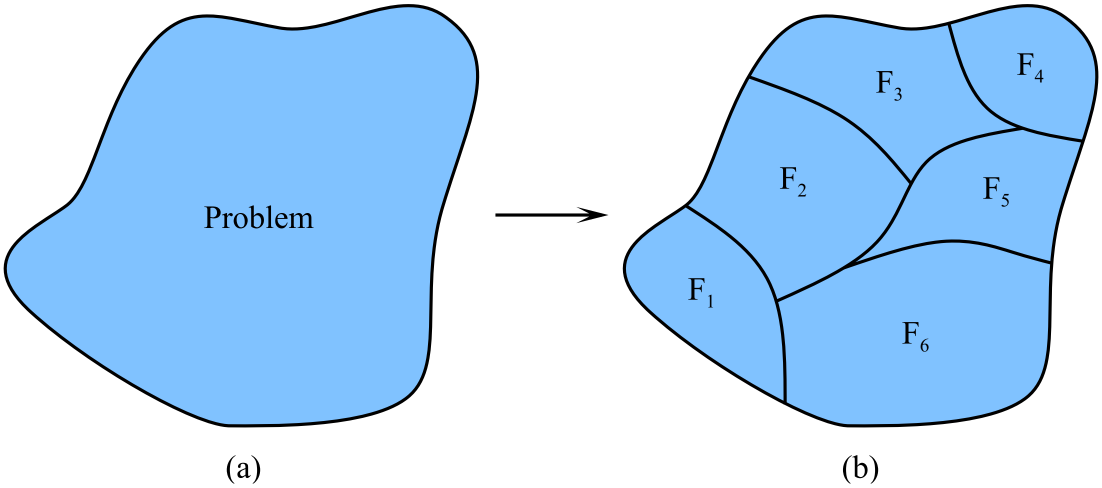


Figure 3.1 Problem to be solved: (a) original problem, (b) problem partitioned into functions.

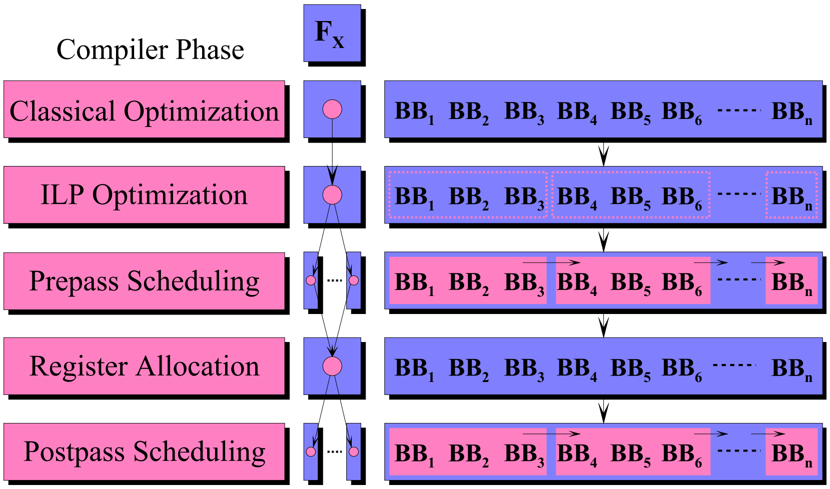


Figure 3.2 Current backend compilation of a function.

take advantage of optimization opportunities that often exist between basic blocks. Thus, to maximize the scope of the classical optimizer, the entire function is optimized as a unit.

After classical optimization, the compiler may apply various ILP transformations, such as superblock or hyperblock formation, to enhance and expose the instruction level parallelism inherent in the function. As with classical optimization, the ILP transformations are applied to the entire function before compilation proceeds to the next phase, code scheduling. The same is true for the register allocation phase of compilation. The scope of the register allocator is expanded beyond the basic block boundaries, to encompass the entire function and improve the quality of the register allocation [6], [33].

Applying each compilation phase to the function body as a whole requires the compilation process to follow a rigid phase ordering in the manner transformations are applied. In addition, an implementation of an ILP compiler under this framework must ensure that transformations are applied such that any code expansion will not adversely affect compilation time. This method of compilation will be termed *horizontal* compilation for the remainder of this thesis. The code scheduling phases of compilation provide an exception to this general behavior and for this very reason tend to be region based. Figure 3.2 shows that the scope of the prepass scheduler is confined to subregions of the function, such as the region containing $\mathbf{BB}_1$, $\mathbf{BB}_2$, and $\mathbf{BB}_3$. The scope of scheduling has evolved beyond basic block boundaries to encompass larger units, such as loops [34], traces [35], superblocks [19], or other regions [20], [36]. In general, functions tend to be too large a unit for the application of aggressive ILP scheduling techniques to be tractable in the context of a production compiler.

3.2.1 Deficiencies of a function-based partition

Throughout the compilation process there is an implicit assumption that the body of a function provides the most desirable environment for the compiler to generate efficient output code. In reality, this is not the case. Consider the two functions shown in Figure 3.3. Blocks 1-4 of function A form a very frequently iterated loop. Block 3 contains a subroutine call to function B. As a result, function B, which consists of blocks 5-8, is very frequently executed.

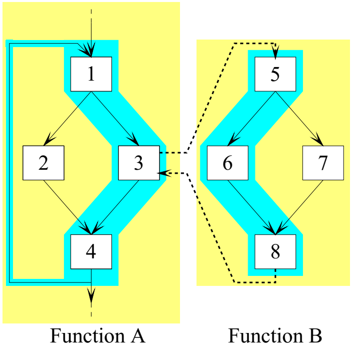


Figure 3.3 Example of an undesirable function-based partition.

The shaded portions represent the dynamic behavior of these two functions and indicate that blocks 2 and 7 are infrequently executed.

As discussed above, while compiling function A, the scope of the compiler is the entire contents of function A, and the contents of function B are hidden. This fact affects the compilation of function A in two ways. First, the presence of the subroutine call in block 3 presents an obstacle to aggressive optimization and scheduling, known as a hazard [37]. Second, since the compiler has no visibility to the contents of function B, the compiler must make conservative assumptions regarding the memory access behavior of function B. These assumptions will constrain aggressive optimization and scheduling of function A. Aggressive transformation of function B is inhibited in a similar manner. The compiler must make conservative assumptions regarding the memory access behavior of function A. In addition, the compiler cannot take advantage of the fact that function B is part of a cycle and apply loop-based transformations, because this information is hidden from the compiler.

In this example, the function-based partition of the program is potentially hiding variable aliasing information, as well as valuable optimization opportunities. Perhaps more important, however, is the fact that the function-based partition is hiding the true control flow structure of the program. This makes the function-based partition of the program rather undesirable from the point of view of the compiler.

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 control flow cycles, 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 [19], [34], [35], [38]-[45]. Exposing more cycles to an aggressive ILP compiler increases the likelihood that these techniques may be applied to generate more efficient code.

3.2.2 Dynamic distribution of cyclic and acyclic codes

The prevalence of the situation in which the function-based partition is hiding cyclic code from the compiler can be determined by examining the dynamic distribution of cyclic and acyclic code within functions. In order for a program to execute for an appreciable amount of time, it must contain a cycle. If the program spends a significant amount of time executing operations that appear acyclic to the compiler at the function level, this implies the presence of cycles spanning function boundaries. These interprocedural cycles are caused by the presence of subroutine calls within loop bodies, such as in Figure 3.3, and procedure call cycles, i.e., recursion.

Figure 3.4 contains the dynamic distribution of cyclic and acyclic codes within the function bodies of several non-numeric programs. For a detailed description of these benchmarks and the inputs used to generate these and all subsequent results in this thesis, the reader is referred to Appendix A. The large percentage of time spent in cyclic code within the programs validates

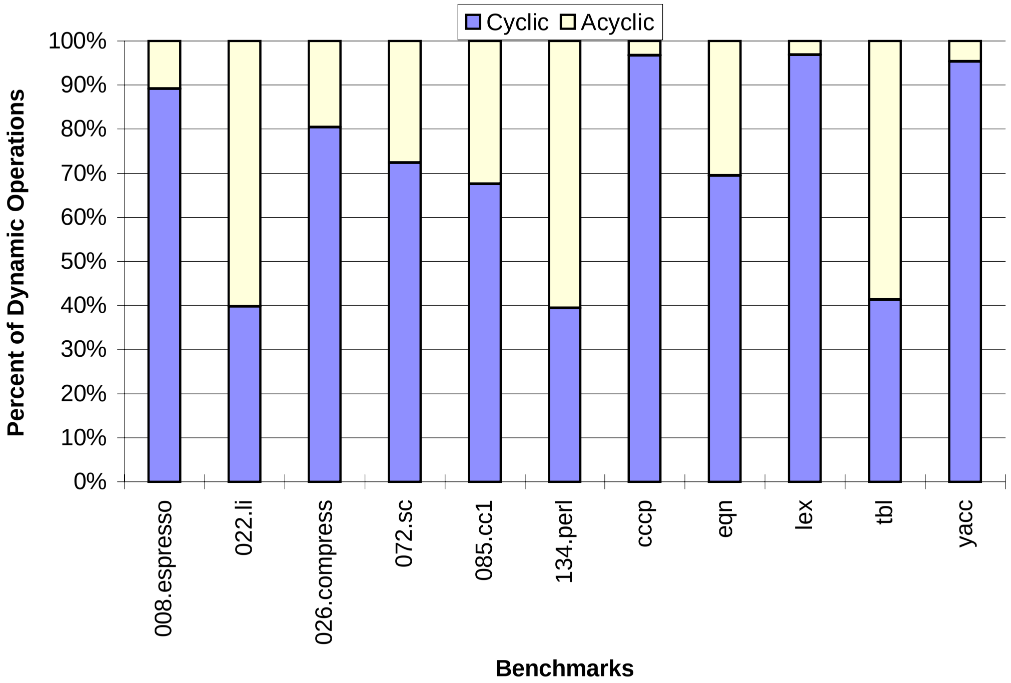


Figure 3.4 Intraprocedural distribution of dynamic acyclic and cyclic codes.

the importance of cycles. The programs **cccp**, **lex**, and **yacc**, 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. 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 **134.perl** spend more than 50% of their time in intraprocedural acyclic code. In these instances, it is apparent that the function-based partition of the program hides the existence of these cycles, as well as other valuable optimization opportunities. To a lesser degree, the situation is the same for **eqn**, **compress**, **072.sc**, and **085.cc1**.

The presence of cyclic code does not imply that the compiler will be able to apply cyclic techniques. The presence of subroutine calls, ambiguous memory dependences, or the sheer size of the cycle may prevent aggressive optimization. However, by exposing the interprocedurally coupled portions of the program to the compiler, the potential for an aggressive compiler to

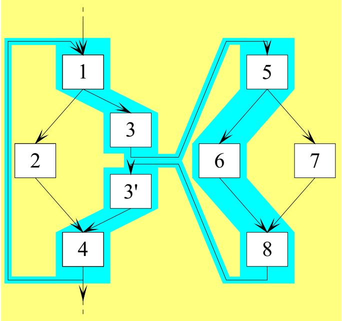


Figure 3.5 Application of inlining to the functions of Figure 3.3.

apply more optimizations is increased. Function inlining is the only well-known technique that will allow interprocedurally coupled portions of the program to be assembled into the same compilation unit. Application of function inlining to the example in Figure 3.3 successfully exposes the entire cycle to the compiler by placing the contents of function B into function A, as shown in Figure 3.5.

3.3 Function Inlining

Traditionally, the goal of function inlining has been to eliminate the overhead of frequent subroutine calls [3],[4]. However, to further improve output code performance, compilers increasingly resort to inlining to support interprocedural optimization and scheduling. 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 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 interprocedural analysis [4].

```
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 3.6 Source code for functions *getline* and *gtc*.

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 3.3.2.

3.3.1 Benefits of inlining

The function-based partitioning of the non-numeric program **eqn** provides an example of the potential ILP benefits of inlining. Intraprocedurally, **eqn** appears to have a large percentage of frequently executed code that does not occur within the body of a loop. Figure 3.4 shows that **eqn** appears to spend 30% of its execution time within acyclic code. This is the result of several interprocedural cycles that are caused by the presence of subroutine calls within the bodies of frequently iterated loops.

One such interprocedural cycle spans the two functions *getline* and *gtc*. The source code for these two functions is shown in Figure 3.6. The function *getline* contains a very frequently iterated loop which calls the function *gtc* once every 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 [19]. The control flow graph (CFG) for the function *getline* is shown in Figure 3.7(a). 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 3.7(b). 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 3.7(a). 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 3.7(a). These basic blocks correspond to the shaded portion of the *getline* source code in Figure 3.6. The contents of the resulting superblock after optimization is shown in Figure 3.8(a). Scheduling this superblock loop for an eight-issue, fully uniform machine, yields the issue times shown to the right of Figure 3.8(a). 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 3.7(b). The corresponding source code lines are shaded in Figure 3.6. Again, the contents of the resulting superblock after optimization and scheduling for the same eight-issue, fully uniform machine are shown in Figure 3.8(b). This superblock requires nine cycles to completely execute. Thus one loop iteration requires 13 cycles *not* including the subroutine call overhead.

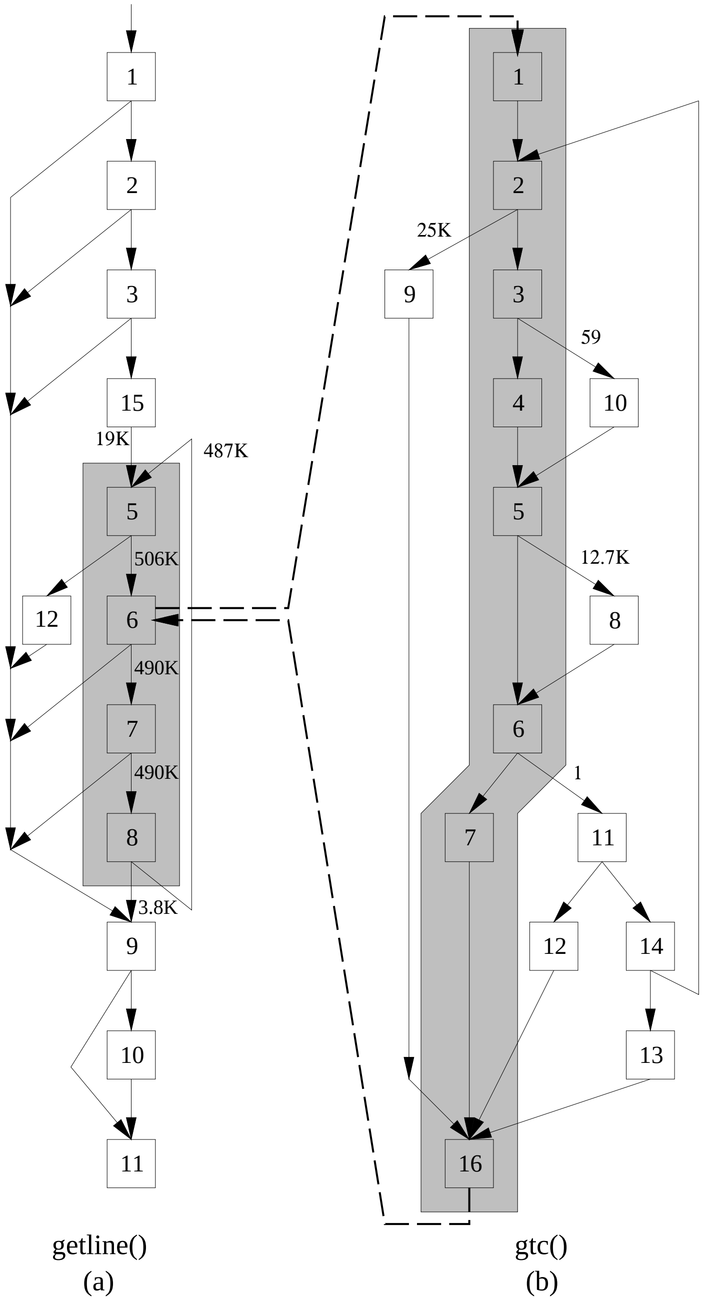
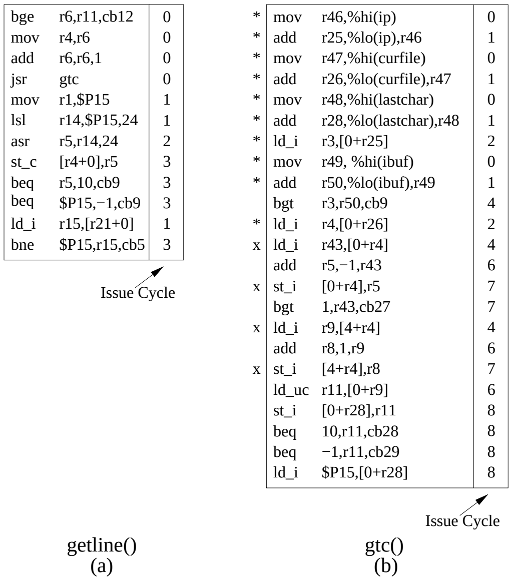


Figure 3.7 Control flow graphs for the functions (a) *getline* and (b) *gtc*.



```
bge   r6,r11,cb12     0
mov   r4,r6           0
add   r6,r6,1         0
jsr   gtc             0
mov   r1,$P15         1
lsl   r14,$P15,24     1
asr   r5,r14,24       2
st_c  [r4+0],r5       3
beq   r5,10,cb9       3
beq   $P15,−1,cb9     3
ld_i  r15,[r21+0]     1
bne   $P15,r15,cb5    3
```

getline()
(a)

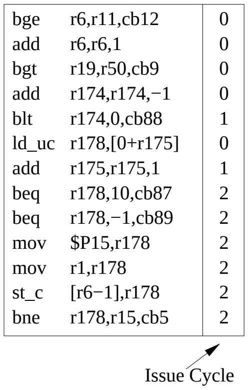
```
* mov   r46,%hi(ip)              0
* add   r25,%lo(ip),r46          1
* mov   r47,%hi(curfile)         0
* add   r26,%lo(curfile),r47     1
* mov   r48,%hi(lastchar)        0
* add   r28,%lo(lastchar),r48    1
* ld_i  r3,[0+r25]               2
* mov   r49, %hi(ibuf)           0
* add   r50,%lo(ibuf),r49        1
  bgt   r3,r50,cb9               4
* ld_i  r4,[0+r26]               2
x ld_i  r43,[0+r4]               4
  add   r5,−1,r43                6
x st_i  [0+r4],r5                7
  bgt   1,r43,cb27               7
x ld_i  r9,[4+r4]                4
  add   r8,1,r9                  6
x st_i  [4+r4],r8                7
  ld_uc r11,[0+r9]               6
  st_i  [0+r28],r11              8
  beq   10,r11,cb28              8
  beq   −1,r11,cb29              8
  ld_i  $P15,[0+r28]             8
```

gtc()
(b)

Figure 3.8 Superblock contents prior to inlining for (a) *getline* and (b) *gtc*.

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* can now be 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 3.7.

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 3.9. 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



```
bge     r6,r11,cb12       0
add     r6,r6,1           0
bgt     r19,r50,cb9       0
add     r174,r174,−1      0
blt     r174,0,cb88       1
ld_uc   r178,[0+r175]     0
add     r175,r175,1       1
beq     r178,10,cb87      2
beq     r178,−1,cb89      2
mov     $P15,r178         2
mov     r1,r178           2
st_c    [r6−1],r178       2
bne     r178,r15,cb5      2
```

Issue Cycle

Figure 3.9 Superblock loop after inlining *gtc* into *getline*.

loop invariant code elimination [5] allows the operations indicated by an (*) in Figure 3.8(b) to be removed. Also, the application of global variable migration [19] allows the operations indicated by an (x) in Figure 3.8(b) to be hoisted outside the superblock loop body. Both of these transformations will be discussed further in Chapter 5. These code optimizations would not be accomplished without compiling functions *getline* and *gtc* as a single unit. Scheduling this superblock for the same eight-issue, fully uniform machine actually produces a schedule requiring only 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 **eqn** on an eight-issue, fully-uniform IMPACT machine shows that the program requires 21.2M cycles to execute before inlining. Of that, 7.24M cycles are spent within the functions *getline* and *gtc*. After inlining and optimization, only 1.9M cycles are spent within *getline* and the overall execution time is 15.9M cycles. This results in a speedup of 3.8 within the function *getline* which corresponds closely to fact that the cycle count of the loop is four times shorter after inlining. Overall, the benchmark sees a speedup of 1.33.

Table 3.1 Intraprocedural distribution of dynamic acyclic and cyclic codes.

	Percent of Inlined Call Sites									
Benchmark	**0%**	**5%**	**10%**	**15%**	**20%**	**25%**	**30%**	**40%**	**60%**	**80%**	**100%**
008.espresso	89.10%	97.36%	97.46%	97.61%	97.79%	97.88%	97.97%	98.05%	98.15%	98.24%	98.30%
022.li	39.74%	49.51%	61.79%	65.55%	69.50%	73.22%	78.99%	83.52%	84.30%	85.91%	85.93%
026.compress	80.46%	80.46%	80.46%	80.46%	80.46%	80.46%	80.46%	80.46%	100.00%	100.00%	100.00%
072.sc	72.34%	84.69%	91.50%	92.63%	96.18%	98.12%	98.77%	99.17%	99.79%	99.86%	99.90%
085.cc1	67.54%	78.04%	81.12%	83.43%	85.51%	86.54%	87.49%	88.82%	90.52%	90.79%	90.86%
134.perl	39.45%	64.48%	71.58%	75.42%	77.22%	79.62%	79.64%	80.07%	80.70%	80.72%	80.72%
cccp	96.76%	97.08%	97.27%	97.31%	97.77%	97.88%	98.16%	98.25%	98.67%	98.71%	98.80%
eqn	69.39%	78.07%	82.26%	90.28%	95.57%	97.02%	98.98%	99.54%	99.64%	99.70%	99.98%
lex	96.85%	97.85%	99.61%	99.67%	99.72%	99.74%	99.75%	99.79%	99.86%	99.87%	99.88%
tbl	41.32%	51.73%	58.36%	62.89%	68.60%	77.47%	88.15%	93.57%	95.58%	97.51%	98.96%
yacc	95.34%	96.68%	98.29%	98.94%	99.03%	99.25%	99.47%	99.61%	99.84%	99.94%	99.98%

In this instance, inline expansion has provided the compiler with many more ILP optimization opportunities than prior to inlining, yielding significant performance improvement.

3.3.2 Impact on code characteristics

The previous example illustrates the benefits of exposing hidden cycles to the compiler through the use of inlining. 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. Table 3.1 shows how the percentage of code that appears cyclic to the compiler increases as the amount of function inlining increases.

The number at the top of each column is the percent of the total number of inline expandable call sites in the program that were inlined expanded, subject to the following constraints. The inliner was constrained from producing functions larger than 25,000 operations to ensure the feasibility of later experiments on these heavily inlined programs. Also, the inliner was not allowed to inline expand functions taking a variable number of arguments nor was it allowed to inline expand a self-recursive function into itself. Thus, the 100% indicates that all of the expandable call sites were inline expanded, but that does not imply that the program has been fully inline expanded.

For each benchmark, the knee of the curve occurs at the point where 95-97% of the dynamic cyclic code visible at the 100% inline expansion level has been exposed to the compiler. The highlighted entry for each benchmark in Table 3.1 represents this point. The amount of inlining required to achieve this state varies greatly between benchmarks. For example, the benchmark **134.perl** contains 39% cyclic code prior to inlining. With 100% inlining, the cyclic code portion becomes 80.72%. To capture 95% of the 80.72% exposure, one only has to inline expand the first 20% of the expandable call sites. On the other hand, the number of dynamic function calls is spread across more call sites in the benchmark **022.li**, requiring 40% of the expandable call sites be inlined to expose 95% of the dynamic cyclic code. For most of these benchmarks it is unnecessary to inline expand them beyond 30%. Subsequent experiments in this dissertation will be applied to these benchmarks at this desired level of function inline expansion.

With the aid of profile information, it is relatively easy for the compiler to determine the necessary amount of inline expansion. However, in the absence of profile information, the compiler must employ heuristics to determine which call sites to inline expand. As a result, the compiler may perform more inline expansion than is necessary for a benchmark. The 40-80% percent inlining levels will be used to illustrate the effect of over inlining on function size and how this affects compiler performance.

Figure 3.10 graphically illustrates the distribution of cyclic and acyclic codes for the 100% inlined case in Table 3.1. For the most part, it is possible through the application of inlining to expose all of the frequently executed cycles to the compiler. The three exceptions to this are **022.li**, **085.cc1**, and **134.perl**, which still appear to contain 10-20% dynamic acyclic code.

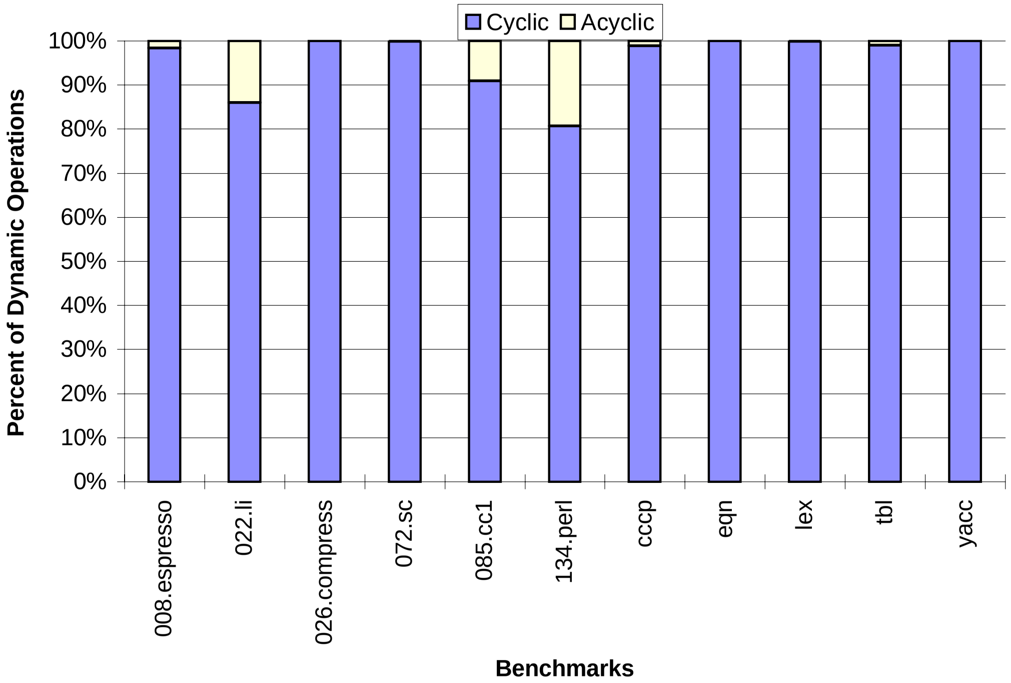


Figure 3.10 Distribution of dynamic acyclic and cyclic codes after aggressive inlining.

The principle reason for this is recursion. Inlining through a recursive cycle does not expose the cycle to the compiler 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 that loop unrolling is applied to increase the scope of the compiler for iterative cycles.

It should be noted that Figure 3.10 gives a somewhat optimistic view of the amount of the program that may benefit from cyclic transformations. The compiler may not be able to apply cyclic transformations to some cycles for several reasons. The cycle may be too large or the control flow structure may not be suitable for the transformations available to the compiler. Also, the cycle may contain a subroutine call that was not inline expandable or other operations that inhibit cyclic optimization.

Table 3.2 Static code growth as the amount of inlining increases.

	Num Opers	Static Code Growth									
Benchmark	**0%**	**5%**	**10%**	**15%**	**20%**	**25%**	**30%**	**40%**	**60%**	**80%**	**100%**
008.espresso	37426	1.02	1.04	1.07	1.16	1.21	1.26	1.43	1.74	2.53	4.52
022.li	11802	1.03	1.10	1.15	1.24	1.38	1.57	2.02	3.22	6.77	8.82
026.compress	1698	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.10	1.10	1.12
072.sc	16046	1.07	1.22	1.27	1.64	1.84	2.15	3.14	4.24	5.07	6.40
085.cc1	134594	1.05	1.11	1.18	1.29	1.36	1.48	1.79	2.81	3.24	4.34
134.perl	64569	1.02	1.05	1.08	1.19	1.44	1.45	1.49	1.61	1.74	2.02
cccp	7129	1.01	1.02	1.02	1.07	1.09	1.17	1.23	1.46	1.55	2.89
eqn	5326	1.04	1.16	1.30	1.45	1.52	1.75	2.47	3.29	3.55	4.56
lex	7549	1.01	1.05	1.08	1.12	1.15	1.18	1.27	1.79	1.95	2.16
tbl	8443	1.03	1.07	1.11	1.16	1.22	1.35	1.51	1.87	2.69	8.18
yacc	6610	1.02	1.05	1.10	1.11	1.15	1.21	1.30	1.55	1.97	2.83

3.3.3 Impact on problem size

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 at which the resulting spill code negates any benefit to be gained from inline expansion [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.

Table 3.2 shows the code expansion resulting from the inlining levels shown in Table 3.1. The worst-case code expansion (100% inlining) ranges from 1.12 to 8.82 times the original code size, with an average increase of 4.3 times the original code size. At the desirable levels of inlining, the average code expansion is 1.33 times, which is much less. However, static code growth itself is not a good indicator of how the inline expansion will affect compilation time and memory usage. There will be a more pronounced effect on compilation time and memory usage if the inline expansion results in a couple of large functions versus several moderately sized functions.

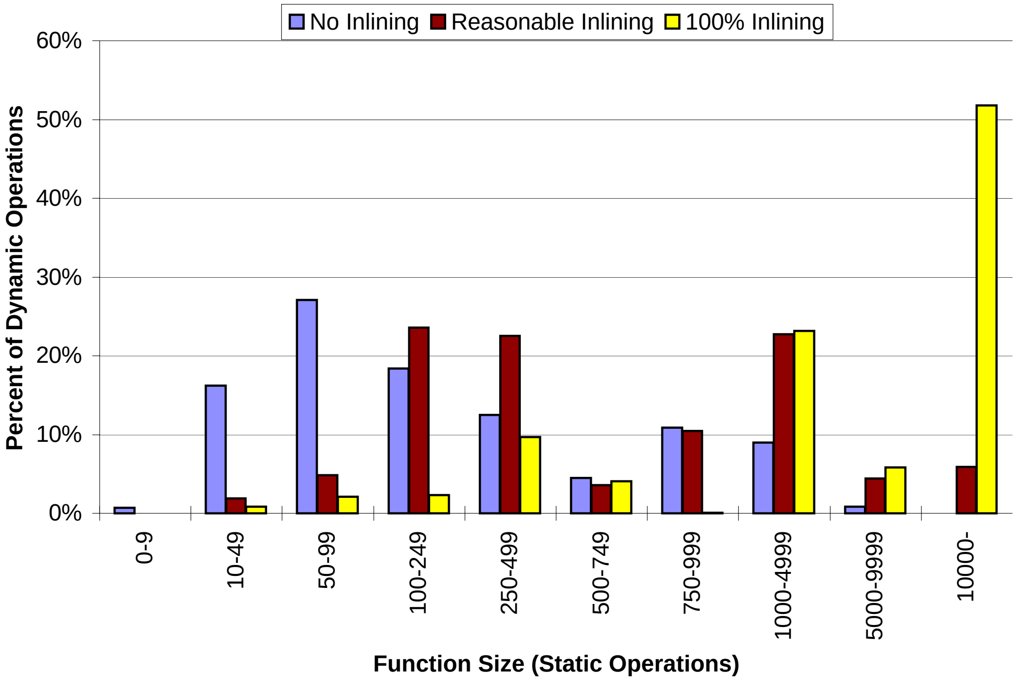


Figure 3.11 Histogram of function size before and after inlining.

The data presented in Figure 3.11 provide better insight into the effect inlining may have on compilation. Figure 3.11 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 11 programs spent 75% of their execution time in functions with fewer than 500 operations,[1] whereas at the 100% inlining level there is a drastic shift to the right. Over 80% of the program's execution time is spent in functions with more than 1000 operations. Even with a reasonable amount of inlining there is a noticeable shift to the right. Though the amount of code expansion at this level is not extreme, the effect of the inline expansion tends to manifest itself in the form of extremely large functions.

[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.

Table 3.3 Percent increase in classical optimization time as inlining increases.

	Time (s)	Increase in Optimization Time as Inlining Increases									
Benchmark	0%	5%	10%	15%	20%	25%	30%	40%	60%	80%	100%
008.espresso	78	1.08	1.15	1.27	2.31	3.06	3.81	12.33	36.33	254.72	*
022.li	15	1.07	1.33	1.47	2.07	3.00	4.40	11.73	126.73	68.27	*
026.compress	5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.20	1.20	1.20
072.sc	84	1.02	1.51	1.63	4.93	6.69	11.00	50.40	159.68	341.20	917.71
085.cc1	790	1.41	2.18	3.29	7.73	10.04	15.31	34.62	50.92	*	*
134.perl	524	1.02	1.19	1.64	3.94	10.64	11.26	12.39	33.48	72.59	*
cccp	29	1.00	1.07	1.03	1.24	1.31	2.17	2.17	4.03	4.52	108.59
eqn	16	1.06	2.94	5.13	8.06	9.13	11.63	34.81	84.44	110.63	260.00
lex	53	1.00	1.09	1.11	2.17	2.38	2.70	3.09	12.75	19.94	23.38
tbl	53	1.04	1.60	1.68	2.02	3.62	4.60	8.13	21.91	43.91	0.00
yacc	39	1.18	1.33	1.67	1.72	2.21	2.97	4.08	7.41	28.51	169.54

Inlining has succeeded in assembling the interprocedurally 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. The next section discusses the effect of increasing function size on compile time and memory usage.

3.3.4 Impact on compiler performance

The amount of time required to perform classical optimizations for the various levels of inlining discussed in previous sections is shown in Table 3.3. The 0% column contains the number of seconds[2] required for classical optimization of each benchmark with no inlining. The remaining columns contain the percent increase in compile time over the no inlining case for that level of inlining.

For all benchmarks, with the exception of **026.compress**, the compilation time increases sharply as the amount of inline expansion is increased. For example, the benchmark **072.sc** incurs a compile time increase of only 2% at 5% inlining; however, at 20% inlining it takes

[2] These and all subsequent execution times were generated on a HP 9000/735 running at 125 MHz with 128MB RAM.

Table 3.4 Percent increase in classical optimization memory usage as inlining increases.

	Mem (MB)	Increase in Memory Usage as Inlining Increases									
Benchmark	0%	5%	10%	15%	20%	25%	30%	40%	60%	80%	100%
008.espresso	2.18	1.00	1.00	1.00	1.27	1.44	1.63	4.52	7.07	18.32	*
022.li	0.64	1.16	1.53	1.53	2.33	3.47	3.47	9.49	34.42	41.22	*
026.compress	1.21	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
072.sc	4.05	1.00	1.00	1.00	2.50	3.37	4.73	15.39	16.91	18.74	29.61
085.cc1	4.91	1.93	1.93	2.61	3.41	4.69	7.08	13.41	30.75	*	*
134.perl	17.96	1.00	1.00	1.24	2.34	2.88	2.88	3.04	4.80	5.45	*
cccp	1.44	1.00	1.00	1.00	1.34	1.42	2.10	2.10	3.00	3.00	11.26
eqn	1.37	1.00	1.46	1.94	2.98	3.37	5.10	7.17	17.09	19.98	35.36
lex	2.67	1.00	1.00	1.06	1.31	1.56	1.59	1.91	3.91	5.10	5.84
tbl	1.95	1.00	1.14	1.14	1.21	1.50	1.66	2.74	5.36	7.14	0.27
yacc	1.53	1.00	1.00	1.00	1.00	1.00	1.20	1.70	1.88	7.02	21.04

4.9 times as long to perform classical optimization. At 100% inlining, classical optimization takes 917 times as long, which is clearly not feasible within a production compiler. Because the benchmark **026.compress** contains only two expandable call sites, there is no change in the benchmark until the 60% inlining level. The drastic increase in compile time across the benchmarks results from the fact that increasing function size is aggravating the algorithmic complexity of the optimization algorithms.

Relatively small increases in static code size can result in significant increases in compile time. The desired level of inlining for **yacc** is 5%. At this level, there is a 2% increase in code size, however, that 2% increase in code size results in an 18% increase in optimization time since most of the code expansion occurs within one function body. The benchmark **134.perl** provides another example. The desired inlining level for **134.perl** is 20%. At this level, there is only a 19% increase in static code size, see Table 3.2, but the classical optimization time has increased almost 4 (3.94) times. Again, the principal reason is that the code expansion is localized to two function bodies.

Table 3.4 contains the memory usage during classical optimization as the amount of inlining is increased. The 0% column contains the memory usage, in megabytes, for the no-inlining case.

The remaining columns contain the percent increase in memory usage over the no-inlining case for that level of inline expansion. As with compile time, there are significant increases in the amount of memory required for classical optimization. For example, the benchmarks **072.sc**, **134.perl**, and **eqn** all require two to three times more memory at their desired level of inlining than they do with no inlining. For **134.perl** this is especially significant since this benchmark requires 18MB of memory with no inline expansion due to several large function bodies. The benchmarks **022.li**, **085.cc1**, and **tbl** require 9.5, 4.7, and 5.4 times the memory, respectively, at the desired level of inlining. This table clearly illustrates the effect of applying algorithms that are non-linear in memory complexity to entire functions as code expansion increases.

From Tables 3.3 and 3.4 it is clear that under extreme code expansion, and in many cases reasonable code expansion, the function bodies are too large for aggressive optimization in a production compiler. The situation will not improve when aggressive ILP transformations are applied to expose parallelism inherent in programs. 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.

CHAPTER 4

REGION-BASED COMPILATION FRAMEWORK

The compilation difficulties that arise as a result of inlining are due to the function-based compilation framework. Aggressive application of inlining can reduce the side effects of the function-based partition, namely, hidden memory aliasing behavior, optimization, and control flow structure. However, inlining tends to exacerbate the algorithmic complexity of the compilation process by increasing function size. In other words, inlining alleviates several symptoms of the problem, but the root cause of the problem still exists, the function-based compilation framework.

Examination of the function-based compilation framework reveals that, regardless of the implementation, the tasks performed by the compiler during the compilation of a program fall into three categories: compilation unit selection, transformation, and state maintenance. Compilation unit selection involves determining the scope of the program over which later transformations will be applied. Transformation involves the application of optimization, scheduling and register allocation techniques, to a particular portion of the program, selected from the repertoire available to the compiler. Finally, state maintenance is the bookkeeping required to maintain correctness during the separate compilation of the selected areas of the program.

Within a function-based compiler, compilation unit selection is trivial since all phases of compilation are typically applied over the entire function, with the exception of code scheduling. The selection and application of transformations is crude in that current function-based compilers tend to apply the available transformations in a rigid phase order. The final category, state maintenance, requires little work on the part of a function-based compiler. The interface between functions is well-defined by the procedure calling convention, meaning that the compiler need not maintain any state information during the separate compilation of each function.

This thesis proposes a compilation framework wherein the compiler is provided with the capability to dynamically select the compilation unit and, based upon the characteristics of the compilation unit, apply the most appropriate transformations from the available repertoire. The remainder of this chapter discusses the ramifications of this concept to the design of an ILP compiler. Section 4.1 will discuss the environment of compiler selected compilation units, called *regions*. Section 4.2 will discuss the application of transformations in a region-based compilation framework. Section 4.3 will introduce the state maintenance issues exposed by this technique. Finally, Section 4.4 will discuss the transformation of a function-based compiler into a region-based compiler.

4.1 Compilation Environment

Consider an environment in which the compiler is allowed to repartition the program into a new set of compilation units, called *regions*. A region is a subgraph of the global control flow graph of the program. More formally, a program P can be represented by a global control flow graph $G(N, E)$, where N is the set of all basic blocks in the program and E is the set of all control flow arcs between two blocks in N. These arcs include interprocedural transitions through subroutine calls and returns.

Definition: A region R is represented by the control flow graph $R(n, e)$ where $n \subset N$ and $e \subset E$.

Note that under this definition, a function is a region, but a region is not necessarily a function. The subgraph making up a region within the program must either be disjoint from all other regions, contain a nested region or be nested within another region.

A region is simply a collection of basic blocks and control flow arcs selected to be compiled as a unit. These regions will be the fundamental units to which optimization, scheduling and register allocation will be applied. In this environment, control of the compilation scope is transferred from the software designer to the compiler. Under the assumption that the compiler has knowledge of the transformations available and understands their behavior, the compiler is better able to select desirable compilation units than the software designer.

The compiler transformations applied in a function-based environment tend to be restricted during implementation of the compiler or during compilation itself to prevent excessive code expansion that may adversely affect compilation time or compilation memory requirements. Allowing the compiler to repartition the program into regions allows the compiler to isolate compilation of program partitions from code expansion effects within the surrounding function body. Region-based compilation units provide an environment that reduces the importance of the time and memory complexity of compiler algorithms. This allows the compiler to apply more aggressive transformations and may make the compiler easier to design.

This remainder of this section will discuss several issues in region formation, present the region formation algorithm used for the remainder of this thesis and investigate the ability of regions to provide a more desirable compilation environment in the presence of aggressively inlined functions.

4.1.1 Region formation

The goal of the region formation process is to provide the best possible compilation unit for aggressive transformation. The properties of the selected region should be such that aggressive optimization is both feasible and beneficial. The properties that are important are heavily dependent upon the transformations to be applied by the compiler and the efficiency of the implementation. In general, these properties can be separated into two categories: those that affect compiler performance and those that affect output code quality.

Compiler performance

As was shown in Section 3.3.4 even a small amount of code expansion can have a significant impact on compiler performance. Compilation time and compilation memory usage depend upon the input size and the algorithmic complexity of the applied transformations as well as the efficiency of the implementation. Thus, the most important region property from the point of view of compiler performance is size. The performance of these transformations will be determined by the number of operations, the number of virtual registers, and the number of memory dependences within the region. Care must be taken when selecting upper bounds for these values. The applied transformations may require less time and memory but they may produce lower quality output code.

As an example, consider a global, graph-coloring register allocator [6]. The computational and memory complexity of this approach to register allocation is $\mathrm{O}(n^2)$, where n is the number of virtual registers in the region. Thus, the amount of time and memory required for global register allocation is heavily dependent upon the size of the region in terms of the number of virtual registers. If a function containing 1000 virtual registers is repartitioned into ten regions

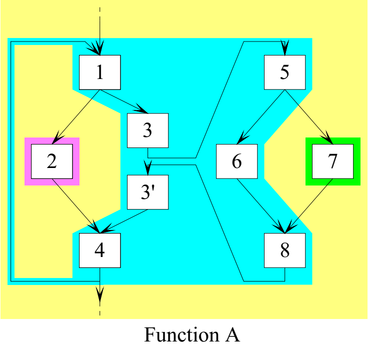


Figure 4.1 Profile-sensitive region selection.

containing 100 virtual registers, algorithmically the time and memory required to allocate all ten regions is ten times less than the time and memory required to allocate the entire function all at once. However, narrowing the scope of register allocation to this degree may result in a lower quality allocation. This will be discussed in more detail in Chapters 5 and 6.

Output code quality

The quality of the output code generated by the compiler depends upon the ability of the compiler to efficiently transform the code within the region. The region formation process should consider the presence of optimization hazards, the control flow structure, dependence height and dynamic program behavior when selecting regions. By excluding basic blocks that are dynamically irrelevant or contain operations that will inhibit optimization and scheduling, the region formation algorithm may produce a compilation unit that will allow the compiler to produce better quality code.

Figure 4.1 contains function **A** from Figure 3.5 after profile sensitive region formation. Recall that the most frequent path through the loop is composed of basic blocks **1, 3, 4, 5, 6, and 8**. Blocks **2** and **7** have been placed in regions separate from the most frequently executed portion of the loop. This allows the dynamically important portion of the loop body to be optimized without the constraints that may be imposed by the contents of blocks **2** and **7**. The region formation algorithm used in this thesis is profile sensitive for this reason. The benefits of optimization and register allocation over profile sensitive regions will be discussed in Chapters 5 and 6.

Profile-sensitive region formation

This region formation algorithm is a generalization of the profile-based trace selection algorithm used in the IMPACT compiler [46]. The principal difference is 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 algorithm consists of four steps. The first step is to select a starting block, s, known as the seed, to begin growing the region. The seed block selected is the most frequently executed block not yet in a region. The second step is to expand the scope of the region from the seed block by selecting a path of desirable successors from s. For this discussion, we assume that desirability is based solely upon execution frequency. In reality, the algorithm is constrained by an upper bound on region size. In this context, 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 Therefore, y is a desirable successor of x if it satisfies the following equation.

$$Succ(x, y) = \left(\frac{\mathcal{W}(x \rightarrow y)}{\mathcal{W}(x)} \geq T\right) \&\& \left(\frac{\mathcal{W}(y)}{\mathcal{W}(s)} \geq T_s\right) \tag{4.1}$$

The values T and T_s are threshold values defined by the compiler. Through empirical evaluation it was found that setting both T and T_s to 0.5 produces reasonable regions. At higher values, the regions tend be too large and with smaller values the algorithm is constrained and the regions are too small.

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. The third step is to add a path of desirable predecessors from s. The conditions under which block y is a desirable predecessor of a block x are analogous to the successor case. Thus, y is a desirable predecessor of x if it satisfies the following equation.

$$Pred(x, y) = \left(\frac{\mathcal{W}(y \rightarrow x)}{\mathcal{W}(y)} \geq T\right) \&\& \left(\frac{\mathcal{W}(y)}{\mathcal{W}(s)} \geq T_s\right) \tag{4.2}$$

The resulting path forms the seed path of the region. In the final step, the region is further extended by applying Equation (4.1) to all blocks in the region, selecting all possible desirable successors. 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 4.2.

Consider the application of this algorithm to the CFG shown in Figure 4.3. The numbers next to the arcs represent the execution frequency of that control flow arc. Region formation begins by selecting the most frequent block not yet in a region. The header of the inner loop,

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

/* select path of desirable successors */
$x = seed$
$y$ = most frequent successor of $x$
while ( $y \ni R$ && $Succ(x,y)$ ) {
      $R = R \cup \{y\}$
      $x = y$
      $y$ = most frequent successor of $x$
}

/* select path of desirable predecessors */
$x = seed$
$y$ = most frequent predecessor of $x$
while ( $y \ni R$ && $Pred(x,y)$ ) {
      $R = R \cup \{y\}$
      $x = y$
      $y$ = most frequent predecessor of $x$
}

/* select desirable successors of all blocks */
$stack = R$
while ( $stack \neq \emptyset$ ) {
      $x$ = Pop($stack$)
      for each successor of $x$, $y \ni R$ {
            if ( $Succ(x,y)$ ) {
                  $R = R \cup \{y\}$
                  Push($stack$,$y$)
            }
      }
}
```

Figure 4.2 Profile-sensitive region formation algorithm.

the highlighted block, is the most frequently executed block and is selected as the seed block. The most likely target block of the seed, block **2**, is selected as the desirable successor to block **1**. Block **4** is selected as the successor of block **2**. Block **4** contains the backedge of the inner loop, but since the inner loop only iterates two times on average according to the profile information, block **7** is selected as the successor to block **4**. If the inner loop tended to iterate more frequently, the successor path selection process would have terminated because the block

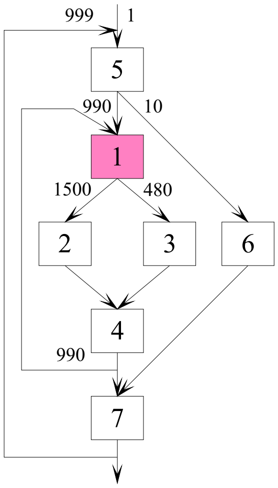


Figure 4.3 Profile-sensitive region formation: seed selection.

7 would not be a candidate and block **1** already lies within the region. Blocks **4**, **7** and **5** are then selected, in that order, as desirable successors. The converse is true for block **7**. In this instance the region will not grow outside the loop because the most frequent path is around the backedge to block **5**. At this point, successor path growth terminates, since block **1** is already in a region and the transition from block **5** to block **6** does not satisfy Equation (4.1). The result is shown in Figure 4.4. The third step of the region formation process is to select the most desirable predecessor of the seed block; in this example, block **5** already resides in the region so no desirable predecessors are selected. Finally the fourth step is to select desirable successors of all blocks within the region. The transition from block **1** to block **3** satisfies Equation (4.1) so block **3** is added to the region. As stated before, the transition from block **1** to block **6** is not desirable. The region formation process terminates having selected all basic blocks except block **6**. The result is shown in Figure 4.5. The region shown in Figure 4.5 is now complete and is representative of the dynamic behavior in this area of the program.

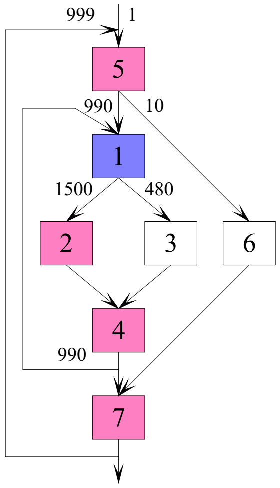


Figure 4.4 Profile-sensitive region formation: successor path selection.

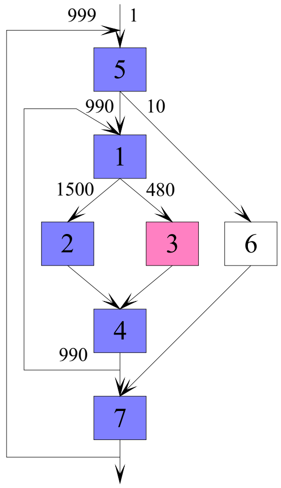


Figure 4.5 Profile-sensitive region formation: final step.

Natural regions

In the presence of profile information, the compiler has the advantage of knowing how a program behaves dynamically. The absence of profile information does not preclude region formation. The region formation process simply has to partition a function based upon the natural regions present within the CFG. The most obvious of such regions is a loop. In the previous example, the dynamic profile information guided region formation to grow the region in Figure 4.5 that contains both the inner loop and the outer loop. The same result may be achieved in the absence of profile simply by starting the region formation process at the innermost loop bodies expanding the region to outer loops until the upper bound on region size is reached.

Once all of the basic blocks within loops have been grouped into regions, there are several methods of dealing with the general acyclic code that remains. All adjacent basic blocks within acyclic areas of the function may simply be grouped into the same region or a more sophisticated approach may be applied. The acyclic code itself may be partitioned into natural regions such as hammocks, single-entry subgraphs as proposed by Mahadevan [36], or control equivalent regions as proposed by Ferrante *et al.* [47]. Control equivalent regions, however, tend to be too small a unit for efficient region-based optimization.

Hazard avoidance

Profile information provides the compiler the ability to exclude dynamically irrelevant blocks from a region. Again, the absence of profile information does not preclude this. Hank showed that hazard-free paths tend to be frequently executed and that heuristics designed to avoid basic blocks that contain hazards tend to approximate the benefits achieved with profile information [37]. Thus, by excluding basic blocks containing unsafe subroutine calls or ambiguous store operations from a region, the compiler can achieve similar results to a compiler using frequency sensitive regions. For example, if basic block 6 in Figure 4.5 contained a library call, then a heuristic that avoided subroutine calls combined with the above loop-based heuristic would form the exact same region, without the aid of profile information.

4.1.2 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 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 4.1.1 was applied to the aggressively inlined (100%) benchmarks to determine how well the dynamic behavior of the program naturally controlled the size of the compilation unit. An upper bound of 200 basic blocks was placed on the region size. Figure 4.6 adds a histogram of the selected regions to the function size histograms shown in Figure 3.11. For all 11 programs, 75% of the execution time was contained in regions with fewer than 250 operations. The large percentage (15%) of regions that contain less than ten 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. Overall, the region selection algorithm is successful in controlling the problem size even in the presence of aggressive inlining.

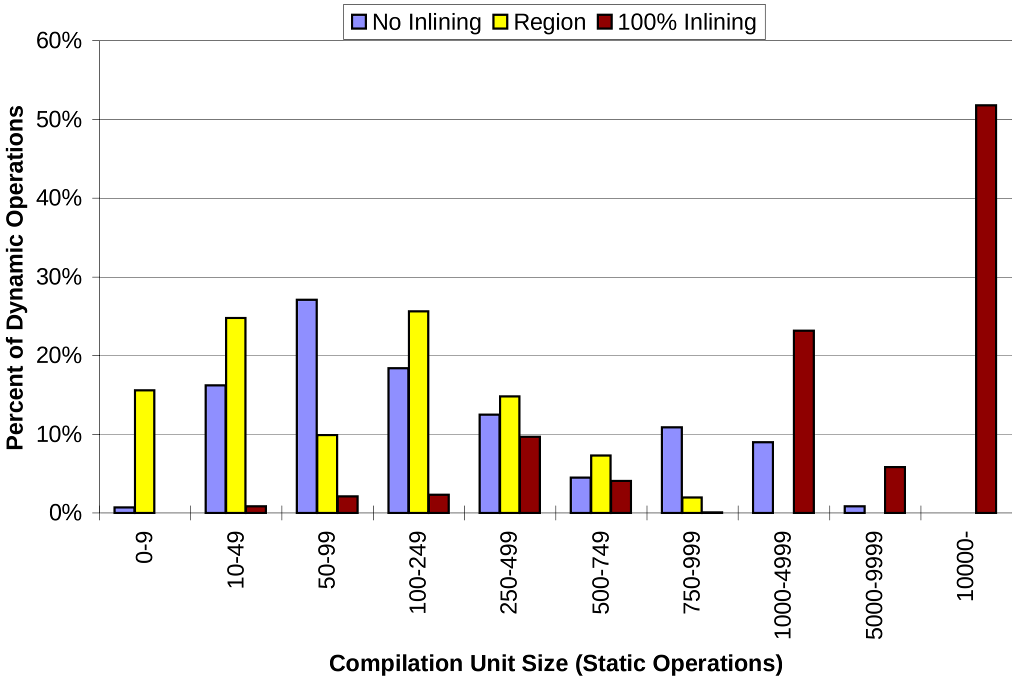


Figure 4.6 Histogram comparing function size before and after inlining with regions selected on inlined code.

Presenting the region size characteristics of all 11 programs within the same histogram hides some information. The comparison of region size and function size for the program **022.li** provides some interesting insights. In Figure 4.7, note 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 10000 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

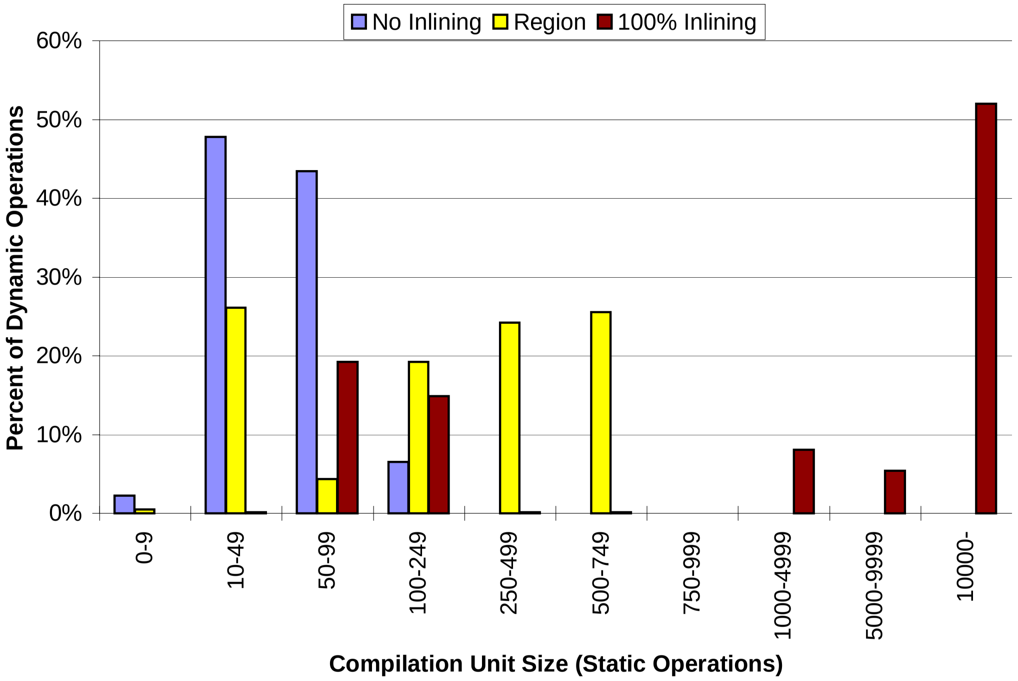


Figure 4.7 Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for 022.li.

into more desirable compilation units rather than the drastic increase seen at the function level. A similar result is seen for **tbl** as shown in Figure 4.8.

The region formation results for the remaining benchmarks shown in Figures 4.12 - 4.17 illustrate that the dynamic behavior of the benchmarks effectively limits the compilation scope. This is especially important for the benchmarks **072.sc**, **085.cc1**, and **134.perl**, which are shown in Figures 4.9 - 4.11, respectively. All three of these programs contain extremely large functions that will tend to dominate the compilation time required for the benchmark if an ILP compiler were to apply aggressive transformations to the entire function. The compile time and memory usage comparison of function-based optimization and region-based optimization in Chapter 5 substantiates this observation.

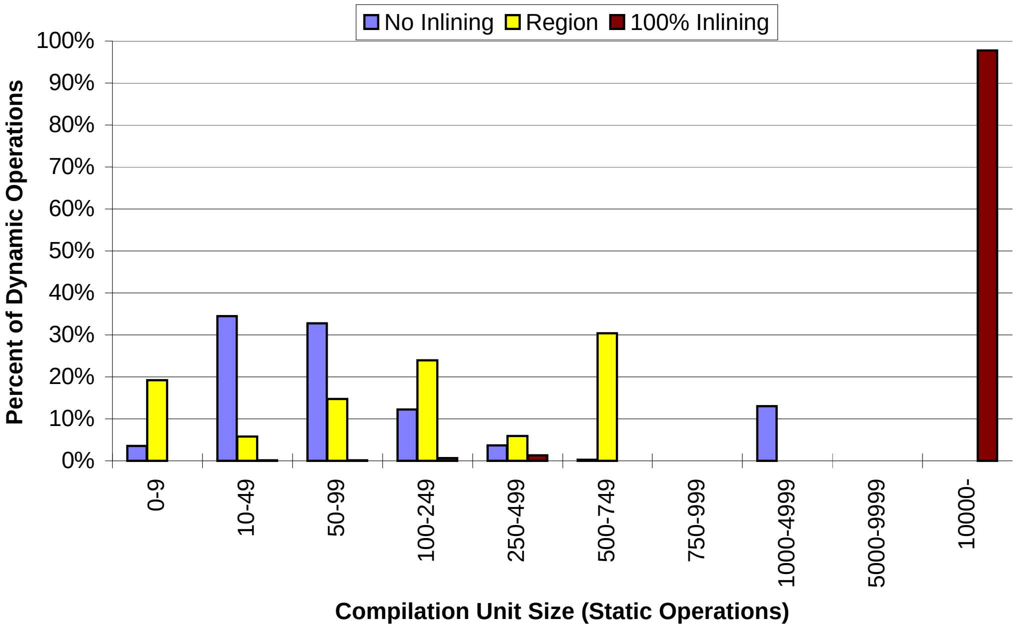


Figure 4.8 Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for tbl.

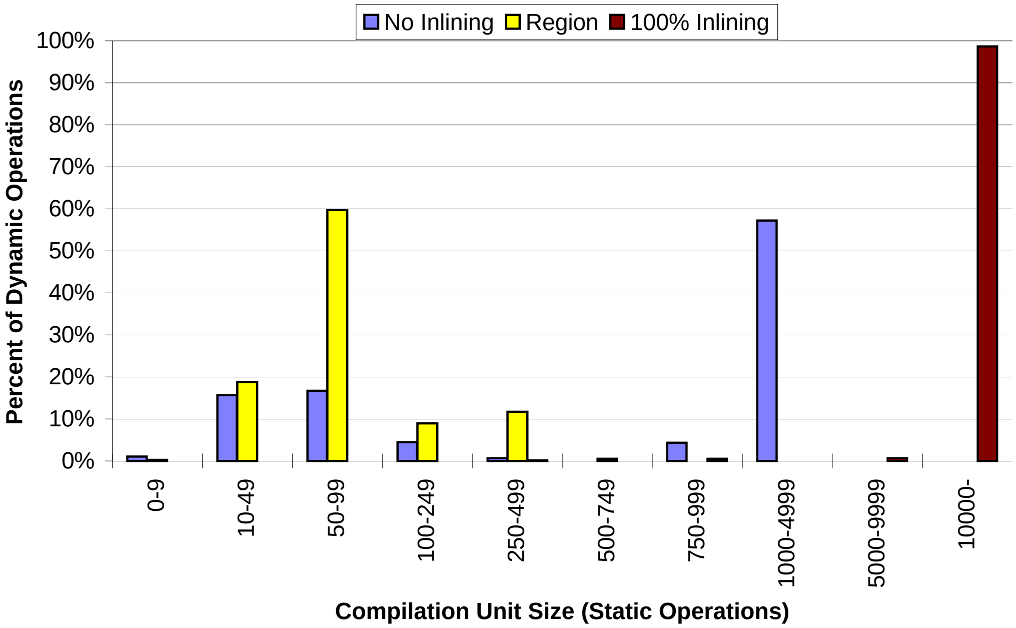


Figure 4.9 Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for 072.sc.

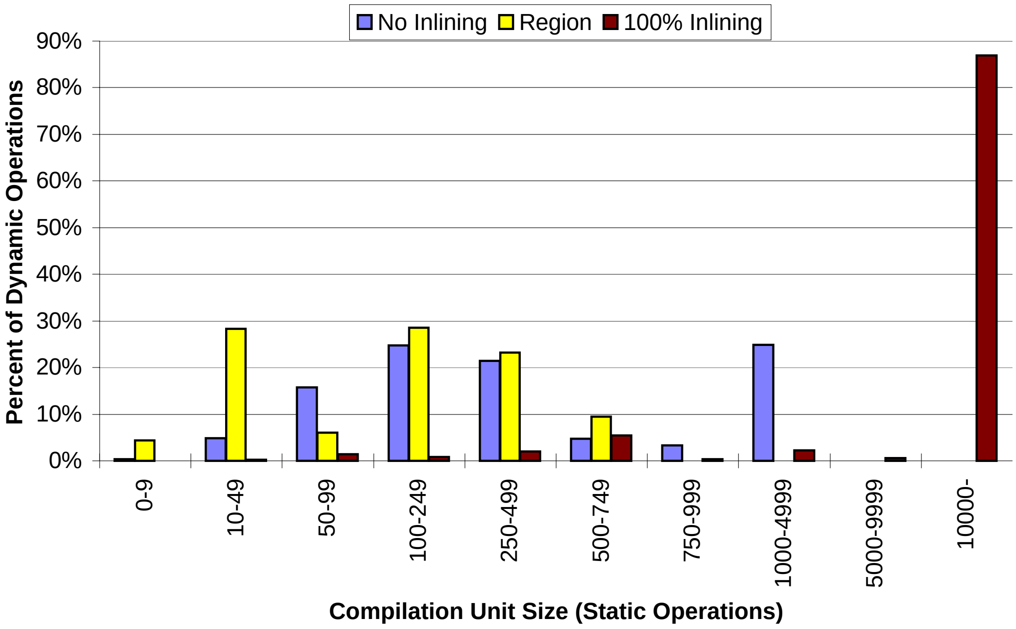


Figure 4.10 Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for 085.cc1.

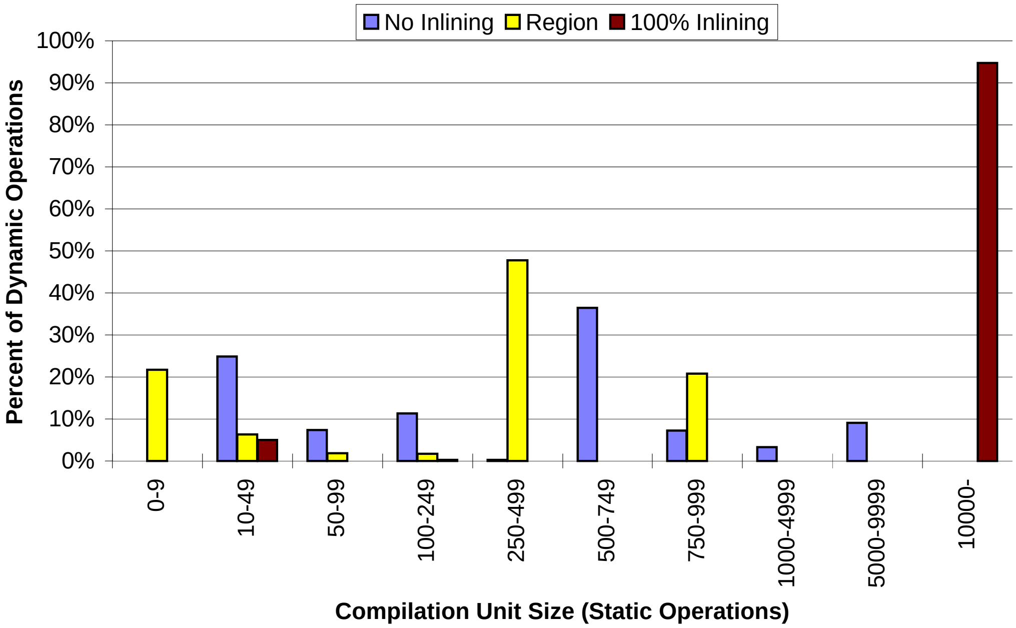


Figure 4.11 Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for 134.perl.

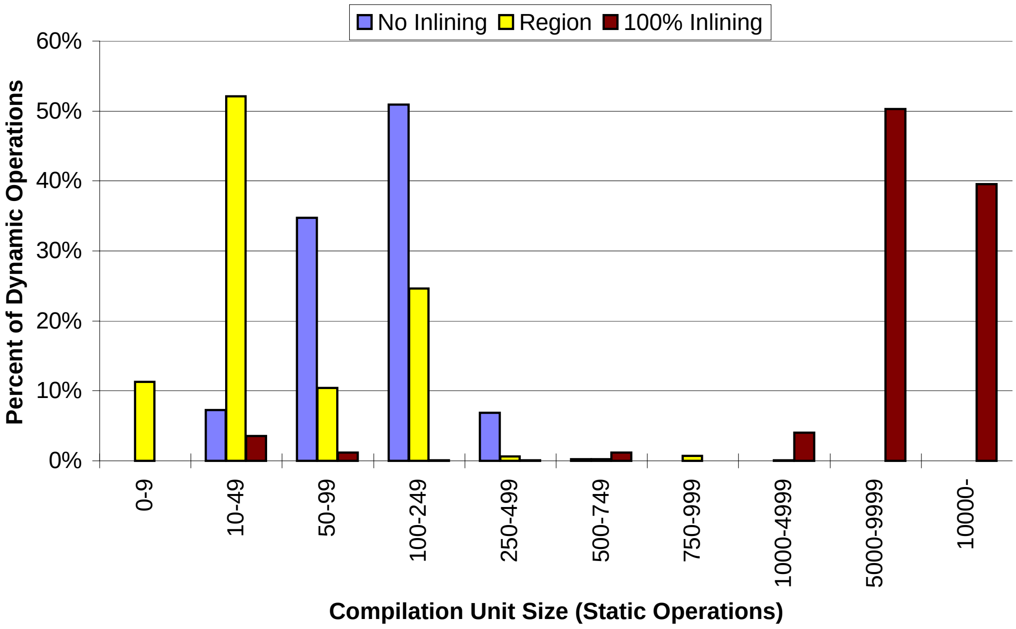


Figure 4.12 Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for 008.espresso.

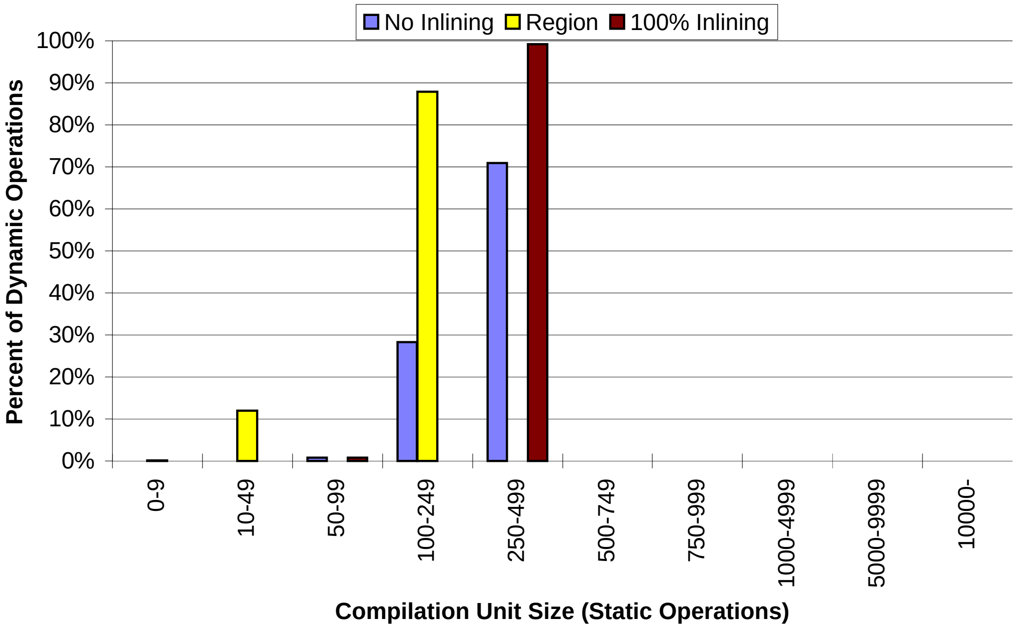


Figure 4.13 Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for 026.compress.

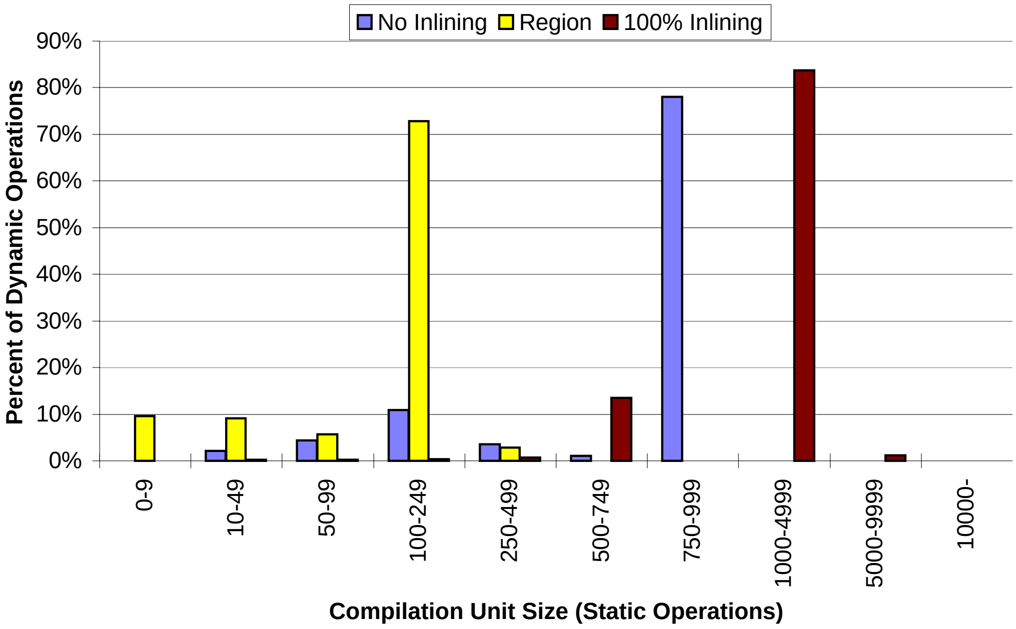


Figure 4.14 Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for cccp.

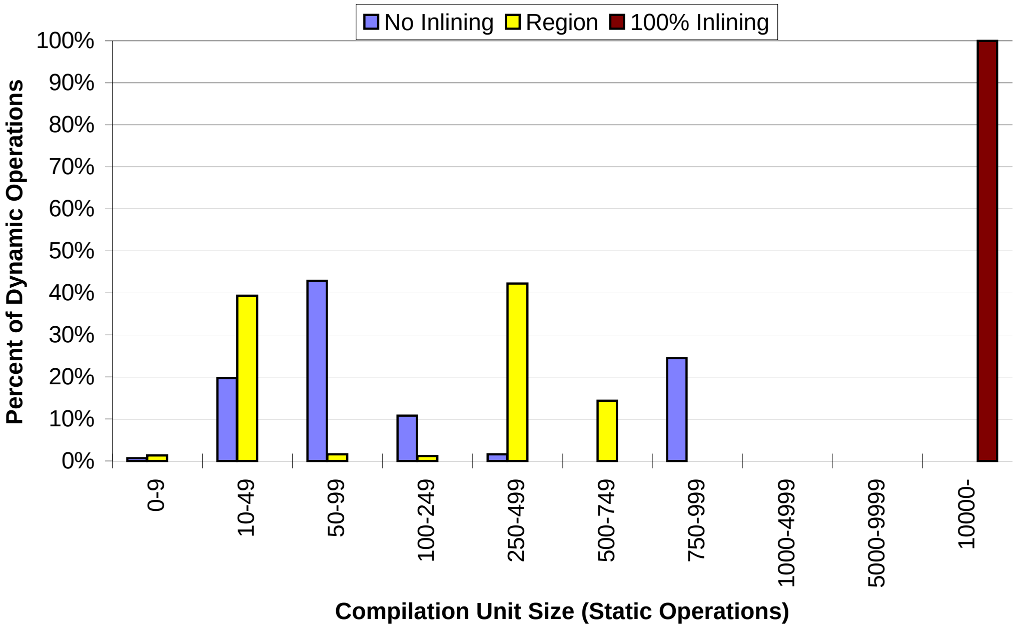


Figure 4.15 Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for eqn.

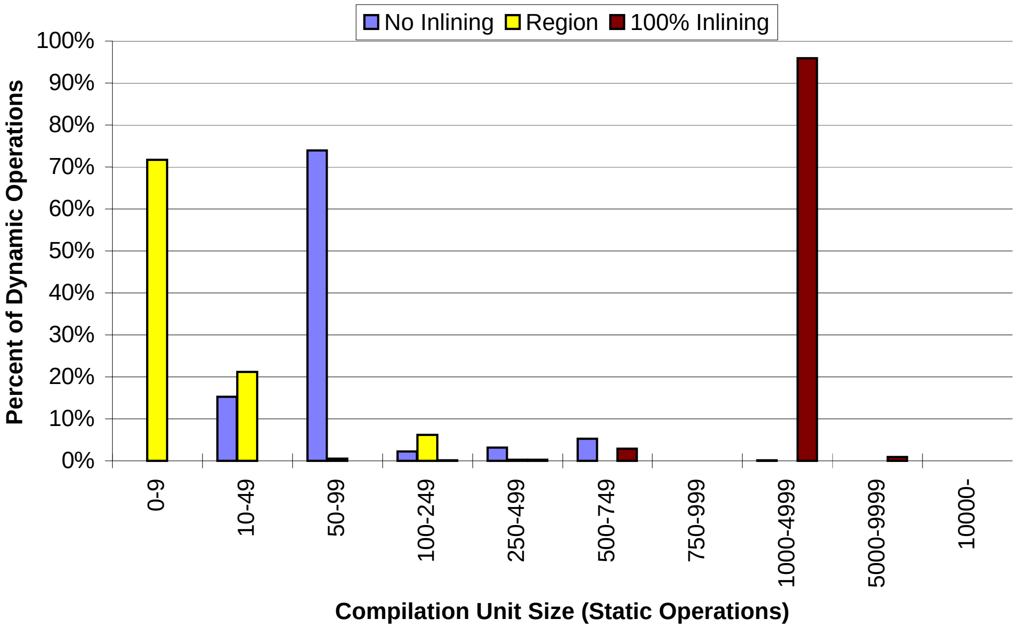


Figure 4.16 Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for lex.

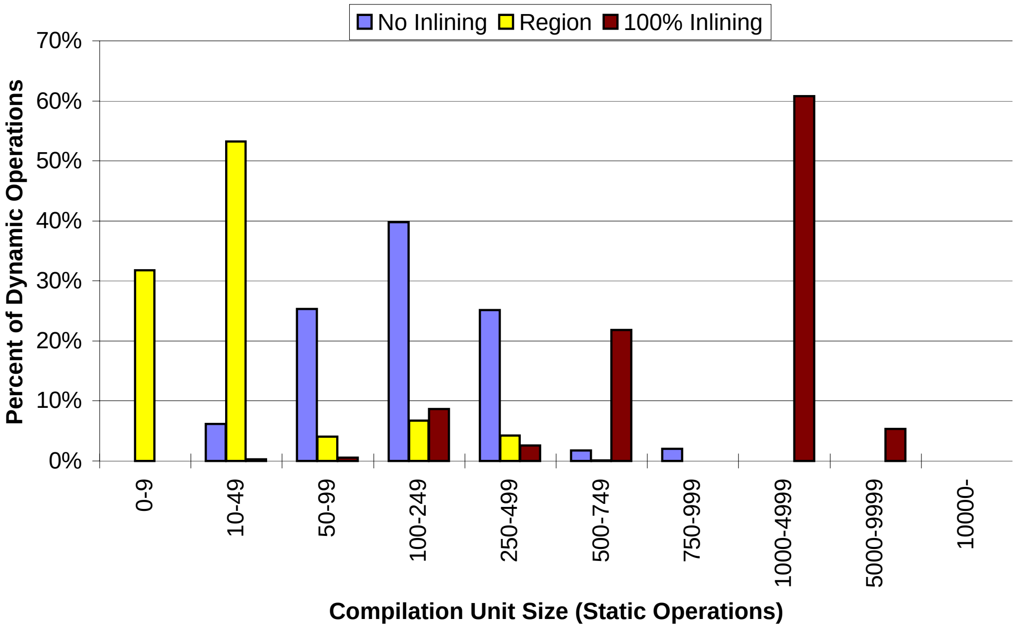


Figure 4.17 Histogram comparing function size before inlining, function size after inlining, and selected region size after inlining for yacc.

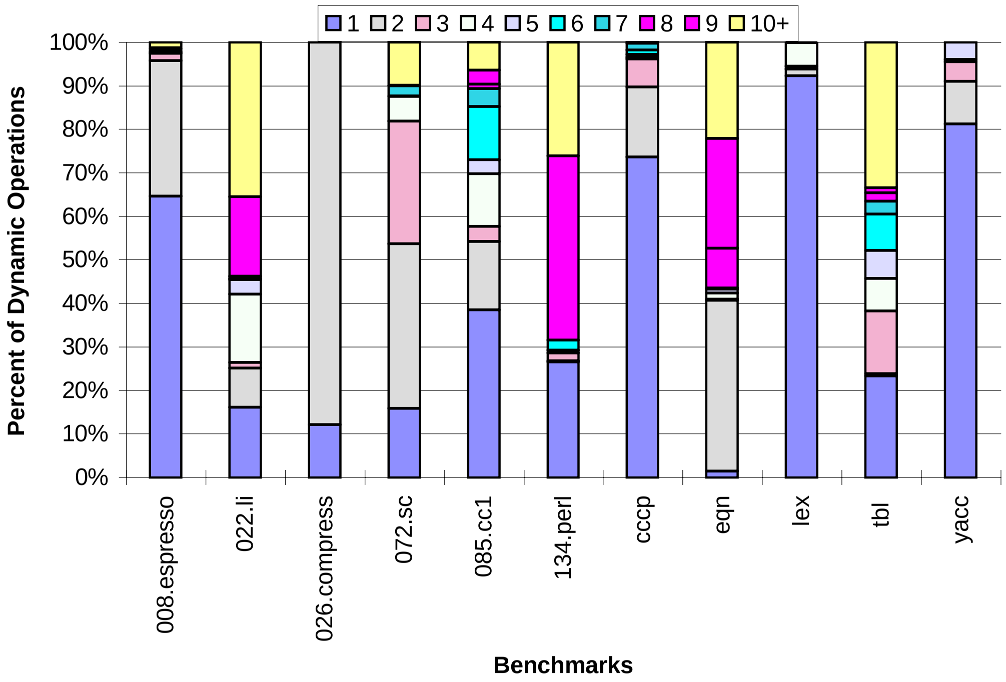


Figure 4.18 Dynamic distribution of the number of functions spanned by selected regions.

4.1.3 Regions spanning multiple functions

The intraprocedural distribution of cyclic vs. acyclic code in Figure 3.4 implies a significant amount of interprocedural coupling between program functions. Figure 3.10 shows that through aggressive inlining, most of the interprocedural transitions have been transformed into intraprocedural transitions. By examining the selected regions, we can determine the degree of interprocedural coupling, as indicated by the dynamic profile information. Figure 4.18 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 is counted. These regions are then weighted by dynamic execution frequency to produce the distribution.

As an example recall the inlining example in the benchmark **eqn** in Section 3.3.1. The region selected from the function *getline*, after the function *gtc* has been inline expanded, contains blocks from two functions. This region accounts for 30% of the dynamic operations within that program. Thus, Figure 4.18 shows that for **eqn** more than 30% of the dynamic operations are in regions spanning two functions. As selected, this region contains 32 of the 9600 operations in the function body that contains it.

The distributions in Figure 4.18 indicate the depth of the interprocedural coupling within these programs. Several of the benchmarks have a significant amount of interprocedural coupling. More than 40% of the dynamic operations in the benchmarks **eqn** and **022.li** are contained within regions spanning nine or more functions. For **022.li** most of these regions are small, containing fewer than 250 static operations. In the benchmark **085.cc1**, more than 70% of the dynamic operations are within regions spanning ten or more functions. Within **perl**, through inlining and region formation, a cyclic region is formed that spans nine functions and represents 25% of the dynamic operations in the program. The importance of this region shows that it is desirable to assemble interprocedurally 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.

4.2 Compilation Methodology

As discussed in Chapter 3, current compilers typically compile each function as a unit and each function is compiled completely before proceeding to the next function. Under a region-based environment, the compilation process can be generalized in several significant and beneficial ways.

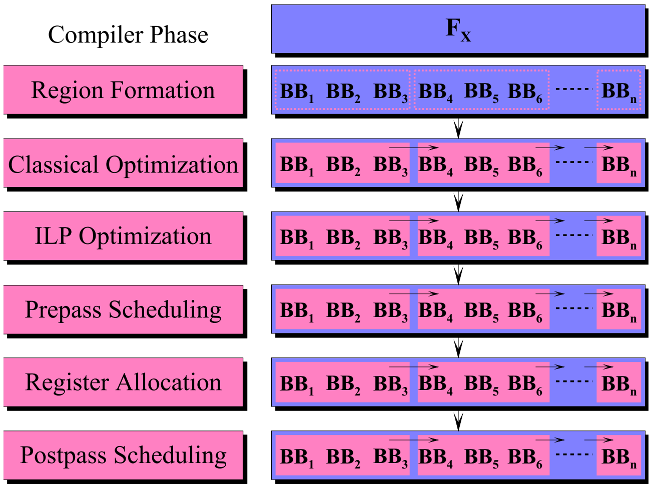


Figure 4.19 Horizontal region-based compilation of a function.

The region-based compilation process requires an additional phase of compilation, region-formation. The addition of this phase to the function-based compilation process shown in Figure 3.2 results in the compilation process shown in Figure 4.19. In this process, the function, $\mathbf{F}_x$, is first repartitioned into regions. Once repartitioned, each subsequent compilation phase processes the regions one at at time until the entire function body has been processed. Under this compilation model, restricting the compilation scope to individual regions within the function can result in significant improvements in the time and memory required to compile the function due to the nonlinear algorithms applied during each phase. Restricting the scope of the compiler in this manner will also affect the quality of the output code. Both of these issues will be discussed in detail later in this dissertation.

In addition, treating each region as a separate compilation unit allows the compilation process to be modified as shown in Figure 4.20. Under this model, a region is selected and all compilation phases are applied to that region, in the same manner that the phases are applied under the function-based model, before subsequent regions are selected. This adds a *vertical*

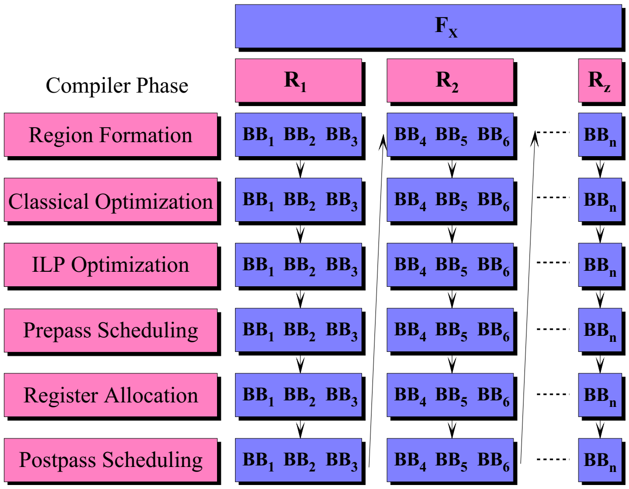


Figure 4.20 Vertical region-based compilation of a function.

component to the compilation of a function. Since a region has a scope that differs from the function it resides in, basic blocks in different regions may be in two completely different phases of the compilation process. [1]

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. Consider the application of global optimization, followed by scheduling, to a region. Compensation code generated while applying a transformation to a region may be pushed outside the transformed region. This compensation code may be selected into subsequent regions where it will be subject to optimizations applied when that region is processed. This is not the case under the function-based model, since the entire function is optimized prior to scheduling and reapplication of the optimizer after scheduling may destroy the schedule. For this reason, may code schedulers tend to avoid introducing compensation code during scheduling. The region-based compiler framework provides an environment where the benefits of removing

[1] The intermediate representation used by the compiler must be able to handle the fact that different portions of the function are in different phases of compilation

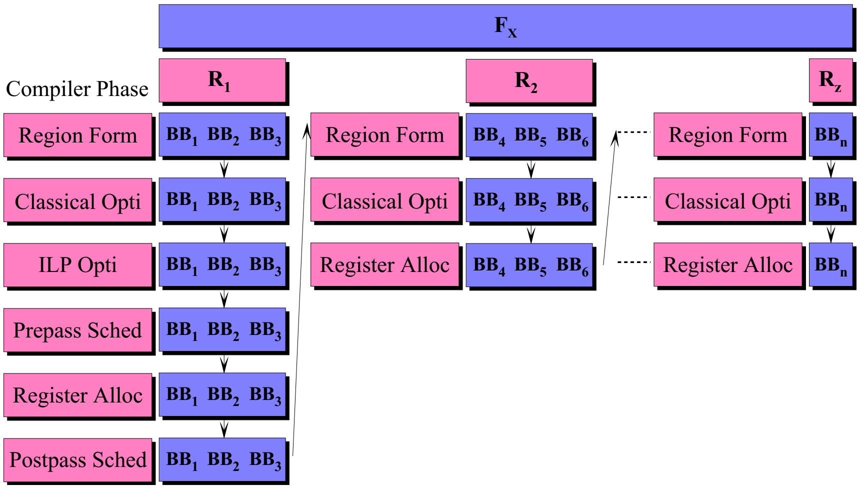


Figure 4.21 Specialized vertical region-based compilation of a function.

these restrictions from the code scheduler can be investigated in the future. These benefits will not be quantified within this thesis.

The region-based compilation process can be further refined as shown in Figure 4.21. Under this model, the compiler can determine the path a region follows through the compilation process based upon the characteristics of the region. In general, this is not feasible within a function-based compiler, since function-based compilation units are difficult to characterize. The region characteristics that may be relevant to the compiler include dynamic importance, topology, content, and previously applied transformations. For example, in Figure 4.21, region $\mathbf{R}_1$ is subjected to aggressive ILP optimization and scheduling, whereas region $\mathbf{R}_2$ is subject to only classical optimizations. In this instance the characteristics of region $\mathbf{R}_2$, such as dynamic importance, indicate to the compiler that aggressive ILP transformations are unnecessary. The compilation time benefits of guiding the compilation process in this manner will be investigated in Chapter 5.

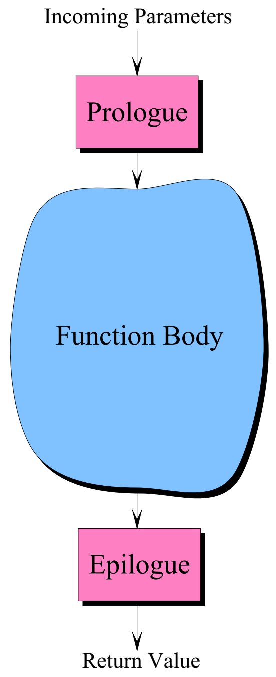


Figure 4.22 Function boundary conditions.

4.3 State Maintenance

Separate compilation of a program using a function-based framework is facilitated by the fact that the boundary conditions of a function are fixed. Consider the function body shown in Figure 4.22. The function has a single entry point, the prologue, and the variables live across this entry point, function parameters, and their locations are fixed by the parameter passing convention. Likewise at the exit point, the epilogue, and the location of the single live variable across that point, the return value, is fixed by the parameter passing convention. Because of this, each function can easily be treated as a self-contained entity. As long as the compiler obeys this convention, there is no need for the compiler to maintain any information between functions.

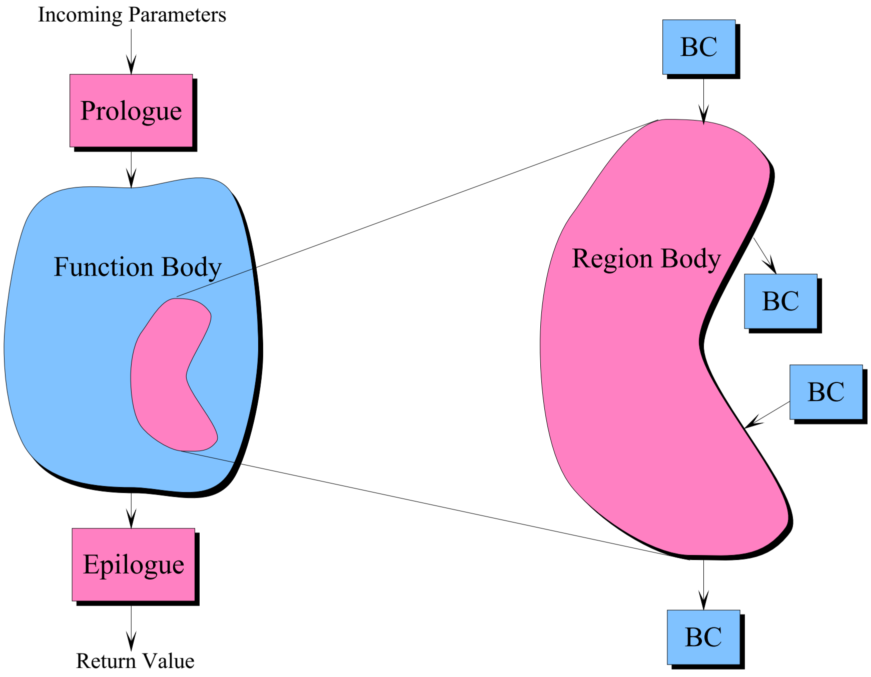


Figure 4.23 Region boundary conditions.

On the other hand, a region is an arbitrary partition of the program's control flow graph. There may be any number of variables live across each region's entry and exit points. Figure 4.23 shows a region that is a subgraph of the function in Figure 4.22. This region contains two entry points and two exit points. The boundary conditions at each of these points are represented by the blocks labeled **BC**. The variable liveness conditions at a region boundary point are dynamic and will change as transformations are applied. For correctness, the compiler has to maintain the variable liveness conditions at each entry and exit point throughout the compilation process. Chapter 5 will discuss methods of maintaining this information and the effect of conservative information on optimization quality.

In addition to variable liveness, 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 was done within the Multiflow compiler in the context of a combined scheduler and register allocator, which was applied to individual traces. For correctness, the scheduler must take into account information on processor resources and register bindings at trace boundary points [1]. Chapter 6 will discuss the boundary condition information required for efficient register allocation of arbitrary regions.

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.

4.4 Region-Based Compiler

The functionality of a region-based compiler can be decoupled into two broad categories. The first category is compilation management. This category includes the three compilation tasks discussed in Sections 4.1 - 4.3. The second category consists of all transformations in the compiler's repertoire. Decoupling the available transformations from the management of the compilation process allows existing function-based compilation techniques to be applied to region-based compilation units in a straightforward manner.

Figure 4.24 contains a block diagram of a region-based compiler. The compilation manager examines the input program and is responsible for region formation, transformation selection, and state maintenance. The compilation manager sends regions to the appropriate transformations and receives the result through some interface, which is implementation dependent. In order to implement a region-based compiler within the IMPACT compilation framework, the general region-based compilation process is modified slightly from that shown in Figure 4.21. After region formation, a new step, encapsulation, is performed prior to applying any compiler transformations. The purpose of this step is to encapsulate the region such that it appears to

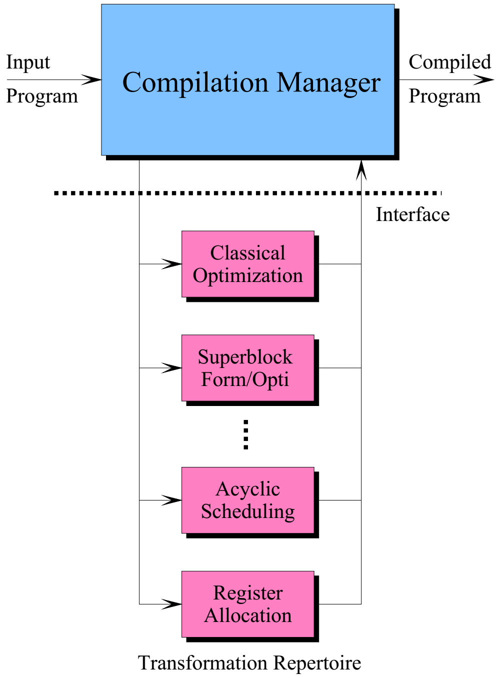


Figure 4.24 Region-based compiler block diagram.

be a function. This encapsulation allows existing function-based transformations within the IMPACT compiler to be applied directly to regions with only minor modifications and facilitates the comparison of region-based and function-based compilation techniques. After the transformation of a region is complete, the region proceeds through a re-integration step. The remainder of this section will discuss encapsulation and re-integration in more detail.

4.4.1 Encapsulation

Representing a region so that it appears to be a function requires the encapsulation to resolve two issues. The first concerns control flow entering and leaving the region. The second concerns variable liveness conditions at the region entry and exit points. Both of these issues can be resolved in the same manner. Consider the outer-loop region selected within the control flow graph in Figure 4.25(a). After extracting the region from the function it resides in, explicit

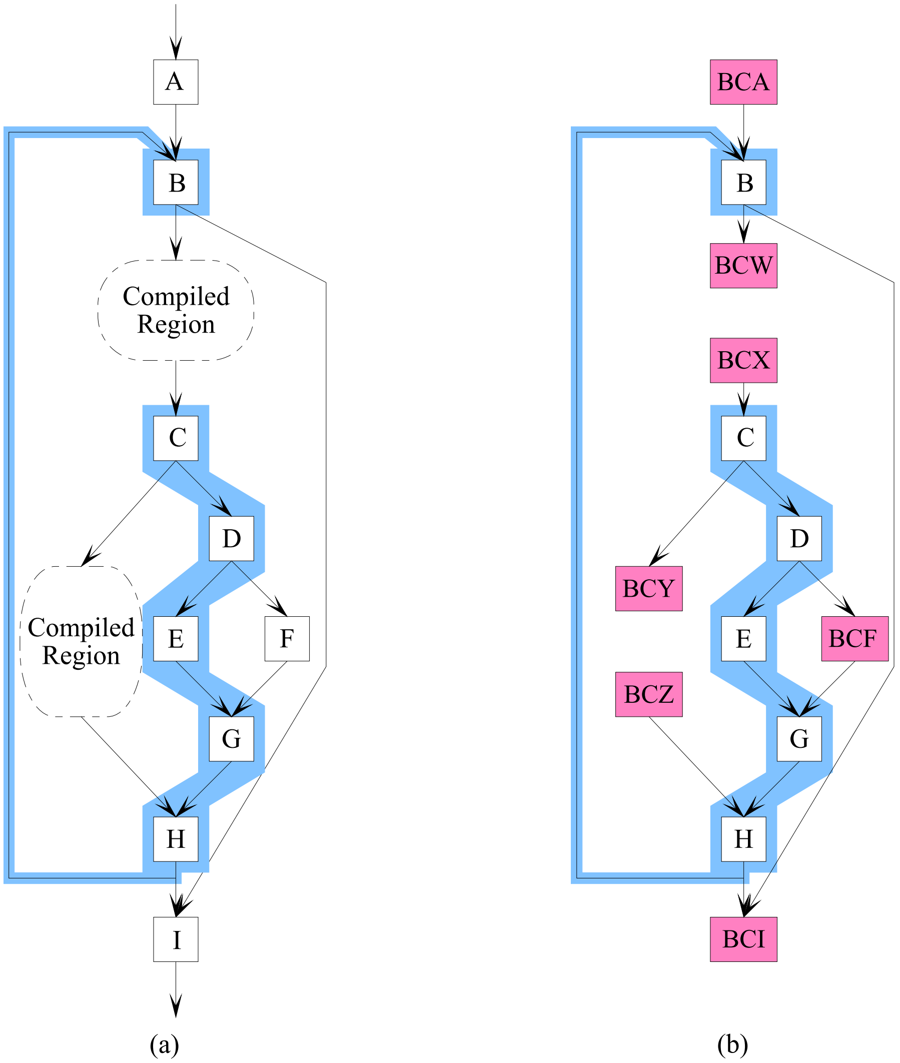


Figure 4.25 (a) Selected outer loop region, (b) extracted region with boundary condition blocks.

boundary condition blocks are added at each region entry point and exit point. These boundary condition blocks represent the actual blocks within the containing function that branch into the region or are reached when the flow of control exits the region. The boundary condition blocks required for the region selected in Figure 4.25(a) are shown in Figure 4.25(b).

It is desirable to compile selected regions in the same manner as functions so that existing function-based compilation techniques are applicable within a region-based compiler. To achieve

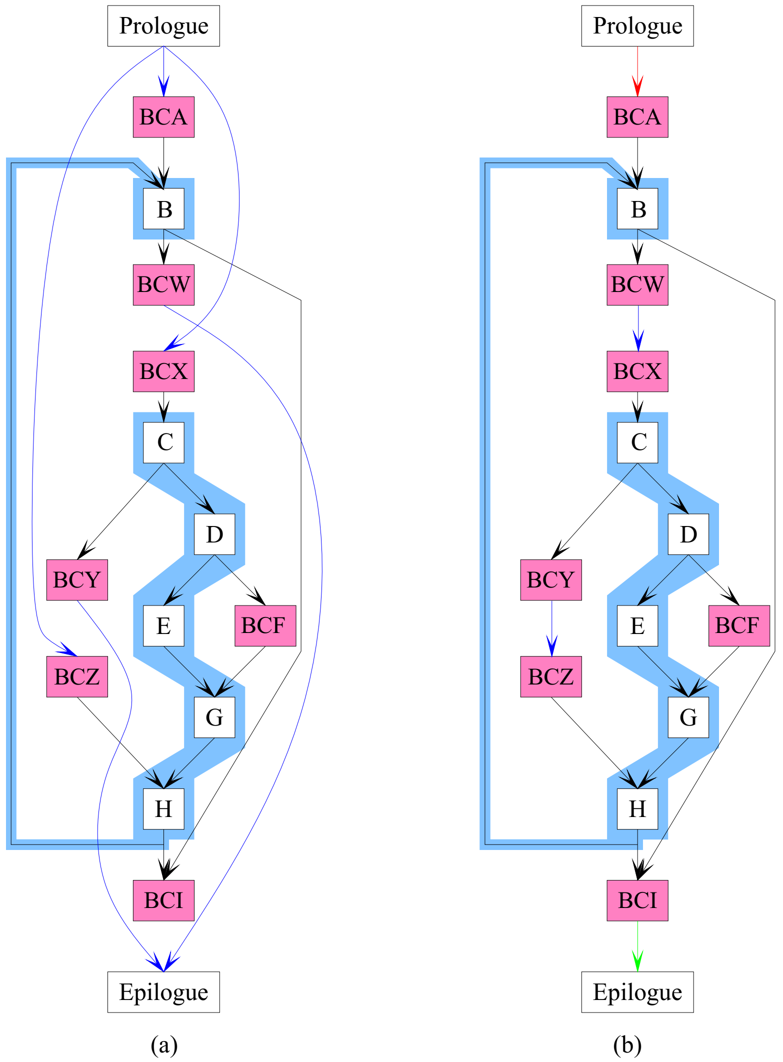


Figure 4.26 Control flow encapsulation equivalent to a function: (a) cyclic control flow hidden, (b) cyclic control flow exposed.

this, the compilation manager must generate control flow that is equivalent to a function at the region boundary points, i.e., the region should appear as a single-entry/single-exit control flow graph. This can be done by adding adding dummy prologue and epilogue blocks and connecting the prologue to all region entry points and connecting all region exit points to the epilogue as shown in Figure 4.26(a). The control flow of the region is equivalent to that of a function, however, note that the cyclic nature of the region is lost with this encapsulation. Within the home function the region was cyclic, as encapsulated in Figure 4.26(a) it is not. Regions must be encapsulated in such a way that the original control flow structure is apparent to applied transformations. With this metod of encapsulation, outer loop regions cannot be optimized as loops.

The encapsulation process may take advantage of the control flow relationships between entry and exit boundary condition blocks. For example, if an exit boundary condition block dominates an entry boundary condition block and those blocks lie within the same region, a control flow arc may be added between those blocks. Consider blocks **BCW** and **BCX** in Figure 4.26. If the exit block **BCW** dominates **BCX**, a control flow arc may be added between them as shown eliminating the need for a control flow arc between the prologue and **BCX**. The same may be true for blocks **BCY** and **BCZ**. Any remaining exit blocks that do not dominate an entry block are connected to the epilogue, such as block **BCI** in Figure 4.26(b). Finally, any entry blocks reachable from outside the region are connected to the prologue, such as block **BCA**. In this instance, the control flow is again equivalent to that of a function, but the original cyclic nature of the region is apparent. Applying these control flow relationships allows outer-loop regions to benefit from cyclic transformation.

Conveying that a particular set of variables is live across a region exit boundary can be done by adding dummy operations to the boundary condition block that explicitly reference those variables. For example, Figure 4.27(a) shows a region exit where virtual registers 1, 2, and 3 are live-out. This fact can be conveyed by adding three dummy operations to a boundary condition block that reference these three virtual registers as shown in Figure 4.27(b). Conveying variable liveness across a region entry point is trivial. Live variable analysis works backward from the use of a virtual register until it reaches a definition [5]. Thus, a virtual register is live into a

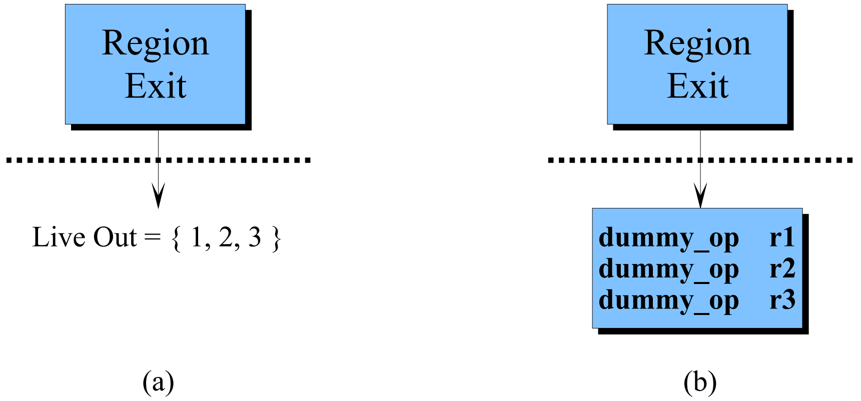


Figure 4.27 Representing live-out conditions at a region exit: (a) virtual registers live-out, (b) represented by dummy operations.

region if a boundary condition block is reached prior to locating a definition of that virtual register. Live variable information may also be expressed to a transformation using some other data structure that is associated with the region to be processed. However, expressing live variable information in this manner allows the existing IMPACT dataflow analysis routines to operate upon regions without modification.

4.4.2 Re-integration

Re-integration is simply the process of placing the compiled region back into the containing function. The region-based compiler within IMPACT also performs all region state maintenance procedures at this time. As a result of transformation, the region entry and exit point may change, the dataflow conditions, specifically live variable information may change, and the register allocation constraints have to be updated. Chapters 5 and 6 will discuss how optimization will affect the live variable information at the boundaries and what constraint information is provided for register allocation, respectively.

In addition to representing control flow and hold live variable information, the boundary condition blocks serve as place holders for any compensation code that is generated during the compilation of the region. When transformation of a region is complete, any code residing within the boundary condition blocks is placed in a new block in the function. If the block that the boundary condition block represents already resides within a processed region, the new block becomes part of the region being re-integrated. However, if the block that the boundary condition block represents does not lie in a region, the new block participates in the region formation process to be grouped with subsequent regions. In this way, compensation code resulting from compilation of one region may benefit from later optimization when it is selected into a subsequent region.

CHAPTER 5

REGION-BASED OPTIMIZATION

The goal of code optimization is to improve the efficiency of the code produced by the compiler. In a classical optimizer, the goal is to improve efficiency by reducing the number of dynamic operations as much as possible. Within an ILP compiler, optimizations further improve the classically optimized code by performing dependence breaking transformations to expose more parallelism and increase execution efficiency on wide-issue processors. In both instances, the compiler designer must trade off the use of aggressive optimizations and, therefore, output code efficiency for the sake of containing compilation time and memory usage. This tradeoff is made more difficult by the need to increase the scope of optimization to maximize available optimization opportunities. The application scope of classical optimizations has expanded beyond the basic block boundaries to encompass the entire function for this very reason [48]-[51]. However, for the reasons discussed in Chapter 3, the function body is not the most suitable optimization environment.

In the literature, global optimization has only been applied at these two extremes, the basic block level or the function level. Classical optimizations have been applied in both a superblock [19] and hyperblock [20] context. After superblock and hyperblock formation, local classical optimizations are applied along with ILP transformation, but no optimizations are attempted between blocks. There has been no previous work in reducing the scope of global optimization to an arbitrary region. Although there has been work done to improve the efficiency of function-based global optimization, for example, the program dependence graph proposed by Ferrante *et al.* [47].

The goal of region-based optimization is to restrict the scope of the application of classical optimizations in such a way that compilation time and memory requirements improve and the code quality is not overly affected. Optimization time benefits result from the reduction in problem size and by focusing aggressive optimization on important regions and performing minimal optimization on less important regions. The compiler designer may use the time gained to apply more aggressive ILP transformations to important regions, further improving the quality of the code generated by the compiler. In addition to compilation time benefits, the region partition allows the classical optimizer to take advantage of optimization opportunities that typically require more complex transformations.

Classical optimizations reduce operation count by removing redundant computation. In order for an operation to be a candidate for application of an optimization it must be fully redundant, i.e., the re-computation is unnecessary along all possible paths between the two operations. Consider the example in Figure 5.1(a). Blocks **A** and **B** both contain an operation that performs the same computation, $x + y$. If there is no redefinition of variables x and y along paths **1** and **2** between these two operations, the operation in **B** is fully redundant and can be eliminated. If there is a redefinition of y along path **2** as shown in Figure 5.1(b), the the computation is redundant only along path **1**. In this instance, the re-computation is partially redundant and cannot be eliminated with traditional classical optimizations. The operation may be eliminated from block **B** using partial redundancy elimination techniques [52]-[55], but these techniques tend to be computationally expensive.

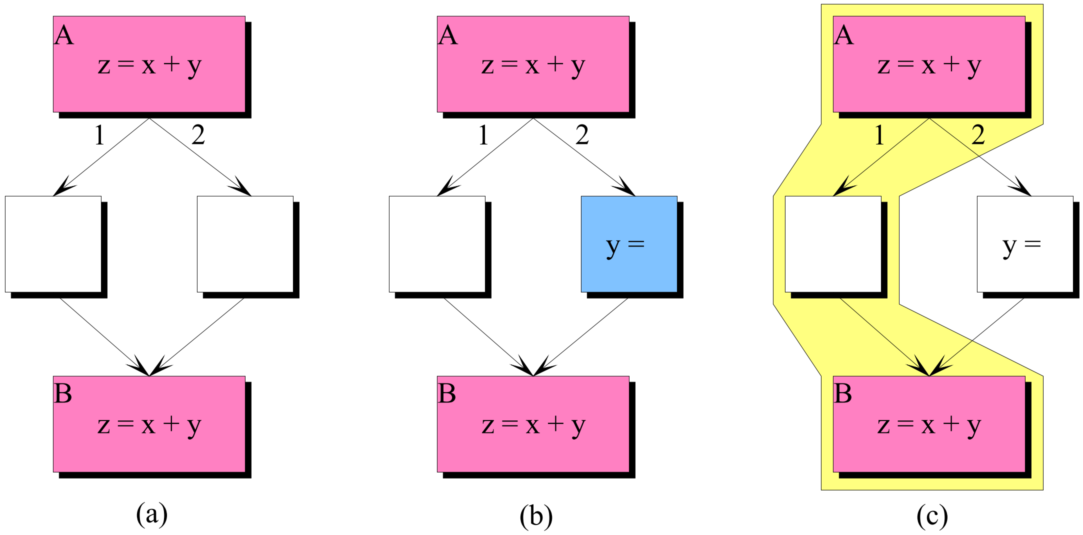


Figure 5.1 Redundant computation: (a) fully redundant, (b) partially redundant, (c) fully redundant within selected region.

Under a region-based environment, this optimization opportunity may be available to a classical optimizer. If path **2** is infrequently executed, region formation may exclude it as shown in Figure 5.1(c). This provides the classical optimizer with a straightforward method of ignoring path **2** and since it is not in the region, the optimizer has no visibility to it. Within the region, the operations in blocks **A** and **B** are fully redundant. This chapter will investigate the issues involved in performing separate classical optimization of regions and allowing a classical optimizer to take advantage of optimization opportunities exposed by the region partition, such as the one in Figure 5.1(c).

The IMPACT classical will be used as a test bed to investigate the effects of region-based optimization in terms of output code quality and optimization time and memory usage. The effect of the region partition has the most significant impact on the behavior global, interbasic block optimizations. As such, Section 5.1 presents a more detailed overview of the IMPACT classical optimizer. The boundary condition information required for effective optimization of regions is discussed in Section 5.2. In Section 5.3 the effect of region partitioning on selected global and loop-based classical optimizations is assessed. Finally, Section 5.4.2 investigates the impact of the region partitioning on classical optimization time and memory usage.

5.1 IMPACT Classical Optimizer

The IMPACT classical optimizer [17] will serve as a vehicle to investigate the effects of region-based compilation. The encapsulation method discussed in Chapter 4 will allow the use of the same optimizer for both the region-based experiments and the function-based experiments. The benefits of this are twofold. First, the IMPACT optimizer is known to generate good quality output code [17], [56]. Using the same optimizer allows region-based optimization to benefit from the quality of those transformations. Second, the compile time and memory usage benefits of region-based optimization are directly comparable.

The optimization process can be broken into two components. The first is the search for optimization opportunities and performing the transformation. The second is the generation of the dataflow information required to support the transformation. The optimization process within the IMPACT optimizer is iterative. Local, global and loop-based optimizations are repeatedly applied until convergence, i.e., there are no more optimization opportunities available. Tables 5.1 - 5.3 contain the local, global and loop transformations, respectively, that are applied by the optimizer. The algorithmic time complexity of these transformations is $O(n^2)$, where n is the number of operations in the scope of application of the transformation. For local transformations the scope is the basic block. For loop and global transformations, that scope is a loop or the entire function, respectively. Constraining the scope of these transformations has significant compile time benefits as shown in Section 5.4.2. The transformations annotated with

Table 5.1 Local optimizations.

constant propagation
forward copy propagation
memory copy propagation
common subexpression elimination
redundant load elimination
constant folding
strength reduction
constant combining
arithmetic operation folding
branch operation folding
operation cancellation
dead-code removal
code reordering

Table 5.2 Global optimizations.

constant propagation
forward copy propagation*
backward copy propagation
memory copy propagation
common subexpression elimination*
redundant load elimination
redundant store elimination
dead-code elimination*

an * in Tables 5.2 and 5.3, global common subexpression elimination, global copy propagation, global dead code elimination, loop invariant code motion, and global variable migration, will be used to investigate the effects of restricting optimization scope to a region.

Optimizers spend a great deal of time performing dataflow analysis. Dataflow information is summary information regarding the previous and future uses of variables at a particular point in a function. This information includes live-variable, reaching-definition, available-expression and available-definition information. There are a number of methods for collecting dataflow

Table 5.3 Loop optimizations.

invariant code motion*
global variable migration*
branch simplification
induction variable strength reduction
induction variable elimination

information [5], [57]-[59]. The most common and straightforward of these techniques is utilized within the IMPACT optimizer, iterative dataflow analysis.

For each of the previously mentioned types of dataflow information, iterative dataflow analysis requires the solution of a pair of recurrence equations for each point in the region being transformed. Using live-variable analysis as an example, we can derive the algorithmic time and memory complexity of this approach to dataflow analysis. The purpose of live-variable analysis is to determine for a variable v and a point p whether the value of v at p could be used sometime later in the control flow graph from point p. If so, then v is live at p; otherwise, v is dead. Assuming that we have determined the variables used by each block, $use[B]$, and the variables defined by each block, $def[B]$, the variables live-in and live-out of a basic block can be found by solving the recurrence Equations (5.1) and (5.2).

$$in[B] = use[B] \cup (out[B] - def[B]) \tag{5.1}$$

$$out[B] = \bigcup in[S_i],\ S_i\ is\ a\ successor\ of\ B \tag{5.2}$$

For a flow graph of n basic blocks, iterative live-variable analysis requires the solution of $2n$ equations. This dataflow analysis approach essentially iterates over the control flow graph until convergence is reached, i.e., a simultaneous solution to all $2n$ equations is found. The number of iterations required to converge to a solution varies with the control flow graph; however, the upper bound on the number of iterations is the number of blocks in the control flow graph. Thus to reach a solution requires n iterations over $2n$ equations, giving live-variable analysis complexity of $O(n^2)$. Similarly, the memory complexity of live-variable analysis is also $O(n^2)$. This result is also true of available-expression and available-definition analysis.

During the optimization process, the dataflow information eventually becomes incorrect or too conservative as a result of applying transformations. For this reason, dataflow analysis must be performed repeatedly throughout optimization. Constraining dataflow analysis to a region can significantly improve the amount of time the optimizer spends performing dataflow analysis simply by decreasing the number of basic blocks that must be considered simultaneously. However, this reduction in analysis scope will have some effect on optimization quality.

5.2 Boundary Conditions

Recall from the discussion in Section 4.3 that the separate optimization of functions is facilitated by known and fixed boundary conditions. When performing separate optimization of regions, the situation is different. The optimizer has no access to dataflow information outside the region such as variable usage patterns and previously computed subexpressions, thus, is unable to compute the dataflow boundary conditions at region entry and exit points. Consider the cyclic region consisting of basic blocks **A**, **B**, and **C** in Figure 5.2. Because of the region partition, the optimizer cannot determine the variable liveness conditions at the exit points from blocks **A** and **C**. In addition the compiler has no knowledge of the availability of variables or subexpressions at the entry points in blocks **A** and **C**. This dataflow information must be provided to the optimizer by the compilation manager to support effective optimization.

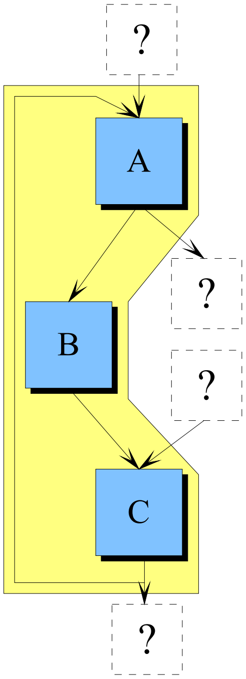


Figure 5.2 Region with unknown boundary conditions.

In addition, the region partition prevents the optimizer from knowing the contents of the potentially re-entrant control flow path exiting block **A** and entering block **C**. The optimizer may need to take these constraints into consideration while transforming the region. The remainder of this section will discuss the information provided to the optimizer by the compilation manager, Section 5.2.1, and how the optimizer can deal with constraints from side entrances, Section 5.2.2.

5.2.1 Dataflow

The compilation manager has the responsibility of providing and maintaining boundary dataflow information for each region within the program. The quality of the information provided depends upon the amount of work put into its maintenance. One purpose of region-based optimization is to minimize the amount of global analysis that must be performed on large function bodies that result from inlining and aggressive ILP compilation techniques. It is desirable that the compilation manager provide dataflow information at a level that is sufficient for efficient optimization without resorting to repeated global analysis.

Live-variable information

At the very minimum, the compilation manager must provide live variable information in some form. Live variable information is required for many global and loop optimizations, without it, incorrect transformations will result. One method the compilation manager may use to provide live variable information is to assume that all variables defined with a region are live outside the region. Under this assumption, the compilation manager need not perform any global live-variable analysis and there is no maintenance involved. However, the extremely conservative nature of this information may hinder optimization. The effect of such conservative live-variable information on global dead-code elimination will be investigated in Section 5.3.3.

Most likely it is desirable to provide live-variable information that is more accurate. This requires at least one global pass of live-variable analysis to determine the initial liveness conditions at basic block boundaries. The degree of difficulty of maintaining live-variable information can be understood by examining how transformations within a region can affect the live-in and live-out conditions at region boundary points. Underlying the subsequent discussion are two assumptions. First, there are no interregion transformations taking place. The reason for this assumption will become clear shortly. The second assumption is that code is allowed to be pushed outside a region as long as the target block does not reside within a compiled region. If this assumption were not made, then there are several more cases to consider; however, the end result is the same.

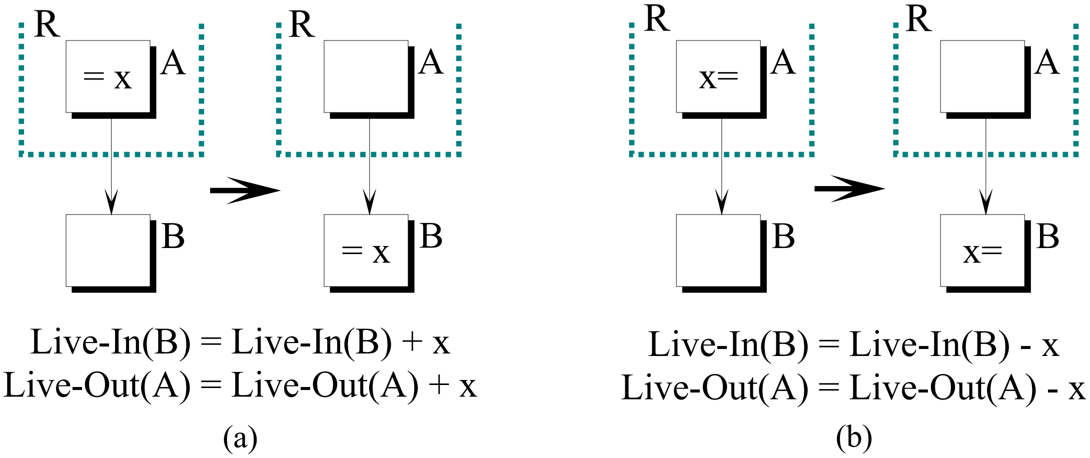

R
= x
A
B
R
A
= x
B
Live-In(B) = Live-In(B) + x
Live-Out(A) = Live-Out(A) + x
(a)
R
x=
A
B
R
A
x=
B
Live-In(B) = Live-In(B) - x
Live-Out(A) = Live-Out(A) - x
(b)

Figure 5.3 Local maintenance of live-out information.

First, consider the live-out conditions at the exit point of a region. Figure 5.3(a) contains a block **A** that resides within region **R** and a block **B** that does not yet reside within a region. If a variable x exists that is **not** live-out of block **A**, the only way that x can become live-out of block **A** is if an operation referencing x was to be pushed outside the region into block **B**, extending the lifetime, as shown in Figure 5.3(a). In this instance, the only live-variable maintenance that has to be done by the compilation manager is to add x to the live-out set of **A** and the live-in set of **B**. Now, consider the case in which the variable x is live-outside block **A** as shown in Figure 5.3(b). Since x is live-out, this implies that there is an operation outside of region **R** that uses x. Thus, the optimizer cannot apply a transformation that will destroy the value of x within that region. Again, the only way in which the variable x can become no longer live-out is for the definition of x to be pushed outside the region into block **B**. If this occurs, the compilation manager need only remove x from the live-out set of **A** and the live-in set of **B**.

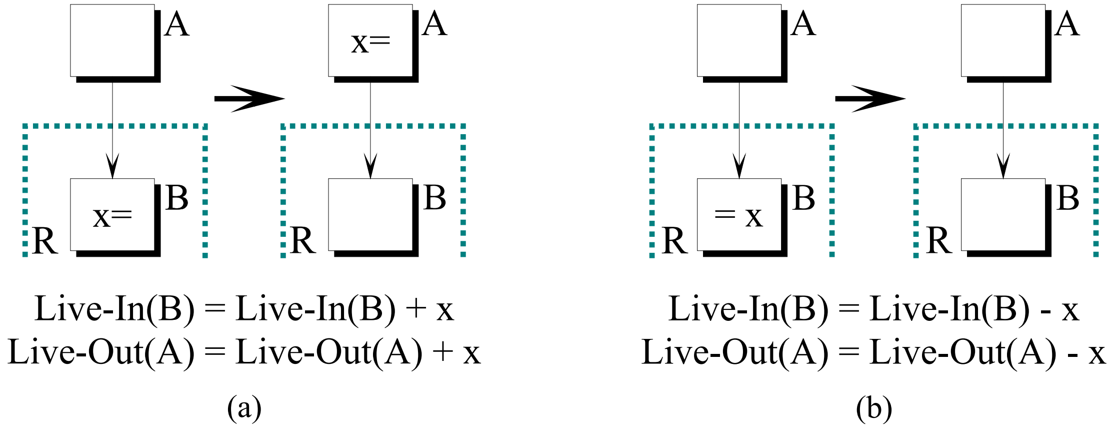

A
x=
A
B
R
x=
B
R
Live-In(B) = Live-In(B) + x
Live-Out(A) = Live-Out(A) + x
(a)
A
A
= x
B
R
B
R
Live-In(B) = Live-In(B) - x
Live-Out(A) = Live-Out(A) - x
(b)

Figure 5.4 Local maintenance of live-in information.

Now, consider the live-in conditions at the entry point of a region. Figure 5.4(a) contains a block **B** that resides within region **R** and a block **A** that does not yet reside within a region. If a variable x exists that is **not** in the live-in set of block **B**, the only way that x can become live-in, as a result of transforming region **R**, is to push the operation defining x outside of block **B** into block **A**. In this case, the compilation manager has only to add x to the live-in set of block **B** and the live-out set of block **A**. The final case is the most interesting. If x is the live-in set of block **B**, as shown in Figure 5.4(b), the optimizer can eliminate the operation that references x from the region. In this instance, the compilation manager can locally update the live-in set of **B** and the live-out set of **A** by removing x. At this point, the live-variable information regarding variable x is correct, but it may be conservative.

The reason for the conservative nature of locally updated live-in information is illustrated in Figure 5.5. If the variable x is defined in block **A** somewhere within the function, and the last use of x resides within the region **R**, then eliminating the last use of x means that x is no longer live-out of block **A**, nor is it live-in or live-out of any block on all control flow paths between block **A** and region **R**. In order to maintain exact live-variable information, the compilation

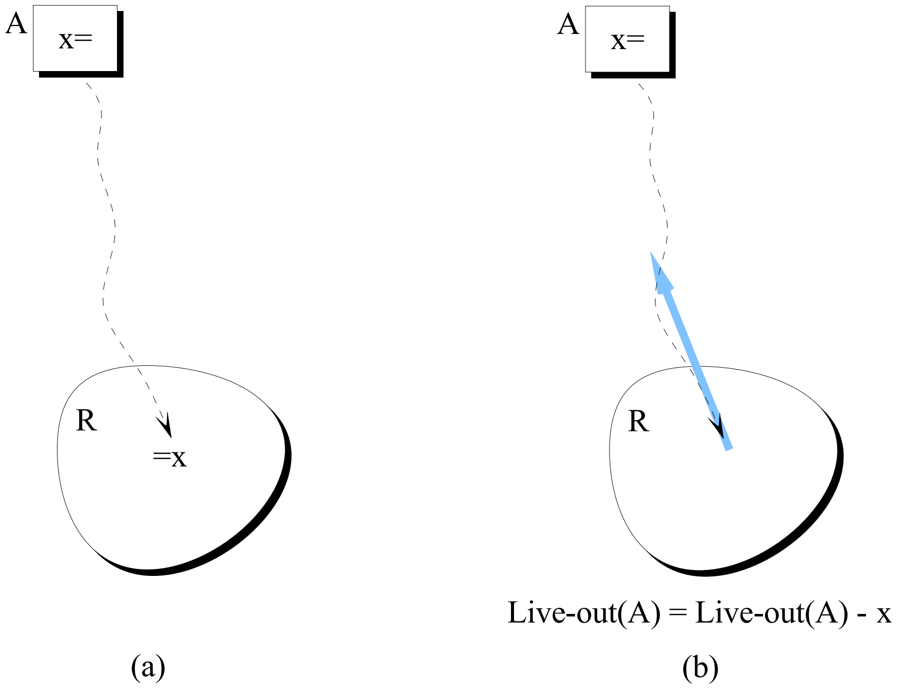


Figure 5.5 Global propagation of live-in information.

manager must update the live-in and live-out sets of all affected blocks. This can be done by performing global live-variable analysis on the entire function to update the live-variable information, which is undesirable. The technique proposed by Pollock and Soffa [60] to allow incremental update of iterative dataflow analysis may reduce the time required to perform this task.

The IMPACT region-based compilation manager limits the updates of live-variable information to the local changes required at the region-boundary. To improve compilation time, the assumption is made that the conservativeness introduced into the live-variable information by not propagating live-variable updates beyond the region boundary will not be detrimental to optimization. The validity of this assumption is investigated in Section 5.3.3 and Section 5.4.2.

Other dataflow information

In addition to live-variable information, the optimizer also makes use of available-expression and available-definition information. Providing this dataflow information to the optimizer allows interregion transformations to occur; however when an interregion transformation is made the live-variable, available-expression and available-definition information must be updated to ensure correct transformation of later regions. Even intraregion transformations may require propagation of available-expression and available-definition changes, just as deleting an operation may result in live-variable changes propagating outside the region. Performing these updates efficiently is challenging. In addition, the feasibility of interregion transformation is unclear within a vertical compilation model. The IMPACT region-based compilation manager does not provide this information under the assumption that the loss of interregion optimization opportunities will not overly affect output code quality. Again, the validity of this assumption is assessed in Section 5.4.2.

5.2.2 Boundary constraints

As shown in Figure 5.1(c), region formation may transform a partial redundancy into a full redundancy within a region by changing the scope of the optimization. However, for the operations in Figure 5.1(c) to become fully redundant, the side entrance to the region must be dealt with. The compiler has two options, each having advantages and disadvantages.

Ignore side entrances

The first option is to allow the compiler to ignore side entrances while performing a transformation. After a transformation has been applied, compensation code may have to be inserted to

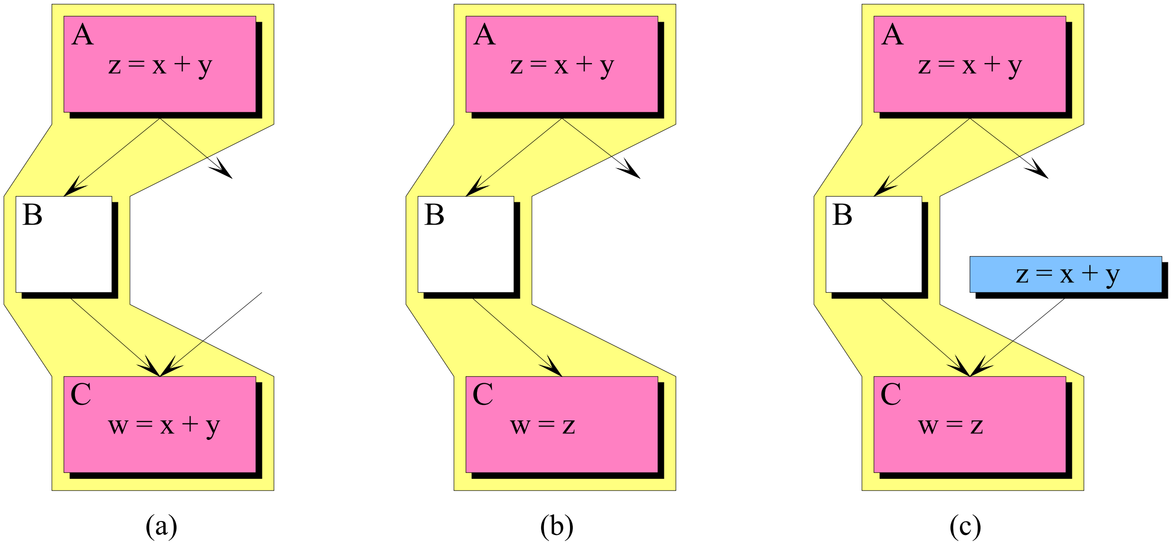


Figure 5.6 Ignoring side entrances: (a) side entrance prevents redundancy, (b) ignoring side entrance allows transformation, (c) insert compensation code.

ensure correct execution. Consider the region shown in Figure 5.6(a). Blocks **A** and **C** contain a redundant computation, but the uncertainty introduced by the side entrance into block **C** precludes full redundancy. If the optimizer were to ignore the side entrance, as shown in Figure 5.6(b), the re-computation of $x+y$ can be replaced by the destination of the first operation, z. However, if either x or y is modified along the path entering block **C**, the compensation code in Figure 5.6(c) is required.

The principal advantage of this technique is that compensation code is only required when a transformation is performed across a region side entrance. If available-expression and available-definition information are present, then the optimizer need only insert compensation code when it is truly necessary. When such information is not present, the optimizer must always insert compensation code to ensure correctness. On the negative side, inserting additional optimizations along the side path may increase the operation dependence height along that path. The path then requires more cycles to execute and is less efficient. Also, by requiring each trans-

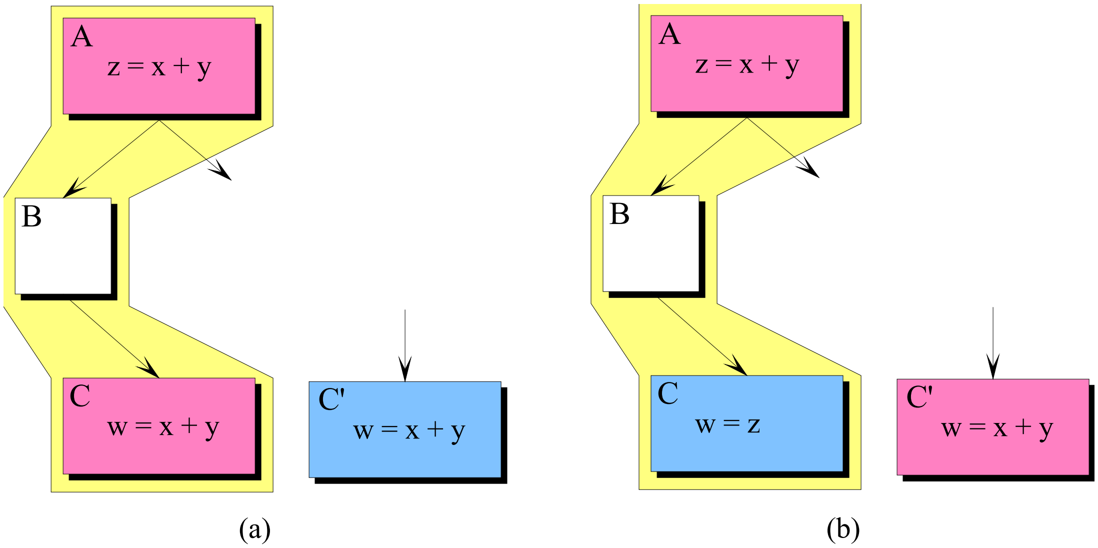


Figure 5.7 Eliminate side entrances: (a) remove via tail duplication, (b) perform transformation.

formation to insert the appropriate compensation code, the implementation and verification of the transformations employed by the optimizer are made more complex.

Eliminate side entrances

The second option is to eliminate the side entrances from the region so that the optimizer need not take them into consideration. A technique known as tail duplication has been applied to remove side entrances from superblocks [56]. This same technique can also be applied to eliminate side entrances from arbitrary regions. Tail duplication eliminates side entrances by duplicating all blocks within a region that are reachable from a side entrance. For example, Figure 5.7(a) contains the region in Figure 5.6(a) after tail duplication has been performed. Block **C** has been duplicated to form a new block **C'**. The computation within the region is now fully redundant and the same transformation can be applied as shown in Figure 5.7(b).

For this approach, the optimizer need not insert any compensation code. All possible compensation code is inserted a priori by the tail duplication process. Thus, conventional optimizations can be applied without modification because the optimizer never sees the side entrances. This approach does, however, result in more code expansion, which has a significant effect beyond increasing static code size. By performing tail duplication prior to the optimization of a region, the amount of code that must be optimized increases. This results in a large number of regions to optimize. If too much a priori tail duplication is done, the compile time benefits of the region partition may be lost. For this reason, the IMPACT compilation manager only performs tail duplication on the most important regions. The subsequent discussion of the behavior of several classical optimizations on region-based compilation units assumes that tail duplication is employed to eliminate region side constraints.

5.3 Optimizations

This section will investigate the effect of region-based optimization on three classical global optimizations: global common subexpression elimination, global copy propagation and global dead code elimination as well as two loop optimizations: loop invariant code motion and global variable migration. The region partitioning and the assumptions made in regard to dataflow conditions at region boundaries will affect the number of optimization opportunities available for each transformation. Finally, this section will assess the overall output code quality of region-based optimized code.

```
For each operation o in a basic block b of the form x = y + z,
where + is a generic operator, such that:


Conditions:
        • expression y + z is available at the beginning of b
        • neither y nor z are redefined prior to o

Actions:
        1. Locate prior evaluations of y + z that reach o
        2. Create a new variable u
        3. For each statement, w = y + z found in (1) by
               u = y + z
               w = u
        4. Replace o, by x = u
```

Figure 5.8 Algorithm: global common subexpression elimination.

5.3.1 Global common subexpression elimination

The goals of global common subexpression elimination (CSE) are to detect and remove operations generating the same subexpression within different basic blocks. Figure 5.8 contains the CSE algorithm [5] applied by the classical optimizer on both function and region-based compilation units. Consider the example application of this algorithm shown in Figure 5.9. Basic block **B** in Figure 5.9(a) contains an arithmetic operation, $x = y + z$, and the expression calculated by the right-hand side of this operation is available at the beginning of basic block **B**. Also, there are no redefinitions of x or y prior to the operation, so both conditions for common subexpression elimination are satisfied. Once the redundant operation in block **A** is located, a new temporary variable t is created and the operation in block **B** is replaced by a copy operation as shown in Figure 5.9(b). Note that the application of CSE does not directly improve the quality of the code. In fact, no operations have been eliminated and the addition of the copy operation in block **A** has increased the dependence height of the block by one and no

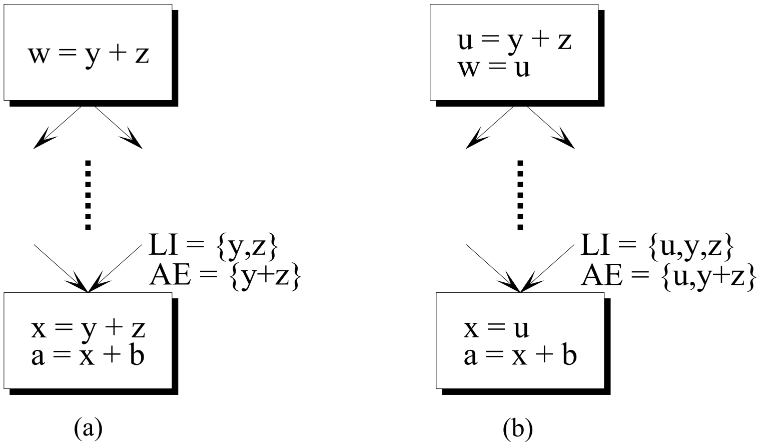


Figure 5.9 Application of global common subexpression elimination: (a) before and (b) after.

operations have been eliminated. Application of copy propagation and dead code elimination are required to realize any benefit.

From the algorithm in Figure 5.8 the global dataflow information required for global CSE is expression availability information. The only operations that are candidates for global CSE are those whose expressions are available at the beginning of the basic block containing the operation. However, the only dataflow information provided at region boundaries is live-variable information, available expression information is not provided. Figure 5.10 will be used to illustrate the effect of this and region partitioning, in general, on the number of CSE opportunities available to the optimizer. Figure 5.10(a) contains a small control flow graph, in which a region has been selected to contain basic blocks **B**, **C**, **E** and basic blocks **A** and **D** have been excluded. This figure contains several opportunities for application of global CSE within a function-based environment between operation pairs: (op1, op2), (op4, op5), and (op3, op6). Each basic block has been annotated with available-expression information as computed by the optimizer. The

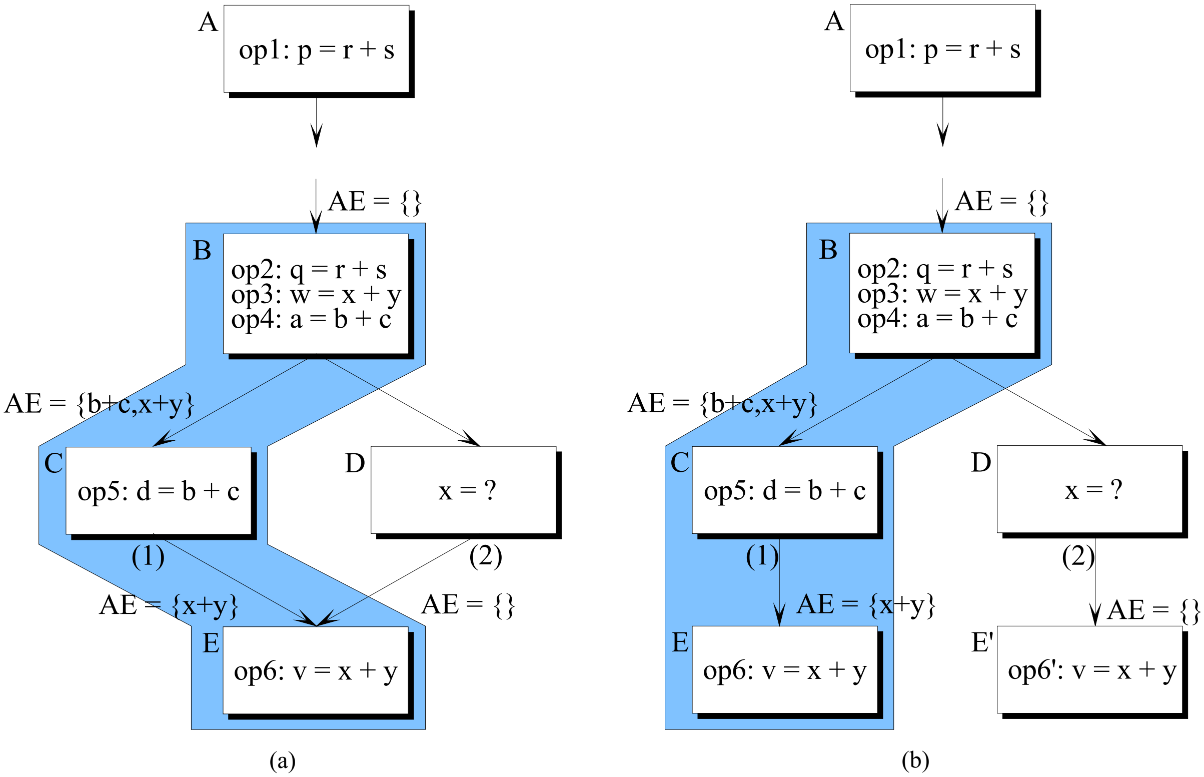


Figure 5.10 Application of global common subexpression elimination within a region: (a) with side entrances and (b) without side entrances.

available expression information for the region entry points is empty, since the optimizer has no visibility outside the region and the compilation manager is not providing the information.

The CSE opportunity that exists between op1 and op2 is unavailable to the optimizer, since the available expression information indicates that $r + s$ is not available at the entry point of basic block **B**. In general, under the assumption that available expression information is not available at region boundaries, all interregion CSE opportunities are lost to the region-based optimizer. A similar situation occurs for the CSE opportunity that exists between op3 and op6. The expression $x + y$ is available along path (1) entering block **E**, but the availability of $x + y$ along path (2) is unknown. Given that tail duplication is being utilized to eliminate constraints from side entrances, the region-based optimizer would actually view the region as shown in Figure 5.10(b).

There are two scenarios depending upon whether or not the expression $x + y$ is fully available at block **D**. If the expression $x + y$ is truly available along path (2), i.e., neither x nor y is redefined in block **D**, global CSE may be applied between op3 and op6 as in the function case. However, if $x + y$ is not available along path (2), tail duplication allows the region-based optimizer to perform global CSE, where a function-based optimizer could not without the aid of partial redundancy techniques. In this instance, the region partition has made an additional optimization opportunity available to the classical optimizer by removing the incoming path from block **D** from consideration; thus, its contents cannot hinder optimization. The final global CSE opportunity between op4 and op5 presents no problem for region-based CSE since both operations reside within the region and there are no side entry points between the two operations.

Boundary condition effects

Recall from the example application of global CSE in Figure 5.9 that a new variable is created as a result. It is apparent from the global CSE examples in Figure 5.10(b) that application of global CSE to a region does not alter the liveness conditions at the region boundaries, since the new variables that would result from applying global CSE between operations (op3, op6) and operations (op4, op5) are entirely contained within the region boundaries. Thus, global CSE requires no updates to the global live-variable information maintained by the compilation manager. However, if available-expression information were provided and global CSE were applied between op1 and op2, then the compilation manager must update, to ensure correctness,

Table 5.4 Region-based performance of global common subexpression elimination.

		Region							
	Function	**Expected Loss**		**NoTail**		**Tail**		**Ideal**	
Benchmark	**#**	**#**	**%**	**#**	**%Loss**	**#**	**%Loss**	**#**	**%Gain**
008.espresso	3,633,375	1,305,965	36%	2,179,945	40%	2,529,724	30%	3,942,241	9%
022.li	171,170	268	0%	240,883	-41%	265,109	-55%	227,991	33%
026.compress	4,109,656	57	0%	4,109,599	0%	4,109,599	0%	4,109,656	0%
072.sc	152,991	61,192	40%	57,060	63%	112,993	26%	186,405	22%
085.cc1	2,929,233	808,766	28%	1,822,509	38%	2,094,147	29%	3,193,166	9%
134.perl	6,573,974	216,833	3%	6,132,905	7%	12,599,737	-92%	13,182,904	101%
cccp	2,243	2	0%	2,074	8%	1,967	12%	2,321	3%
eqn	280,224	2,560	1%	200,187	29%	282,130	-1%	313,191	12%
lex	239,410	60,688	25%	175,335	27%	144,505	40%	239,597	0%
tbl	176,402	23,295	13%	127,568	28%	175,174	1%	216,478	23%
yacc	428,292	127,015	30%	257,940	40%	261,771	39%	428,470	0%

the live-variable information for every basic block between block **A** and block **B** in the control flow graph of the function to account for the new variable created between op1 and op2.

Global CSE opportunities

The effect of region partitioning on global CSE can be quantified by applying the optimizer to both function-based and region-based compilation units and summing the number of dynamic global CSE opportunities available to the optimizer. Each application of global CSE is weighted by the execution frequency of the operations involved in order to determine the relative importance of that optimization. That is, if global CSE is applied to an operation that is executed 100 times, that application of global CSE is given weight 100. Table 5.4 contains the available global CSE opportunities in four different optimization environments for several benchmarks.

The **Function** column contains the number of dynamic opportunities to apply global CSE in a standard function-based environment. The **Expected-Loss**(%) column contains the number (percentage) of global CSE opportunities that occur across region boundaries. These opportunities will not be visible to the region-based optimizer under the assumption that global available-expression information is not provided. Under this assumption, region-based optimization will lose 30-40% of the global CSE opportunities available to a function-based optimizer. For the most part these interregion CSEs result from global CSEs between operations in different loop bodies that are selected into different regions because the loops tend to iterate frequently. However, in order for global CSE to be applied between these two operations, the subexpression must be available at the second operation. Since the second operation is contained within a loop, the fact that the subexpression is available implies that the operation is loop invariant. Thus a later pass of loop-invariant code motion will eliminate the operation from the loop, lowering the significance of this lost optimization.

The **Region-NoTail** column shows the number of global CSEs available to the region-based optimizer without tail duplication. Subtracting the expected loss from the number of function-based global CSEs provides an upper bound on the number of global CSEs available in this environment since the absence of global available-expression information disallows interregion CSEs. The **Region-NoTail** column shows that several of the benchmarks, **008.espresso**, **134.perl**, and **lex**, come relatively close to achieving this upper bound, while the benchmarks **085.cc1** and **yacc** do not. The reason performance is less than expected for **085.cc1** and **yacc** is that, in this environment, the optimizer must make conservative assumptions about constraints from region entry points. Recall the CSE opportunity between op3 and op6 in Figure 5.10. Without tail duplication, the optimizer must assume that the computation is only partially redundant so it can not apply global CSE in this case. The anomalous **022.li** value which is higher than the upper bound results from the phase ordering of global optimizations within the optimizer. Constraints from a side entrance prevent a copy propagation that exposes an application of global CSE elsewhere in one region of **022.li**. If the copy propagation is performed first as in the function-based environment, global CSE is not possible between those two operations.

The **Region-Tail** column shows the number of global CSEs available to the region-based optimizer when tail duplication is applied. This represents the best case for global CSE without available-expression information. There are two combined effects within this column: the additional optimization opportunities exposed by tail duplication and optimization opportunities lost via tail duplication. The additional optimizations exposed via tail duplication actually allow all but two of the benchmarks, **cccp** and **lex**, to exceed the the upper bound imposed by the region partition, as indicated by the **Region-Tail** column. The region-based optimizer actually had almost twice as many global CSE optimization opportunities in the benchmark **134.perl** as did the function-based optimizer. The benchmarks **lex** and **cccp** appear to have lost optimization opportunities as a result of tail duplication. This occurs when an entire loop body is tail duplicated. Consider the cyclic region shown in Figure 5.11(a). Application of global CSE between blocks **A** and **B** is given a weight of 100. After tail duplication, the CFG appears as shown in Figure 5.11. The execution frequency of basic block **B** is now split between **B** and $\mathbf{B}'$. Thus, applying global CSE within the region now only provides a potential benefit of weight $100 - z$. Global CSE cannot be performed between **A** and $\mathbf{B}'$, because $\mathbf{B}'$ is outside of the region of **A**.

The final two columns show the global CSE opportunities available to an ideal region-based optimizer, where global available-expression information is provided, and the percent increase over function-based optimization. Given that the compilation manager does not provide this information, an ideal region-based optimizer can be emulated by a function-based optimizer applied to a function where region selection and tail duplication have been performed. Thus, the

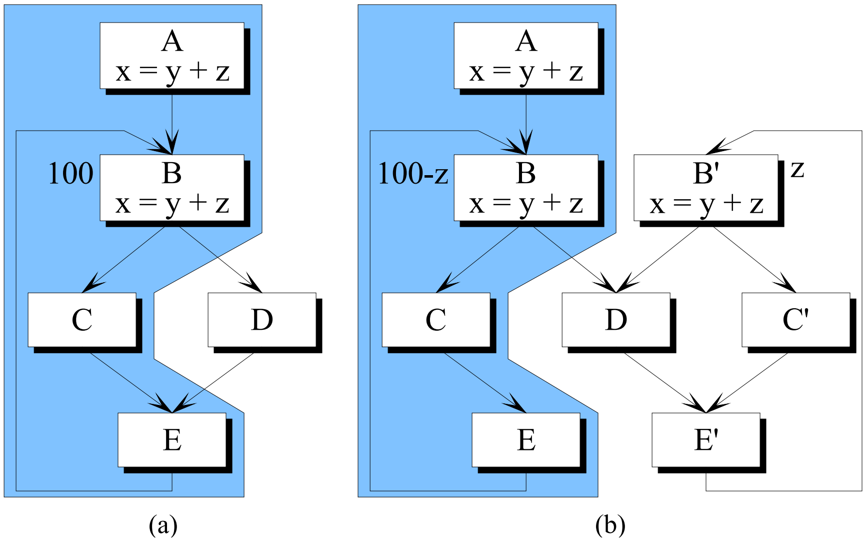


Figure 5.11 Loss of a global common subexpression due to tail duplication: (a) cyclic region before tail duplication and (b) after tail duplication.

optimizer has access to accurate available-expression and available-definition information as well as the additional optimization opportunities exposed by tail duplication. With the exception of **026.compress**, **lex**, and **yacc**, the region partitioning, after application of tail duplication, provided a significant percentage increase over the global CSE opportunities available in a function-based environment with the increases ranging from 3-101% with an average increase of 20%. Even though the ideal region-based optimizer has an apparent advantage over the region-based optimizer without available-expression information in terms of the number of global CSEs available, Section 5.4 will show whether or not this advantage manifests itself as an improvement in output code performance.

5.3.2 Copy propagation

The goals of global copy propagation (CP) are to detect and remove unnecessary copy operations across basic blocks. Figure 5.12 contains the global CP algorithm applied by the

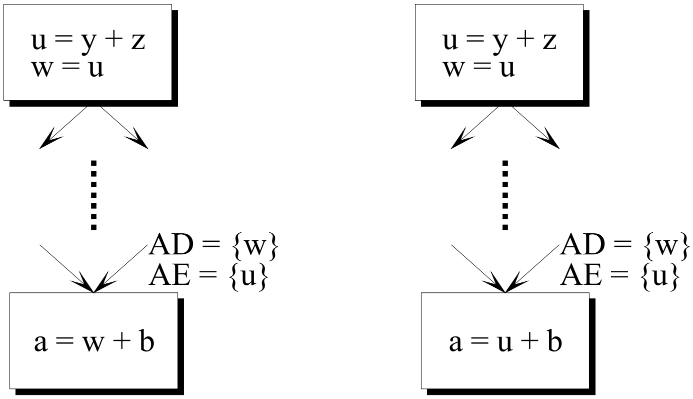
```
For each copy operation o: x = y, find all operations u in a
basic block b using x, such that:


Conditions:
        • definition of x is available at the beginning of b
        • expression y is available at the beginning of b
        • neither x nor y is redefined prior to within b u

Actions:
        Replace use of x in u by y

Region Extension:
        If basic block b is a boundary condition block, rather
        than replace x by y in u, insert a copy of operation o
        in b prior to u.
```

Figure 5.12 Algorithm: global copy propagation.



(a) (b)

Figure 5.13 Application of global copy propagation: (a) before and (b) after.

classical optimizer on both function and region-based compilation units. Consider the example application of this algorithm shown in Figure 5.13. Basic block **A** contains a copy operation,$w = u$, and basic block **B** contains an operation that references w, $a = w + b$. According to dataflow analysis, the definition of w and the expression u are both available at the beginning of basic block **B**. Given there are no redefinitions of w or u prior to the operation, all conditions for copy

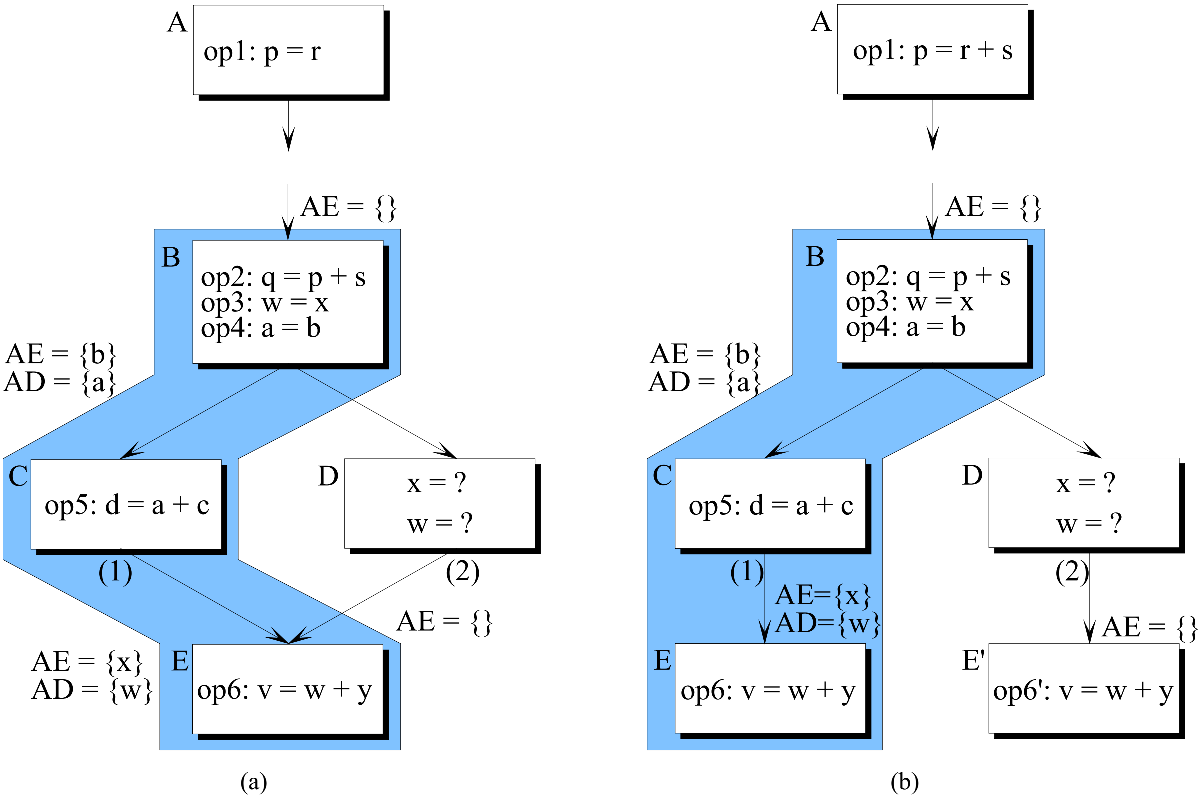


Figure 5.14 Application of global copy propagation within a region: (a) with side entrances and (b) without side entrances.

propagation are satisfied. The use of w is replaced by a use of u as shown in Figure 5.13(b). Note that the application of global CP does not directly improve the quality of the code since no operations have been eliminated. A subsequent application of dead code elimination is required to realize any benefit.

From the algorithm in Figure 5.12, both available-expression and available-definition information are required for global CP. The only operations that are candidates for global CP are those copies whose source and destination operands are both available at the beginning of the basic block that contains an operation using the destination of the copy. Figure 5.14 will be used to illustrate the effect of region partitioning, in general, and the lack of global available-expression and available-definition information, in particular, on the number of global CP opportunities available to the optimizer. Figure 5.14(a) contains a small control flow graph, where a region has been selected to contain basic blocks **B**,**C**, **E** and basic blocks **A** and **D** have been excluded. This figure contains several opportunities for application of global CP within a function-based environment between operation pairs: (op1,op2), (op4,op5), and (op3, op6). Each basic block has been annotated with available-definition and available-expression information as computed by the optimizer. The available-definition and available-expression sets at the region entry points are empty, since the optimizer has no visibility outside the region and the compilation manager is not providing the information.

Similar to the global CSE example, the global CP opportunity that exists between op1 and op2 is unavailable to the optimizer, since available-definition and available-expression information is not provided at region entry points. As with global CSE, interregion CP opportunities are lost to the region-based optimizer without this dataflow information at region boundaries. To minimize the effects of this, the global CP propagation algorithm has been extended to allow the optimizer to place copy operations in boundary condition blocks. Consider the example in Figure 5.15(a). Block **A** contains a copy operation and block **BC**, a boundary condition block, contains a dummy operation that uses the destination of the copy, x. The presence of the dummy operation indicates that the variable x is live outside the region. The region boundary is preventing an opportunity for application of global copy propagation. Replacing the variable x in the dummy operation with y will interfere with the live-variable maintenance performed by the compilation manager, instead a copy of the copy operation is placed in the boundary block as shown in Figure 5.11(b). The contents of the boundary block will later be formed into a region containing a true use of the variable x, effectively allowing interregion global CP.

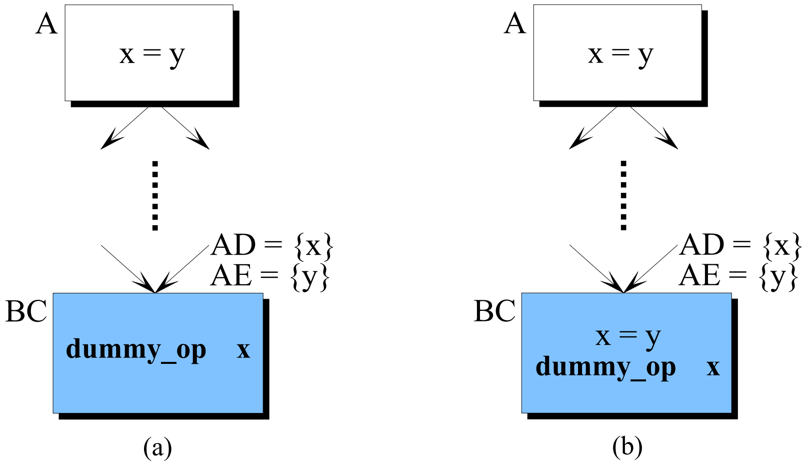


Figure 5.15 Global copy propagation and boundary condition blocks: (a) copy propagation opportunity into boundary condition block, (b) duplicate copy operation inserted into boundary condition block.

The global CP opportunity between op3 and op6 is affected similarly as the opportunity between op1 and op2. The definition w and the expression x are available along path (1) entering block E, but availability along path (2) is unknown. Given that tail duplication is being utilized to eliminate these constraints, the region-based optimizer sees the region as shown in Figure 5.14(b). There are two scenarios for application of global CP depending upon the availability conditions of w and x along the control path entering block E from path (2). If both are truly available along path (2), i.e., neither x nor w is redefined in block **D**, then tail duplication will allow application of global CP between op3 and op6 as in the function-based case. However, if both conditions are not met along path (2), tail duplication still allows the application of global CP, where a function-based optimizer could not perform the transformation without partial redundancy support. In this instance, the region partition has prevented the contents of block **D** from hindering optimization. The final global CP opportunity between op4 and op5 presents no difficulty for the region-based CP since both operations reside within the region and there are no side entrances between the two operations.

Boundary condition effects

Global copy propagation does not eliminate any operations; however, it does extend the lifetime of the source operand of the copy operation and potentially shorten the lifetime of the destination. Unlike the global CSE, global CP may alter the live-out conditions at region exit boundarys by inserting copy operations into region boundary blocks as shown in Figure 5.15(b). Thus, global CP may require local updates to the global live-variable information maintained by the compilation manager. If available-expression and available-definition information were provided, and global CP were applied between op1 and op2 in Figure 5.14(b), then to maintain correctness, the live variable information for every basic block between block **A** and **B** must also be updated.

Global CP opportunities

The effect of region partitioning on global CP is quantified in the same manner the behavior of global CSE is quantified in Section 5.3.1. Table 5.5 contains the global CP opportunities found for each of the four optimization environments. The **Function** column shows that there is a significantly higher number of opportunities for application of global CP than global CSE. From the **Expected-Loss** column, the opportunities for global CP tend to be less localized than the opportunities for global CSE. Many more of the opportunities for global CP appear to span region boundaries. The benchmarks **085.cc1**, **cccp**, **lex**, and **yacc** are expected to miss from 50-95% percent of the global CP opportunities because of the region partitioning, under the assumption that available-expression and available-definition information is not provided.

Table 5.5 Region-based performance of global copy propagation.

		Region							
	Function	Expected Loss		NoTail		Tail		Ideal	
Benchmark	#	#	%	#	%Loss	#	%Loss	#	%Gain
008.espresso	9,350,526	2,397,941	26%	6,304,575	33%	8,498,979	9%	9,516,877	2%
022.li	3,934,863	196,247	5%	2,948,551	25%	4,623,645	-18%	4,980,856	27%
026.compress	752,715	52	0%	752,663	0%	752,663	0%	752,715	0%
072.sc	3,945,843	515,297	13%	3,362,071	15%	3,682,381	7%	4,219,733	7%
085.cc1	5,803,730	2,887,251	50%	3,377,018	42%	4,887,105	16%	6,951,267	20%
134.perl	248,306,405	33,522,427	14%	125,670,073	49%	265,330,973	-7%	277,700,396	12%
cccp	41,706	33,930	81%	10,413	75%	19,129	54%	45,686	10%
eqn	2,357,645	65,896	3%	2,300,027	2%	5,459,425	-132%	7,723,819	228%
lex	353,778	211,098	60%	150,634	57%	273,275	23%	435,272	23%
tbl	529,070	14,511	3%	467,472	12%	486,962	8%	559,549	6%
yacc	1,431,321	1,359,784	95%	70,637	95%	112,957	92%	1,472,676	3%

The **Region-NoTail** column shows that without tail duplication, the optimizer is constrained from applying global copy propagation in many instances. Recall that the upper bound on the available global CP opportunities is determined by subtracting the expected loss from the number of function-based global CPs. Note that several of the benchmarks, **085.cc1**, **cccp**, **eqn**, and **lex**, actually exceed this upper bound even without tail duplication. For these benchmarks, the extension to the global CP algorithm that allows interregion global CP is finding several interregion global CP opportunities. In general, however, the remaining benchmarks fall short of the upper bound because the conservative assumptions made at side entrances are dominating any benefit gained from interregion global CP. Global CP opportunities such as the one between op3 and op6 in Figure 5.14 are unavailable, because the optimizer must assume that the copy is not fully redundant.

As indicated by the **Region-Tail** column, tail duplication significantly improves the number of global CP opportunities. Again, this represents the best case for global CP without available-expression and available-definition information at the region boundaries. Recall that there are two combined effects within this column: the additional optimization opportunities exposed by tail duplication and optimization opportunities lost via tail duplication. Tail duplication can actually hide optimization opportunities from the optimizer as shown in Figure 5.11. It is this effect that prevents the region-based optimizer from finding all available opportunities. For example, there are 486,962 region-based global CP opportunities in **tbl** after tail duplication. Adding the expected loss of 14,511, gives a total of 501,473, which is still less than the ideal region-based performance of 559,549 global CP opportunities.

The ability to perform a limited amount of interregion global CP is manifesting itself as an apparent inconsistency for the benchmarks **008.espresso**, **085.cc1**, **134.perl**, **cccp**, and **lex** in the **Region-Tail** environment. Adding the expected loss to the number of global CP opportunities found in the **Region-Tail** environment yields a number higher than the ideal value shown in the **Region-Ideal** column. Again, this shows that the region-based optimizer is able to take advantage of some interregion global CP opportunities so that it does not take the entire expected loss.

The **Region-Ideal** column shows the global CP opportunities available to an ideal region-based optimizer in which global available-expression and available-definition information is provided and perfectly maintained. The ideal region-based optimizer is emulated as described in the previous section. With the exception of **026.compress**, the region partitioning provided a significant percentage increase over the global CP opportunities available in a function-based environment with increases ranging from 2-228%, an average increase of 30%. As with the global CSE, the ideal region-base optimizer has an apparent advantage over the region-based optimizer without available-expression and available-definition information in terms of the number of global CPs available. Wide-issue processors may be able to absorb the slightly less efficient

```
For each operation $o$ in basic block $b$, such that:

Conditions:
    • value defined by $o$ is not used in $b$
    • value defined by $o$ is not live-out of $b$

Actions:
    Delete operation $o$
```

Figure 5.16 Algorithm: global dead-code elimination.

code without significantly affecting program execution time. Section 5.4 will show whether or not this advantage manifests itself as an improvement in output code performance.

5.3.3 Dead-code elimination

The goal of global dead-code elimination (DCE) is to remove operations defining variables that are never referenced. Such operations appear quite frequently as a result of the previous transformations: global common subexpression elimination and global copy propagation. In fact the previous transformations do not directly improve the quality of the code. They convert redundant operations into dead code. Dead-code elimination has the responsibility for deleting these dead operations. Figure 5.16 contains the algorithm for dead-code elimination. Application of this algorithm is extremely straightforward. If the variable defined by an operation is not referenced within the home basic block of the operation and the variable is not live-out of that basic block, the operation may be deleted. The absence of the variable from the live-out set of the home basic block implies that it is not going to be used later. The converse is also true. The presence of the defined variable in the live-out set of the home basic block implies that it is referenced by a later operation. Whether this implication is true or not depends upon the accuracy of the provided live-variable information.

Boundary condition effects

Unlike the previously discussed transformations, dead-code elimination directly affects the liveness conditions at region boundary points. Deleting the last use of a variable within a region removes the variable from the live-in set(s) of the region entry point(s). This change can potentially propagate across the function as illustrated in Figure 5.5. If the compilation manager does not propagate this change, the conservative nature of the live-variable information may prevent truly dead code from being eliminated in subsequent regions. In addition, every operation that is deleted may alter the global available-definition and available-expression conditions, requiring the compilation manager to propagate these changes if this information is being maintained.

Global DCE opportunities

The number of global DCE opportunities available to the optimizer depends upon both the amount of dead code exposed by previous transformations, such as copy propagation and the accuracy of the available live-variable information. Recall that the compilation manager has several options available in terms of maintenance of live-variable information. To investigate the effects that conservative live-variable information has on the amount of dead code that can be eliminated by the optimizer, the following five environments are compared: function-based optimization (**Function**), region-based optimization assuming all variables are live-out (**Region-AllLive**), region-based optimization using only local maintenance (**Region-Maint**), region-based optimization with propagation of live-variable changes (**Region-Prop**), and ideal region-based optimization where available-expression, available-definition, and live-variable in-

Table 5.6 Region-based performance of global dead code elimination.

		Region							
	Function	All Live		Local Maintenance		Propagation		Ideal	
Benchmark	#	#	%Loss	#	%Gain	#	%Gain	#	%Gain
008.espresso	40,255,332	24,810,267	38%	39,955,883	-1%	39,957,481	-1%	40,963,780	2%
022.li	5,868,323	2,646,298	55%	6,856,808	17%	6,857,425	17%	7,199,731	23%
026.compress	8,428,192	2,111,414	75%	12,679,479	50%	12,679,479	50%	12,679,531	50%
072.sc	17,823,799	1,179,766	93%	17,759,567	0%	17,759,922	0%	18,291,545	3%
085.cc1	22,728,843	9,250,113	59%	21,194,904	-7%	21,197,024	-7%	23,093,933	2%
134.perl	287,037,211	80,966,208	72%	332,256,681	16%	332,260,003	16%	344,564,257	20%
cccp	419,120	115,484	72%	414,263	-1%	414,677	-1%	440,289	5%
eqn	2,978,864	1,106,227	63%	5,161,797	73%	5,162,055	73%	7,403,547	149%
lex	9,818,256	1,152,635	88%	9,182,902	-6%	9,183,269	-6%	9,826,299	0%
tbl	771,904	133,923	83%	778,863	1%	780,654	1%	852,176	10%
yacc	8,771,553	1,202,686	86%	6,508,690	-26%	6,509,090	-26%	8,791,379	0%

formation is maintained (**Region-Ideal**). Table 5.6 shows the number of dynamic DCE opportunities available in each of these environments.

As the **Region-Ideal** column in Table 5.6 indicates, the ability of the optimizer to perform DCE is severely inhibited by the assumption that all variables within the region are live-out. The only operations that the optimizer is able to eliminate are those that define the temporaries created during application of previous transformations, such as the temporaries created by global CSE. As Section 5.4 will show, the code produced in this environment contains a significant amount of unnecessary operations.

When live-variable information is maintained by the compilation manager, the number of global DCE opportunities, in column **Region-Maint**, can be more easily compared to those available to the function-based optimizer. For the benchmarks such as **008.espresso**, **lex** and **yacc**, the region-based optimizer finds less dead code than the function-based optimizer. This is due primarily to lost opportunities earlier in the optimization process. Recall that these benchmarks lost a significant number of global CSE and global CP opportunities due to the region partition. On the other hand, the benchmarks **022.li**, **026.compress**, **134.perl**, and **eqn** show an increase in the amount of dead code eliminated. This phenomenon results from an increase in opportunities for application of loop-variant code motion and global variable migration. Loop-invariant code motion may create redundant operations in frequently executed loop preheaders, and global variable migration introduces copy operations that are eliminated by subsequent application of local CSE and global DCE.

The next column, **Region-Prop**, shows the available dead code opportunities when the compilation manager propagates live-variable changes, as mentioned in Section 5.2.1. Note that there is not a significant increase in the number of dead operations eliminated over the environment where live-variable information changes are updated locally. The only situation in which more accurate live-variable information will result in more dead code eliminations is when the last use of a variable is deleted from a region and the operation defining that variable is in a region that is not yet processed. Given that the regions are processed in order of dynamic importance, any additional dead code opportunities will reside within the less important regions. Also, note that the live-variable information is naturally propagated among adjacent regions. Thus by processing the most frequent regions first, most likely innermost loops, any live-variable changes are naturally propagated to the outer loops without the need for global updates. In this environment, the global propagation of live-variable information is unnecessary.

Finally, the **Region-Ideal** column shows the benefit of global analysis to a region-based optimizer. In this environment, the optimizer benefits from the additional optimization opportunities exposed by the region partition, yet it does not lose the global DCE → global CP → global DCE sequences since available-expression and available-definition information is available at region boundaries. As a result, the global DCE opportunities found in this case are slightly higher than for the **Region-Maint** environment.

For each operation o of the form $x = y + z$, where $+$ is a generic operator, in a loop l, such that:

Conditions:

- Operands x and y are invariant
- x is a unique definition in l
- All uses of x are reached one by o
- Block containing o dominates all loop exits where x is live-out
- If o is a memory access, there are no memory conflicts or unsafe subroutine calls in l

Actions:

Place operation o at bottom of loop preheader

Figure 5.17 Algorithm: loop invariant code motion.

5.3.4 Loop-invariant code motion

The goal of loop-invariant code motion (LICM) is to remove operations from the loop body whose operands do not change during loop execution. Figure 5.17 contains the LICM algorithm used for both function-based and region-based compilation units. In order for a operation to be a candidate for LICM, all five conditions must be satisfied for all possible execution paths within a loop. Region-based application of LICM can potentially identify more optimization opportunities because only the execution paths of the loop that are contained within the region have to be considered. This approach to loop optimization has been applied successfully in the special case where only one path through the loop is selected, known as superblock loop optimization [19]. Consider the loop shown in Figure 5.18(a). The load operations that are accessing memory locations x and $x + 4$ in basic blocks **B** and **D** are **not** loop invariant because block **E** contains an increment of x. Selecting a region that includes all blocks of the loop except block **E**, as shown in Figure 5.18(b), removes the path containing the increment

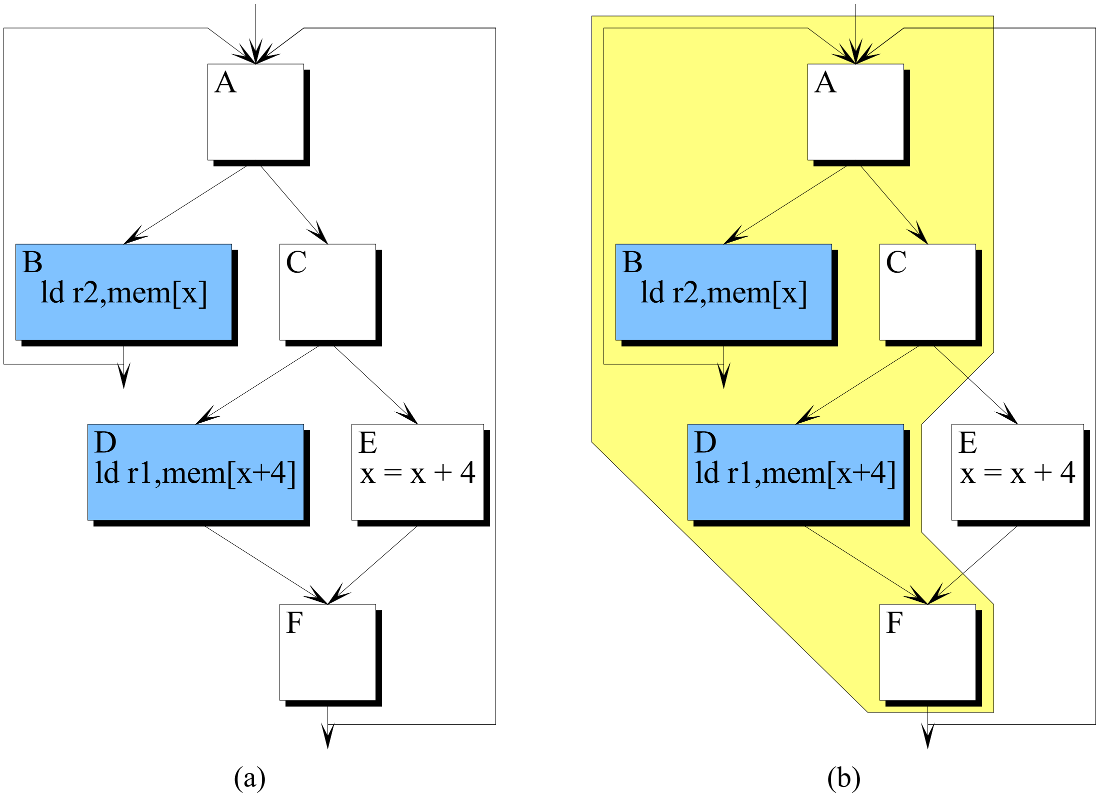


Figure 5.18 Load operations: (a) variant within the loop body, (b) invariant within the selected region.

of x from consideration. Both load operations are loop invariant within the region. In general, excluding infrequent paths through the loop that contain unsafe subroutine calls, ambiguous store operations or operand redefinitions, as in this example, will significantly increase the LICM opportunities available to the optimizer.

Any transformation within the region, however, must ensure correctness when the execution path flows through portions of the loop not contained within the region. Since the region does not contain the entire loop body, any operation removed from the loop as invariant must be re-executed whenever the flow of execution enters the loop body from outside the region. This assumes, of course, that the optimizer considers the selected region a natural loop. A *natural*

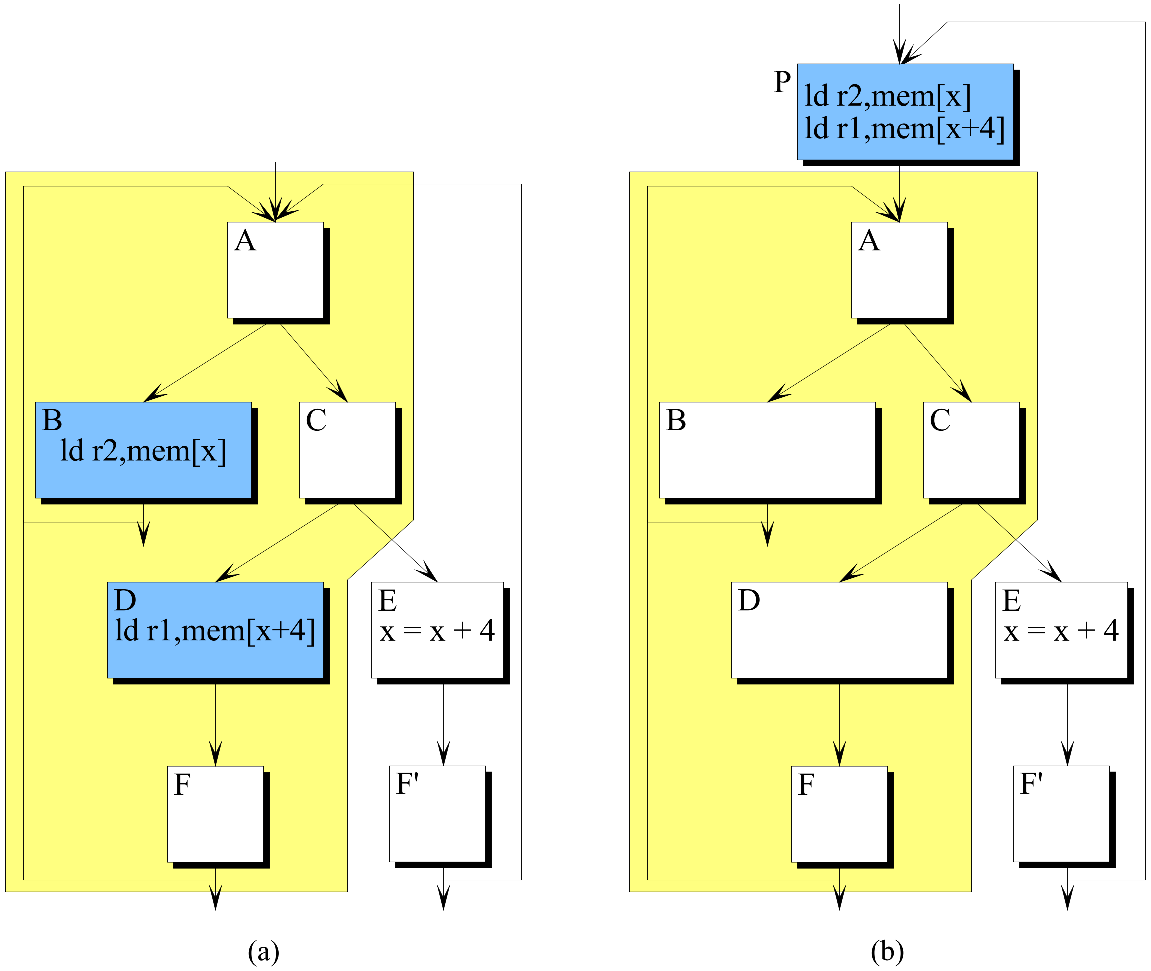


Figure 5.19 Tail duplication: (a) transforms region into a natural loop, (b) ensures invariant code is re-executed properly.

loop is a cycle that contains one entry point and that entry point dominates all other blocks within the loop. The selected region shown in Figure 5.18(b) is not a natural loop because of the side entrance into block **F**.

Both of the above problems can be solved through the application of tail duplication. Consider the CFG shown in Figure 5.19(a) of the loop after tail duplication has been applied to remove the side entrance from the selected region. The selected region is now a natural loop and conventional cyclic transformations, such as LICM, may be applied. Also note that all re-entrant control flow paths now enter the loop through the loop header. Thus, any operations removed from the loop as invariant and placed in a preheader of the selected region will be re-executed when the execution flows through the portion of the loop not selected within the region. As an example, consider Figure 5.19(b), which contains the loop body after LICM has been applied to remove the invariant load operations from the loop body. Both load operations have been removed from the region and placed in a preheader block, **P**. Note that if the flow of execution exits the region from block **C**, it re-enters the loop through the preheader and the load operations are re-executed. This is the desired behavior since the address accessed by the load operations is modified along this execution path.

Boundary condition effects

Application of LICM code motion changes the live-variable conditions at the entry point to the loop, the loop header. The variables defined by all operations moved to the preheader become live over the entire portion of the loop contained within the region. If the loop header lies on a region boundary, which is frequently the case with frequently iterated loops, the preheader should be left outside the region. This has two effects. First, this changes the live-in conditions at the region entry point, but the change requires only local updates by the compilation manager as described in Section 5.2.1. Second, pushing the loop-invariant code outside the region allows it to be optimized in the region formed around the current cyclic region. If that subsequent region is an outer loop, the inner loop-invariant code may be subject to LICM within the outer loop region. In this manner, loop-invariant code may be hoisted out of a deep loop nest even though the entire loop nest is not simultaneously visible to the optimizer.

Table 5.7 Region-based performance of loop-invariant code motion.

		Region					
	Function	Expected Loss		Maintenance		Ideal	
Benchmark	#	#	%	#	%Gain	#	%Gain
008.espresso	26,938,951	3,528,352	13%	44,404,996	65%	44,404,996	65%
022.li	89	0	0%	372,937	*	372,937	*
026.compress	0	0	0%	2,160,310	*	2,160,310	*
072.sc	2,709,081	56,113	2%	8,179,527	202%	8,179,527	202%
085.cc1	14,308,069	2,253,243	16%	15,028,660	5%	15,028,660	5%
134.perl	182,060	3	0%	19,348,876	*	19,348,876	*
cccp	9,870	1,156	12%	8,622	-13%	8,622	-13%
eqn	511,222	451,114	88%	2,605,794	410%	2,605,794	410%
lex	1,691,866	29,447	2%	2,453,287	45%	2,453,287	45%
tbl	91,806	24,679	27%	151,289	65%	151,289	65%
yacc	4,003,486	368,239	9%	6,815,503	70%	6,815,503	70%

LICM opportunities

Table 5.7 contains the available LICM opportunities for three compilation environments: function-based, region-based, and ideal region-based. In addition, the table contains the expected number of global LICM opportunities that are lost due to the region partition. These lost opportunities are operations that are truly loop-invariant; however, the basic block the operation resides in is not contained within a cyclic region, thus the operation cannot be subject to LICM. For example, a loop-invariant operation located in block **E** of Figure 5.19 would reside within an acyclic region and would not be a candidate for LICM. The expected losses due to the region partition ranges from 0% for **022.li**, **026.compress**, and **134.perl** to 88% for **eqn**.

However, as shown in the **Region-Maint** column, the additional opportunities for LICM provided by region partitionning more than compensate for the opportunities lost due to partitioning. For all benchmarks, except for **085.cc1** and **cccp**, the region-based optimizer found at least 45% more, and in most cases far more, loop invariant code than was available to the function-based optimizer. For example, the loop contents of **026.compress** are such that there are no operations that are invariant along all control flow paths; however, the region selected within the main loop exposes over 2M dynamic loop-invariant operations. The loss of LICM opportunities in the region-based optimizer for the benchmark **cccp** stems from the losses due to partitioning and losses due to tail duplication in a manner similar to the way tail duplication results in an apparent loss of global CSE or global CP opportunities.

Unlike the previously discussed transformations, the ideal region-based compiler does **not** have an advantage over a region-based compiler without globally maintained dataflow analysis information. The losses due to the region partition are lost in this environment as well. The addition of global dataflow analysis information cannot change the acyclic nature of a region; thus, the number of LICM opportunities available in the **Region-Ideal** environment is identical.

5.3.5 Global variable migration

The goal of global variable migration (GVM) is to move frequently accessed memory variables, such as globally declared scalar variables, array elements, or structure elements, into registers for the duration of the loop. The loads and stores to these variables are replaced by register accesses for the duration of the loop. Figure 5.20 contains the algorithm for GVM used by the optimizer for both function-based and region-based compilation units. In order for the memory accesses within the loop to be candidates for GVM, all three conditions must be satisfied for all possible execution paths within a loop. As with LICM, region-based application of GVM can identify more optimization opportunities because only the execution paths of the loop that are contained within the region need to be considered. Consider the loop shown in Figure 5.21(a). The load and store operations are accessing the global variable x; however, global variable migration cannot be applied, because of the presence of the ambiguous store

For all memory access operations in a loop *l*, that access the same memory location, *mem[x + y]*, such that:

Conditions:

- Operands x and y are invariant
- There are no ambiguous memory references in l
- There are no unsafe subroutine calls in l

Actions:

1. Replace memory access operations with copy operations
2. Place a load operation in preheader of l
3. Place a store operation in all exits of l

Figure 5.20 Algorithm: global variable migration.

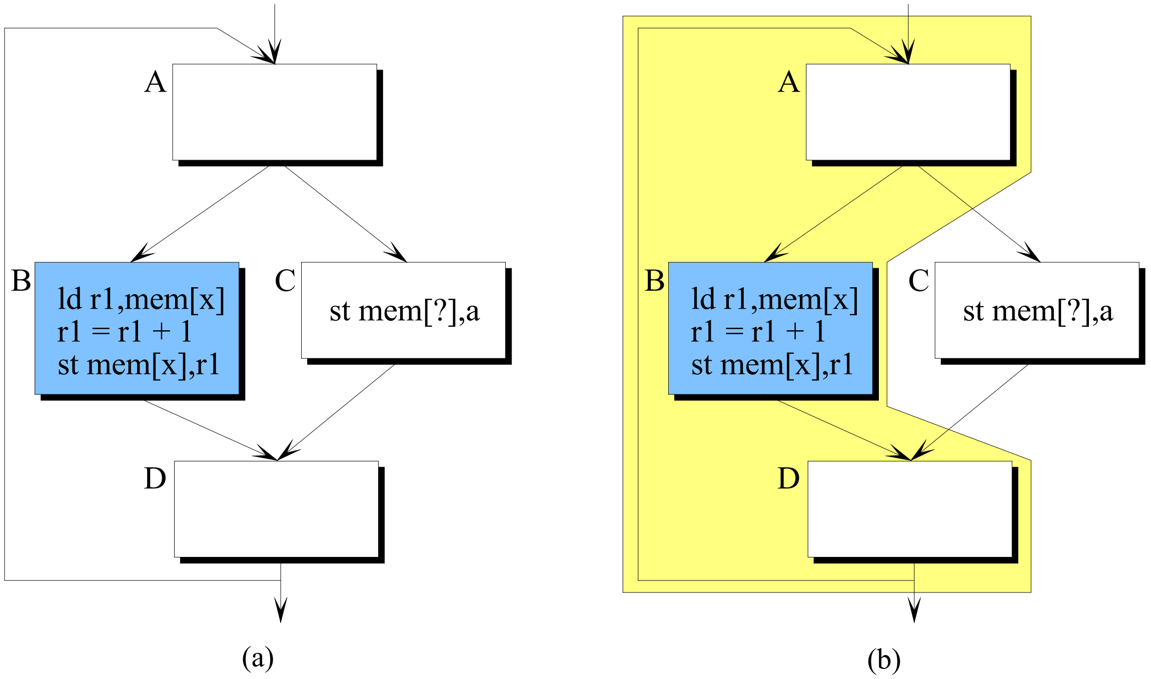


Figure 5.21 Global variable migration (a) increment prevents transformation, (b) optimization exposed within region.

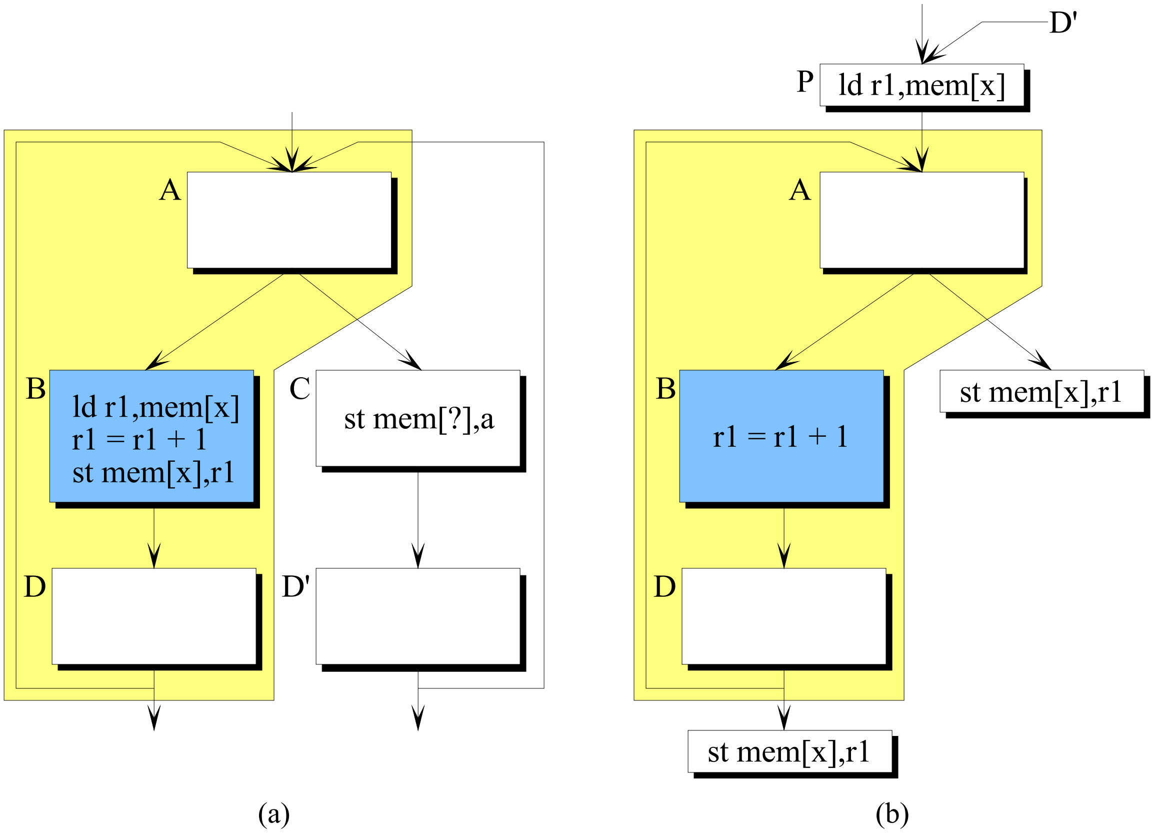


Figure 5.22 (a) Tail duplication transforms region into a natural loop, (b) result of global variable migration.

operation in block **C**. Selecting a region within the loop that excludes block **C**, as shown in Figure 5.21(b), removes the ambiguous store from consideration. GVM may now be applied to the load and store operations in block **B**.

The application of tail duplication by the compilation manager allows the loop optimizer to proceed with GVM transformation without concern for side entrances and the contents of the portions of the loop outside the region, as with LICM. Figure 5.22(a) contains the loop after tail duplication, and Figure 5.22(b) shows the region after global variable migration has been applied. The load operation has been placed in a loop preheader block **P**, and store operations have been placed within boundary blocks at the region exit points, which correspond to the loop exits in this example. The store operation inserted at the exit from block **A** is required in case the exit is a true loop exit or in case the store operation in block **C** does **not** access the same location. This ensures that the load operation in the preheader loads the correct value. Note that the clearly redundant copy operations that replace the load and store operations within the loop body are not shown. They are removed by subsequent application of copy propagation and dead-code elimination.

Boundary condition effects

In addition to altering the live-variable conditions at the entry point to the loop, GVM also changes the live-variable conditions at the loop exit points. The addition of store operations extends the lifetime of the promoted variable beyond the loop boundaries. If the loop exits coincide with the region exit boundaries, the inserted store operations should be pushed outside the region along with the loop preheader, block **P** in Figure 5.22(b). This changes the live-out conditions of the region, requiring local live-variable updates by the compilation manager, as described in Section 5.2.1. Also, the store operations may benefit from memory access optimizations applied to subsequent regions.

Global GVM opportunities

The performance of global variable migration in a region-based framework is evaluated in the same three environments as LICM. The number of available opportunities for GVM in each environment is shown in Table 5.8. Note that with the exception of **008.espresso** and **072.sc**, there are few available opportunities for application of GVM within the benchmarks due to the presence of conflicting memory operations and subroutine calls within loop bodies. As a result, the opportunities lost are negligible for all benchmarks. However, if there exists a load or a

Table 5.8 Region-based performance of global variable migration.

		Region					
	Function	Expected Loss		Maintenance		Ideal	
Benchmark	#	#	%	#	%Gain	#	%Gain
008.espresso	2,971,512	37,394	1%	3,327,694	12%	3,327,694	12%
022.li	6,000	0	0%	871,830	*	871,830	*
026.compress	0	0	0%	7,876,025	*	7,876,025	*
072.sc	1,071,554	348	0%	1,881,219	76%	1,881,219	76%
085.cc1	11,654	0	0%	29,653	154%	29,653	154%
134.perl	76	0	0%	17,845,083	*	17,845,083	*
cccp	4,073	2,292	56%	5,087	25%	5,087	25%
eqn	560	0	0%	534,140	*	534,140	*
lex	5,165	353	7%	3,809	-26%	3,809	-26%
tbl	9,455	0	0%	14,498	53%	14,498	53%
yacc	22,583	333	1%	24,959	11%	24,959	11%

store operation that is accessing the memory location subject to GVM, that resides within the loop, but is not contained within the region, the opportunity to remove that memory access from the loop is lost.

The columns **Region-Maint** and **Region-Ideal** contain the GVM opportunities available to the region-based optimizer. Recall that both cases have the same GVM opportunities available to them. With the exception of **lex**, which loses several GVM opportunities because of tail duplication, the region partition exposes many additional GVM opportunities by removing the constraint that the optimizer must consider all paths within the body of a loop. These additional GVM opportunities, as well as the additional LICM opportunities, will translate into better quality code with fewer dynamic operations and shorter cycle times. This will be discussed in the next section.

5.4 Optimizer Performance

The overall quality of the code produced using region-based classical optimization will be evaluated using a four-issue processor. The processor has no limitations placed on the combi-

Table 5.9 Instruction latencies.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide (SGL)	8
branch	1 / 1 slot	FP divide (DBL)	15

nation of instructions that may be issued each cycle, except that it is restricted to one branch per cycle. This investigation assumes perfect branch prediction and perfect instruction and data caches. Also, infinite registers are assumed for this study, since the effect of region-based register allocation will be investigated in Chapter 6. Finally, the assumed instruction latencies are those of the HP PA-RISC 7100 and are shown in Table 5.9.

5.4.1 Optimization quality

The traditional figure of merit for the evaluation of classical optimization is operation count. Classical optimizations strive to improve the quality of the code by eliminating unnecessary operations. The previous discussion has shown that application of classical optimization within a region-based environment can significantly affect global and loop transformations. In this section, the overall effects of region-based classical optimization are investigated for five compilation environments: function-based optimization **Function**), region-based optimization assuming all variables are live-out (**Region-AllLive**), region-based optimization using only local maintenance (**Region-Maint**), region-based optimization with propagation of live-variable changes (**Region-Prop**), and ideal region-based optimization where available expression, available definition, and live-variable information is maintained (**Region-Ideal**).

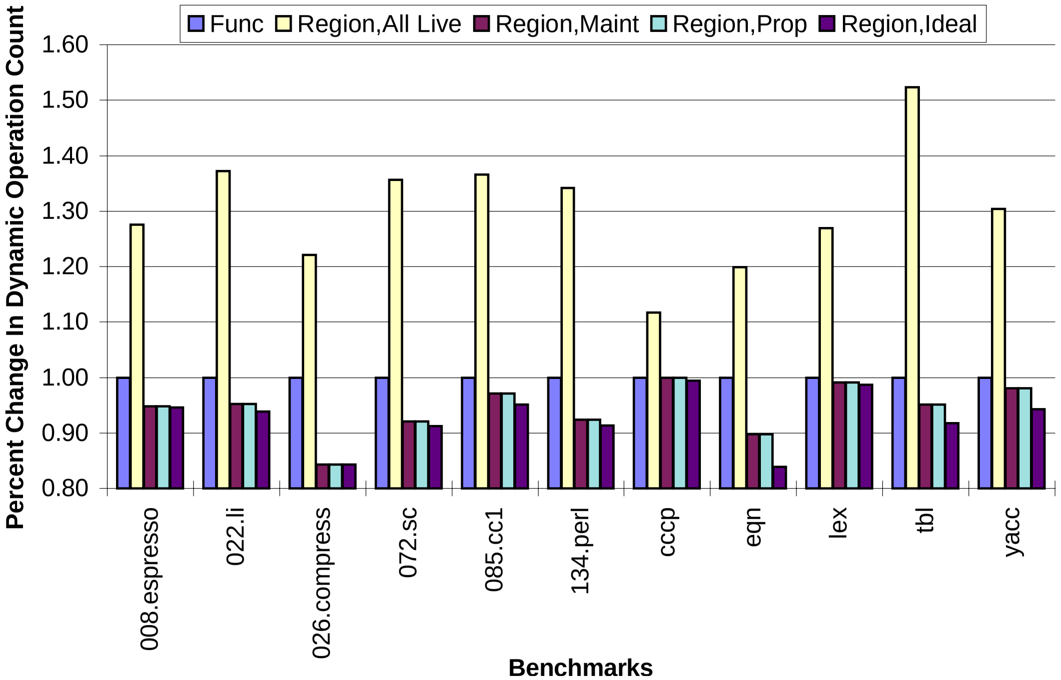


Figure 5.23 Comparison of dynamic instruction counts for function-based and various region-based classical optimization environments.

Figure 5.23 compares the dynamic operation count of each benchmark after classical optimization has been applied in each compilation environment. All results have been normalized to the dynamic operation count resulting from function-based classical optimization, thus the function-based bar has the value 1.0 for all benchmarks. A value less than 1.0 indicates a reduction in dynamic operation count and a value greater than 1.0 indicates an increase. There are several insights that can be gained from this figure.

First, the **Region-AllLive** bar shows that the inability of the optimizer to eliminate truly dead operations in the presence of extremely conservative live-variable information results in a higher dynamic operation count, as expected. In this environment, the region-based optimized code contains 12-52% more dynamic operations than the function-based optimized code. This shows the undesirable side effects of maintaining no global dataflow information. However, when the compilation manager locally maintains only live-variable information, a region-based classical optimizer reduces the dynamic operation count by up to 16% less than the function-based optimizer, as indicated by the **Region-Maint** bar. Across all of the benchmarks an average reduction of 6% is achieved, which clearly indicates that additional loop optimizations exposed by the region partition more than compensate for the lost interregion optimization opportunities. As expected, the **Region-Prop** bar indicates that the additional dead code exposed by propagating only global live-variable information does not have a perceptible impact on the dynamic operation count.

The rightmost bar, **Region-Ideal**, for each benchmark indicates that maintenance of global available-expression, available-definition, and live-variable information provides some additional benefit. The most significant improvements are seen in **eqn**, **tbl**, and **yacc** where the global copy propagation opportunities lost because of the region partition represent a noticeable percentage of the total dynamic operation count. For the most part, this graph shows that the number of dynamic global optimization opportunities lost due to the region partition does not make up a significant portion of the overall dynamic operation count.

Figure 5.24 compares the execution time of the benchmarks under the same compilation environments on a four-issue processor. Again all cycle counts have been normalized to the function-based cycle count. The **Region-AllLive** column shows that the four-issue machine has the bandwidth to absorb the operations that could not be eliminated, unless those operations appear in critic loop bodies, as with **085.cc1** and **tbl**. This is not surprising given the aggressiveness of this machine. More importantly, this graph shows that the region-based classically optimized code produced by maintaining some, **Region-Maint**, or all, **Region-Ideal**, can significantly improve the cycle count over function-based optimized code. This supports

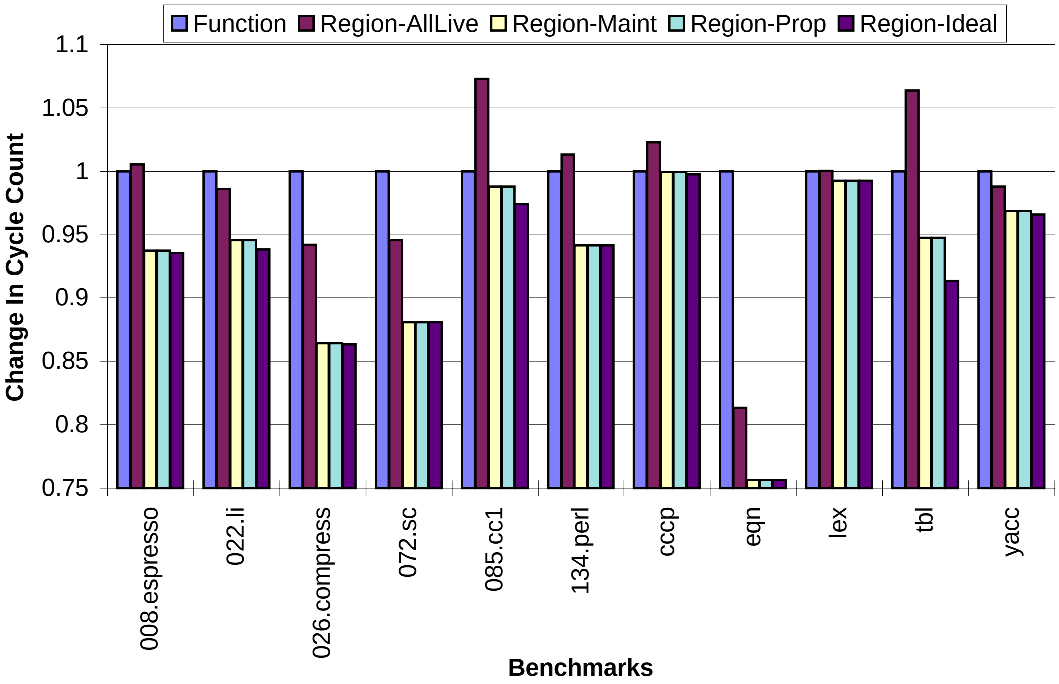


Figure 5.24 Comparison of cycle count for function-based and region-based classically optimized code on a four-issue processor.

the idea that better optimization locally, i.e., within a region, can produce better quality code than global optimization alone.

Figure 5.24 also shows that the decision not to have the compilation manager maintain available-expression and available-definition information is sound. With the exception of **tbl**, the additional optimization opportunities provided by that environment do not significantly improve the cycle count. Within **tbl** there is an important interregion global CSE where the presence of available-expression information allows the ideal region-based optimizer to take advantage of this opportunity, leading to a reduction in the critical path via subsequent global CP and global DCE. However, it should be noted that the statement that maintenance of available-expression and available-definition information is unnecessary is only true for the transformations applied. If an optimizer were to employ a more aggressive transformation such as partial redundancy elimination, the situation may be different.

5.4.2 Optimization time and memory usage

This section investigates the compile time and memory usage benefits of the application of classical optimization within a region-based compilation framework vs. a function-based compilation framework. There are two areas for compile time improvement during the application of classical optimizations. A region-based compiler can potentially exploit both a reduction in problem size and the ability to focus aggressive optimizations only on important areas of the program. In order to fairly investigate the compile time and memory benefits of region-based classical optimization, the same IMPACT classical optimizer was applied to both function-based and region-based compilation units. During function-based compilation, the optimizer applied local, global, branch, and loop optimizations to the entire function, unless dynamic profile information indicated that the function was never called. In this instance, the optimizer only applied local optimizations. To investigate the benefits of problem size reduction, the region-based compile applied treated all regions uniformly, i.e., the same four optimization levels were applied to all regions, unless the region resided within a zero-weight function. As with function-based optimization, only local optimizations were applied. To investigate the benefits of focusing optimization, the region-based compiler applied only local optimizations to zero-weight regions, local, global and branch optimizations to acyclic regions, and all four levels to cyclic regions.

The optimization time improvements for region-based compilation are shown in Figure 5.25. The left bar represents the function-based optimization time for the benchmark over the region-

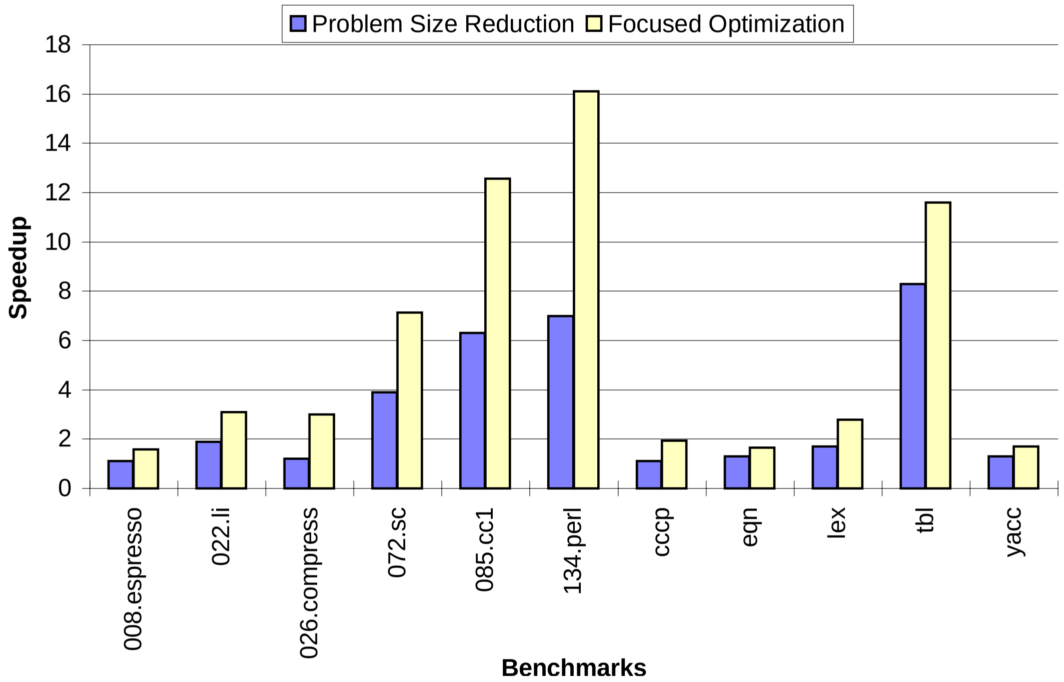


Figure 5.25 Speedup of region-based classical optimization vs. function-based classical optimization.

based optimization time where all regions are treated uniformly. The region-partitioning afforded an average speedup of 3.2 over function-based compilation simply by reducing the optimization scope. Note that the benchmarks that benefit the most from problem size reduction, **072.sc**, **085.cc1**, **134.perl**, and **tbl**, are the benchmarks where the inlining resulted in rather large function bodies. The benchmarks with smaller function bodies, such as **026.compress**, **cccp**, and **yacc**, did not benefit from the problem size reduction, since the functions within those bench marks are not large enough to exacerbate the nonlinear nature of the classical transformations.

The right bar in Figure 5.25 represents the function-based optimization time over the region-based optimization time in which the region-based compiler selected the optimization level based upon the region importance and topology. The addition of this capability increases the average

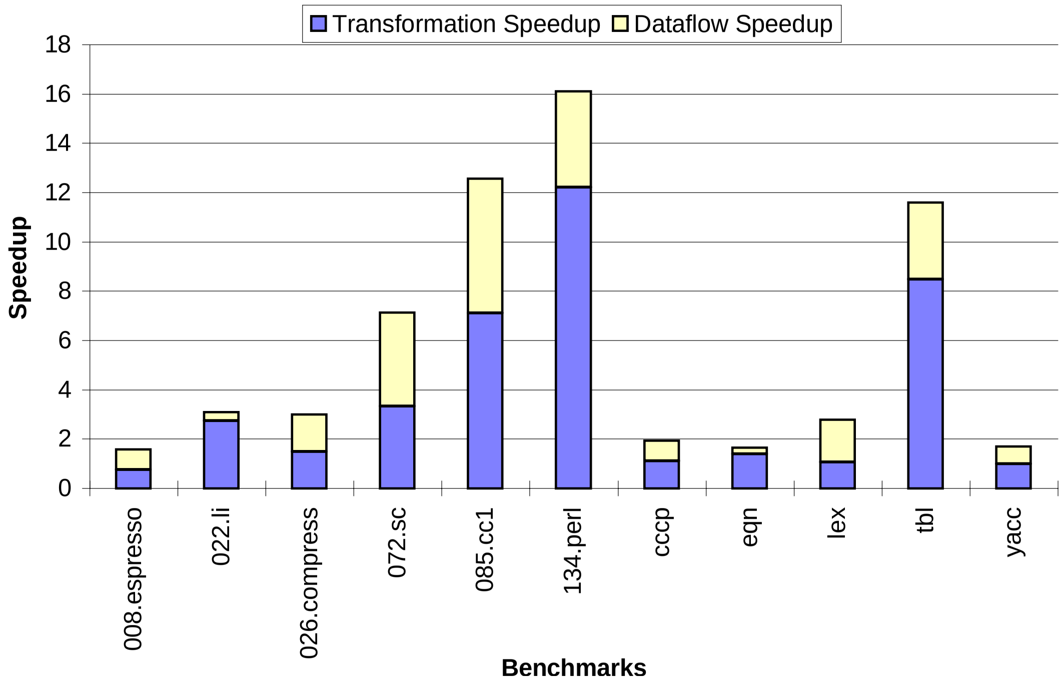


Figure 5.26 Breakdown of region-based speedup between transformation speedup and dataflow speedup.

speedup achieved via region-based compilation to 5.8. These data show that providing the compiler the ability to select optimization levels is more important or just as important as the reduction in problem size. Consider the benchmark **134.perl**. Reduction in problem size results in an optimization time speedup of 7, but by focusing the optimization efforts only on the important regions increases the speedup to 16. The large problem size speedup results from the fact that **134.perl** contains several extremely large function bodies after inlining; however, the dynamic behavior within those functions is localized to a few important regions, resulting in a 16 times speedup when the compiler spends time only where it is needed.

Recall from Section 5.1 that the optimization process can be split into two components: the search for optimization opportunities and the computation of dataflow analysis information. Figure 5.26 contains the speedup breakdown for the region-based compilation model, shown in the right bar of Figure 5.25. In this graph, the speedup bar has been divided into these two components: the portion of the speedup derived from reduction of optimization search time and the portion of the speedup derived from reduction in dataflow analysis time. For all of the benchmarks, the majority of the compilation time speedup results from a reduction in optimization search time. There are two reasons for this. First, the scope of the global and loop optimization is limited by region boundaries. Second, the compiler is applying only local optimization to unimportant regions, which means the scope of optimization is constrained by basic block boundaries. Despite this, a significant portion of the speedup results from reduction of dataflow analysis time, because reducing problem size reduces the amount of time required for the iterative dataflow analysis algorithms to converge. Note that the presence of the boundary condition blocks and dummy operations used to convey live-variable boundary conditions, as discussed in Section 4.4, does affect dataflow analysis time.

In addition to the cost due to boundary operations during region-based optimization, there is additional overhead associated with region-based compilation. This overhead includes the time required for region formation, extraction and encapsulation of regions, re-integration of optimized regions, and maintenance of live-variable information at region-boundaries. Figure 5.27 shows the speedup of region-based optimization including the time required by the compilation manager. The compilation manager overhead ranged from 10-40%, with an average of about 25%, reducing the average speedup for region-based optimization to 4.1. This does not necessarily represent the overhead for region-based compilation as implemented within a production compiler for two reasons. First, the implementation of the compilation manager may not be representative of a production quality implementation. Second, in this experiment classical

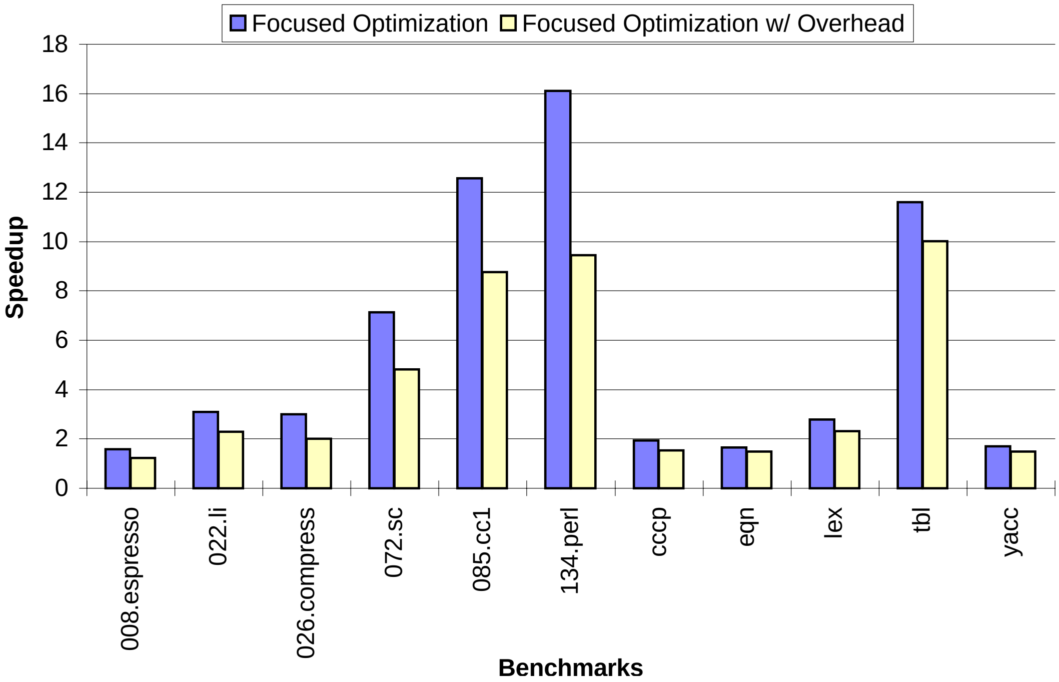


Figure 5.27 Speedup of region-based classical optimization vs. function-based classical optimization including compilation manager overhead.

optimization is absorbing all of the overhead; however, in reality the overhead can be amortized over all phases of the compilation process.

In addition to significant optimization time improvements, region-based optimization provides memory usage improvements as well. Figure 5.28 shows the memory size reduction for region-based classical optimization for each benchmark. This graph contains the ratio of the maximum amount of memory required for function-based classical optimization over the maximum amount of memory required by region-based classical optimization. With the exception of **026.compress**, **cccp**, and **eqn**, which do not contain large function bodies, the amount of memory required for region-based optimization is about half of that required for function-based optimization. On average, region-based classical optimization reduces the memory requirements by 3.8 times. Once again the extreme case, a reduction of over 12 times results in **134.perl** due

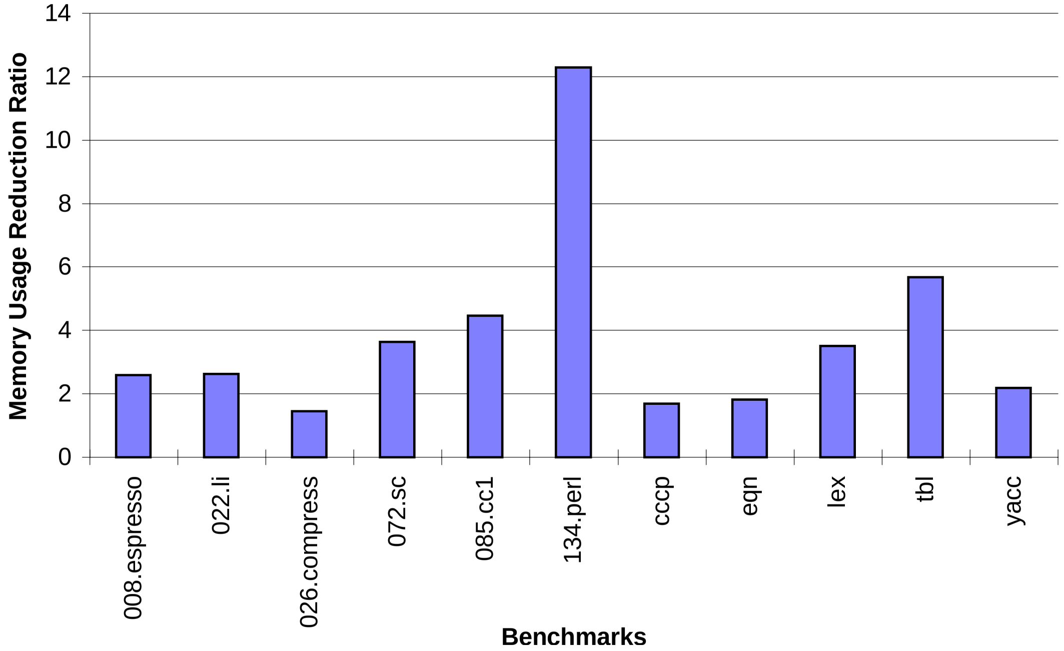


Figure 5.28 Memory usage reduction of region-based classical optimization vs. function-based classical optimization.

to its extremely large function bodies. Containing the optimization scope reduces the amount of memory required to store the dataflow information, memory dependence graph, and hazard analysis information required during optimization.

Finally, Table 5.10 contains the data used to create the graphs in this section. The optimization times, in seconds, are shown for both function-based and region-based compilations on a HP 9000/735,125MHz workstation. The region-based optimization times and overhead are for the model in which the compilation manager is allowed to determine the appropriate optimization level. The maximum amount of memory required by the classical optimizer during both function-based and region-based optimization is shown in increments of 1024 bytes(K).

Table 5.10 Summary of function-based and region-based classical optimization times and memory usages at desired inlining levels.

Benchmark	Function-Based			Region-Based			
	Optimization Time (s)		Memory (K)	Optimization Time (s)			Memory (K)
	Trans	Dataflow		Trans	Dataflow	Overhead	
008.espresso	41	43	2234	26	27	15	861
022.li	132	44	6246	26	31	20	2373
026.compress	3	3	1238	1	1	1	857
072.sc	190	224	10390	24	34	28	2849
085.cc1	4344	3591	23582	207	425	273	5277
134.perl	1521	541	43094	55	73	90	3505
cccp	14	15	1470	6	9	4	869
eqn	72	57	4170	29	49	9	2301
lex	20	33	2730	7	12	4	777
tbl	813	348	10714	39	61	16	1885
yacc	21	25	1562	10	17	4	713

5.5 Summary

Application of classical optimization within a region-based framework can provide significant improvements in output code quality while reducing the amount of time and memory required. The quality of the output code is heavily dependent upon the dataflow information provided at region boundary points. The compiler may maintain no live variable information, maintain local live-variable information, globally propagate live-variable information, or globally maintain availability and liveness information. Of the options available, the assumption that all variables are live-out requires the least amount of work to maintain; however, this information is clearly too conservative. This is reflected by the increased dynamic operation and cyclic counts relative to function-based optimized code resulting from the inability of the optimizer to remove truly dead operations.

With reasonably accurate live-variable information present at region exit points, the region-based optimizer was able to produce better quality code than a function-based classical optimizer. There are two competing effects when applying region-based classical optimization in this environment. The region partition and the absence of availability information at region boundaries prevent the optimizer from taking advantage of opportunities for interregion common subexpression elimination and copy propagation. The results indicate that a significant number of the global CSE and global CP opportunties do occur between regions. On the positive side, the ability of region partitioning to exclude control flow paths from consideration during loop optimizations results in a significant increase in the number of opportunities for application of loop-invariant code motion and global variable migration. The benefits of additional loop optimizations more than compensate for the loss of interregion acyclic optimizations. Region-based optimization is trading global optimization efficiency for better local optimization within cyclic regions.

Global propagation of live-variable information provides little additional benefit over local maintenance because of the order in which the regions are optimized. By using profile information to determine processing order, regions are optimized starting with inner loops and progressing outward through the loop nest. This tends to naturally propagate changes in liveness conditions from the most important regions to the least important regions. As a result, any additional dead code opportunities exposed by the more accurate liveness information tend to occur in infrequently executed portions of the function.

Providing available-expression and available definition information requires the compiler to globally maintain and propagate changes in the availability information, as well as live-variable information to ensure correct transformation. With this information, the optimizer can potentially take advantage of interregion optimization opportunities despite the region partition. However, global maintenance of all three forms of dataflow information may be prohibitively expensive. For the classical transformations applied within this thesis, providing this information did not provide a performance improvement large enough to justify the work required to globally maintain the information. In addition, it is not clear how feasible interregion transformation is within a vertical compilation model. For this reason, the region-based compiler used to investigate the time and memory usage benefits of region-based compilation performed only local maintenance of live-variable information.

In addition to the improvements in output code quality, region-based optimization also provides significant improvements in optimzation time and reductions in optimizer memory usage requirements. The optimization time improvements are due to the reduction in problem size provided by the region partition and perhaps more importantly to the ability to focus the attention of the compiler on important regions while spending little time on unimportant regions. The reductions in memory usage stems from a reduction in the amount of dataflow, memory dependence, and hazard information that must be simultaneously maintained by the optimizer.

CHAPTER 6

REGION-BASED REGISTER ALLOCATION

The purpose of register allocation is to map the variables and compiler temporaries, collectively referred to hereafter as *virtual* registers, of a program to the *physical* register file of the target processor. The goal is to minimize the dynamic number of memory accesses required to access those virtual registers that cannot be assigned, or allocated, to a physical register. The scope of global register allocation typically encompasses the entire function. Global, function-based register allocation is typically mapped to a graph coloring problem [61], [62]. The register allocation process itself generally consists of three steps: interference graph construction, register assignment or graph coloring, and spill code insertion.

The interference graph is the data structure used by the register allocator to ensure a legal allocation. The graph contains a node for each virtual register in the compilation unit and an arc between each pair of variables that interfere, i.e., are simultaneously live. Thus it is illegal to assign the same physical register to two virtual registers connected by an arc within the graph. The interference graph is constructed by comparing the lifetimes of every pair of virtual registers in the compilation unit and inserting an arc between two nodes when the intersection of their lifetimes, or live ranges, is not empty.

Once the interference graph is constructed, the register allocator attempts to obtain a k-coloring of the graph, where k is the number of physical registers in the target processor. Obtaining an optimal k-coloring of the interference graph is NP-complete. For this reason, polynomial time heuristics are used to find a suboptimal coloring. A great deal of work has been done on the subject of coloring heuristics and spill code minimization [61]-[65]. If we assume that the compilation unit on which register allocation is being performed contains n virtual registers, then the computational complexity of register allocation is bound by the complexity of interference graph construction and the complexity of the coloring heuristic. The construction of the interference graph requires $O(n^2)$ live range comparisons. Provided that the comparison itself can be done in constant time, the time complexity of interference graph construction is $O(n^2)$. The coloring heuristic may require up to n passes over the n node interference graph giving the coloring heuristic $O(n^2)$ time complexity as well. Overall, the time complexity of register allocation is $O(n^2)$. Register allocation also has memory complexity of $O(n^2)$ since the interference graph may contain an arc between every pair of n nodes. As such, register allocation can require a large amount of time and memory when applied to large functions.

Application of global register allocation within a region-based compilation framework has two potential advantages. First, confining the scope of register allocation to the contents of a region can potentially reduce register allocation time and memory requirements by simply reducing the number of virtual registers that must be simultaneously considered. Second, focusing the attention of the register allocator on the most frequently executed areas of a function while ignoring constraints from less important areas can potentially improve the quality of the allocation in much the same way the region partitioning exposes additional optimization opportunities. However, because the register allocator loses the global view of the function, it loses the global relative importance of the virtual registers referenced within the region. As with region-based optimization, a higher quality allocation within the region may result in a less optimal global allocation. Yet, by allocating the most important regions first, any additional spill code can be pushed into less frequently executed areas or areas where there are sufficient resources to absorb the additional memory operations.

Several previous approaches to improve register allocation time, memory usage, or quality have made use of the region concept. Callahan and Koblenz [7] partition the register allocation of a function by defining a hierarchical tiling based upon the control flow graph. The tiles are colored individually in a bottom-up fashion, mapping virtual registers to pseudo-registers. In a second, downward pass the pseudo-registers are mapped to physical registers. By allocating leaf tiles first, which tend to correspond to the innermost loop bodies, the regions of the program that would tend to have the highest execution frequently are allocated first. Norris and Pollock [66] propose a similar hierarchical approach, however, the partitioning is based upon the program dependence graph [67] regions, which consist of control equivalent code segments and loop bodies. In this technique, the scope of register allocation tends to be too small, on the order of one C statement, resulting in an unnecessary spill code. Gupta, Soffa, and Ombres [68] use clique separators to reduce the memory requirements of global register allocation. In this approach, a function is partitioned by selecting traces [35] and determining the clique separators within each trace. If the interference graph of a function is divided into m cliques, the memory complexity of register allocation becomes $O(\frac{n^2}{m^2})$ and the time complexity becomes $O(\frac{n^2}{m})$. However, published results show that the overhead of determining the clique separators results in significantly longer overall register allocation time.

The Multiflow compiler utilized the concept of region-based register allocation within the trace scheduler [8], [69]. Rather than applying global register allocation after instruction scheduling as is done in most optimizing compilers, a combined scheduling and register allocation technique is applied to individual traces so that the instruction scheduler can treat registers as machine resources. Register boundary conditions are communicated to the scheduler by means of an explicit data structure called a *Value-Location Mapping* inserted directly into the control flow graph of the function [8]. This will be discussed further in Section 6.2.

The IMPACT register allocator will be used as the testbed to investigate the effects of region-based register allocation in terms of allocation quality and register allocation time and memory usage. As such, Section 6.1 presents a more detailed overview of the IMPACT global register allocator. The boundary condition information required for correct and efficient register allocation of a region is discussed in Section 6.2. Section 6.3 dicusses reconciliation of separately compiled regions. Finally, Section 6.4 assesses the effects of region-based register allocation.

6.1 IMPACT Register Allocator

The IMPACT register allocator [26] will serve as the vehicle to investigate the effects of global register allocation within a frequency sensitive region-based compiler. The encapsulation method discussed in Chapter 4 along with some additional boundary condition information, to be discussed in Section 6.2, will allow the use of the same register allocator for both region-based and function-based experiments. This facilitates the direct comparison of register allocation quality and register allocation time and memory usage differences. Figure 6.1 contains the global register allocation algorithm used within the IMPACT global register allocator. The algorithm has been divided into the three basic steps of register allocation: interference graph construction, register assignment, and spill code insertion. The remainder of the section will briefly discuss each of the three steps as the background for the changes required for application of global register allocation within a region-based environment. The statements marked with

```
Register_Allocation(Func f)
{
    // 1. Interference graph construction
    Compute_LiveVariable(f);
    vr_list = Determine_LiveRanges(f);
    Construct_InterferenceGraph(vr_list);

    // 2. Register assignment, graph coloring
    sorted_vr_list = Determine_Priorities(vr_list); †
    uncolored = 1;
    while ( uncolored ) {
        uncolored = 0;
        for ( vreg = sorted_vr_list; vreg != NULL; vreg = vreg→next )
            reserved = Determine_UnavailRegisters(vreg); †
            register = Find_FreeRegister(vreg,reserved); †
            if ( register != -1 )
                AssignColor(vreg,register);
            else {
                Modify_LiveRange(vreg);
                uncolored += 1;
            }
        }
    }

    // 3. Spill Code Insertion
    Insert_SpillCode(f,vr_list); †
}
```

Figure 6.1 Algorithm: global register allocation.

a † are modified slightly for efficient and correct application of global register allocation to regions.

6.1.1 Interference graph construction

As shown in Figure 6.1, interference graph construction first requires the register allocator to perform global live-variable dataflow analysis in order to construct the live range for each virtual register. The register allocator utilizes the same iterative dataflow analysis method used by the optimizer, the difference is that register allocation requires dataflow analysis to be done only once. Using this information, the register allocator determines the set of operations over which a virtual register is live, its live range. The interference graph is then constructed by placing an arc between two virtual registers if their live ranges contain common operations, as described in the previous section.

6.1.2 Graph coloring

The graph coloring phase of the IMPACT register allocator has two important functions that are relevant to application of global register allocation over regions. They are priority determination and register selection.

Priority determination

The register allocator uses a priority-based approach to color the interference graph [33]. Each virtual register is assigned a profile-based priority to determine the order in which the virtual registers are colored using the following equation.

$$priority(vreg) = \frac{ref\_weight(vreg)^3}{num\_oper(vreg)} \tag{6.1}$$

The *ref_weight()* is the sum of the dynamic weight of the operations that reference the virtual register, and the *num_oper()* is the number of operations contained within the live range. Empirical analysis has shown this to be a very good heuristic for determining the coloring order [26]. During the coloring phase, the register allocator must determine whether a virtual register should be allocated to a register or spilled, i.e., reside in memory on the stack. To make this decision, the register allocator determines the benefit of allocating the virtual register by calculating the cost of spilling it. The cost of spilling a register can be estimated with the following equation.

$$spill\_cost(vreg) = def\_oper(vreg) \times store\_cost + use\_oper(vreg) \times load\_cost \tag{6.2}$$

In the worst case, if the virtual register is not allocated, a store operation is needed after every definition and a load is required for every use. In practice, some of these memory access operations can be eliminated. Essentially the benefit of allocating a virtual register is the number of memory access operations saved by placing it in a processor register.

However, the register allocator must also determine the register saving convention, caller-saved or callee-saved, of the physical register that is most efficient. If a virtual register is allocated to a caller-saved register that register must be saved and restored across every subroutine call within the live range. Thus the true benefit of allocating a virtual register to a caller-saved register is given by

$$caller\_benefit(vreg) = spill\_cost(vreg) - jsr\_weight(vreg) \times (store\_cost + load\_cost) \tag{6.3}$$

where *jsr_weight()* is the sum of the dynamic weight of the subroutine call operations within the live range of the virtual register. If a virtual register is allocated to a callee-save register, that callee-save register must be saved and restored upon entry and exit to a function. Thus, allocation to a callee-save register may require the number of memory access operations given by the following equation.

$$callee\_benefit(vreg) = spill\_cost(vreg) - func\_weight \times (store\_cost + load\_cost) \tag{6.4}$$

where *func_weight* is the execution frequency of the function being allocated. However, if the callee-saved register has previously been allocated to a virtual register, it use is essentially free. During the coloring process, the register allocator selects the type of register, callee-saved or caller-save, that provides the highest benefit. If neither provides a positive benefit, the virtual register is spilled.

Register selection

The coloring algorithm iterates over the interference graph until all virtual registers have been assigned to a processor register. For each virtual register, the physical registers that cannot be assigned to the current virtual register are determined by examining the interference graph. Any physical registers that have been assigned to interfering virtual registers are not candidates for assignment to the current virtual register.

The register allocator uses a first fit algorithm to minimize the number of registers used. The first register in the bank of physical registers that is not contained within the set of unavailable registers is selected. If a free register is not found for any one virtual register during a pass over the interference graph, the register is spilled and its live range is modified to lower the register pressure [26]. Even a spilled virtual register requires a register to hold the value to be stored or loaded from the stack.

6.1.3 Spill code insertion

Since the IMPACT register allocator assumes that all variables reside within virtual registers prior to allocation, memory accesses must be inserted for virtual registers that could not be allocated. The register allocator inserts a store operation after each definition and a load operation before each use of a spilled virtual register. If the allocation permits, the register allocator may not have to insert a load operation before each use. Separate allocation of regions requires an extension of this capability to insert the appropriate compensation code to correct for the separate register allocation.

6.2 Boundary Conditions

The separate register allocation of regions differs from the separate allocation of functions, in that the information flow between functions is achieved through memory or special parameter passing registers as specified by the procedure calling convention. That is, there are no virtual registers that are live into the body of the function being allocated. Given that regions are an arbitrary subgraph of the program control, there may be any number of virtual registers live into and out of the region. These are commonly referred to as outward-exposed virtual registers.

During the allocation of a function, the register allocator essentially has two options for each virtual register, it may assign the virtual register to a physical register or it may spill the virtual register and assign it a stack location. During the allocation of a region, the register allocator has an additional option, it may ignore the virtual register. As an example, consider the regions shown in Figure 6.2. There are four outward-exposed virtual registers in region **A**: **vr1**, **vr2**, **vr3**, and **vr4**. All virtual registers are referenced within the region except **vr1**. By Equation (6.1), an unreferenced virtual register, such as **vr1**, has a priority of zero. Since such lifetimes are unimportant to the current region, the register allocator need not make a decision regarding **vr1** until a later region is allocated which actually references this virtual register. Because of this capability, the register allocator may experience lower register pressure within the region than would be seen in the same area of the function during function-based allocation.

In addition, the register allocator now has the option of taking a different action on a virtual register in two different regions, effectively achieving natural live range splitting. For example, in Figure 6.2 the register allocator may chose to allocate virtual register **vr3** to a physical register in region **A**, yet spill the same virtual register in region **B**. However, the register allocator must

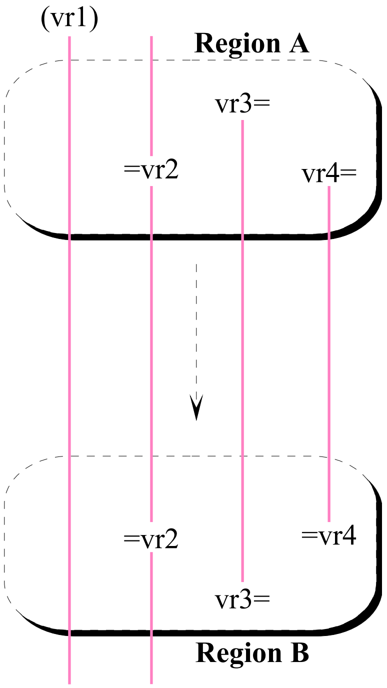


Figure 6.2 Regions containing outward-exposed virtual registers.

ensure that the necessary compensation code is inserted to correct for the differing register assignments between regions. Briggs, Cooper and Torczon [70] have proposed several methods of applying aggressive live range splitting to achieve this effect during the global allocation of a function with limited success.

While applying global register allocation within a region-based framework, the register allocator must be provided with sufficient information to ensure a correct allocation within the region and that the separately allocated regions can be reconciled. This information has to be maintained such that allocation of later regions is subject to the constraints from previously allocated regions. Also, there is information that can be provided to the register allocator to improve the quality of the allocation within the reduced scope resulting from the region partition. The remainder of this section will present this required and beneficial information and discuss how this information is used by the register allocator.

6.2.1 Dataflow information

As with the global optimizer, the register allocator requires live-variable information at the region exit points so that live ranges for virtual registers outward-exposed through region exits are computed correctly. Recall that the compilation manager has several options regarding the quality of the live-variable information provided. Assuming that all variables are live-out can be detrimental to the quality of the register allocation. For the remainder of this discussion it will be assumed that the live-variable information is being locally maintained by the compilation manager so that it is relatively accurate. Also it should be noted that, as implemented, the register allocator does not introduce any compensation code outside the region so the live-variable conditions at region boundaries do not change.

6.2.2 Register bindings

Since the register allocator is allocating the regions separately, it is quite possible that the same virtual register can be allocated to a different physical register in different regions. The compilation manager provides the register allocator with the register binding conditions for each outward-exposed virtual register at each region entry and exit point. In order to do this, the compilation manager simply maintains a table of the bindings of all outward-exposed variables at the entry and exit points of each register allocated region. The information is provided to the register allocator along with the region in the form of an auxiliary data structure.

Figure 6.3 contains two regions **A** and **B**. Region **A** has been allocated and has four outward-exposed virtual registers that are also live-in at the entry point of region **B**. During allocation

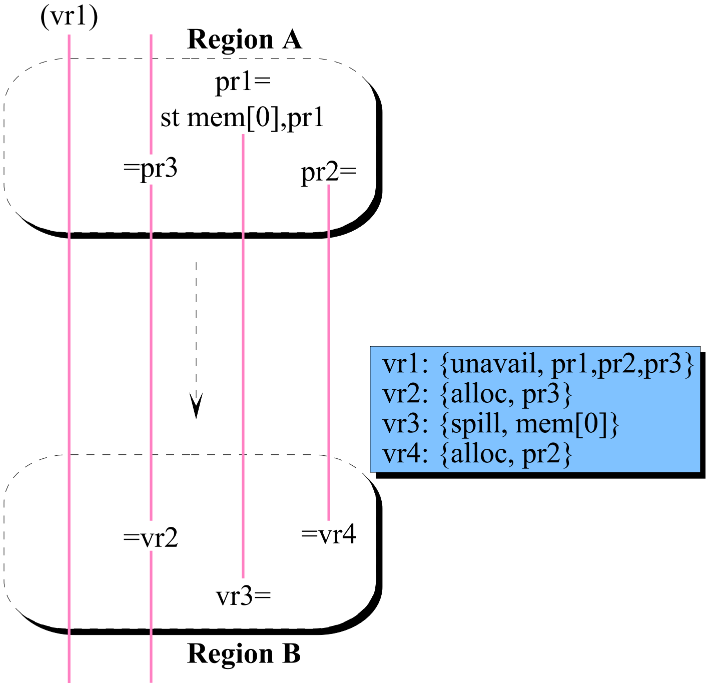


Figure 6.3 Outward-exposed virtual register boundary conditions provided to region B after register allocation of region A.

of region **B**, the register allocator is provided with the allocation results of all four virtual registers. In this case, register binding information is provided for **vr2** and **vr4** indicating that they were allocated to physical registers **pr3** and **pr2**, respectively. The entries for **vr1** and **vr3** will be discussed in later sections. This information has two uses with the region-based register allocator. First, this information is used by the register to insert the appropriate compensation code within the region, in the event that it does not make the same allocation decisions for outward-exposed virtual registers in different regions. Second, the register allocator may use the register binding information at region boundaries as a hint to guide the register selection process in order to minimize the amount of compensation code that is required. In fact, register bindings may stipulate to the register allocator the register that must be used for a virtual register used within this region (see Section 6.3).

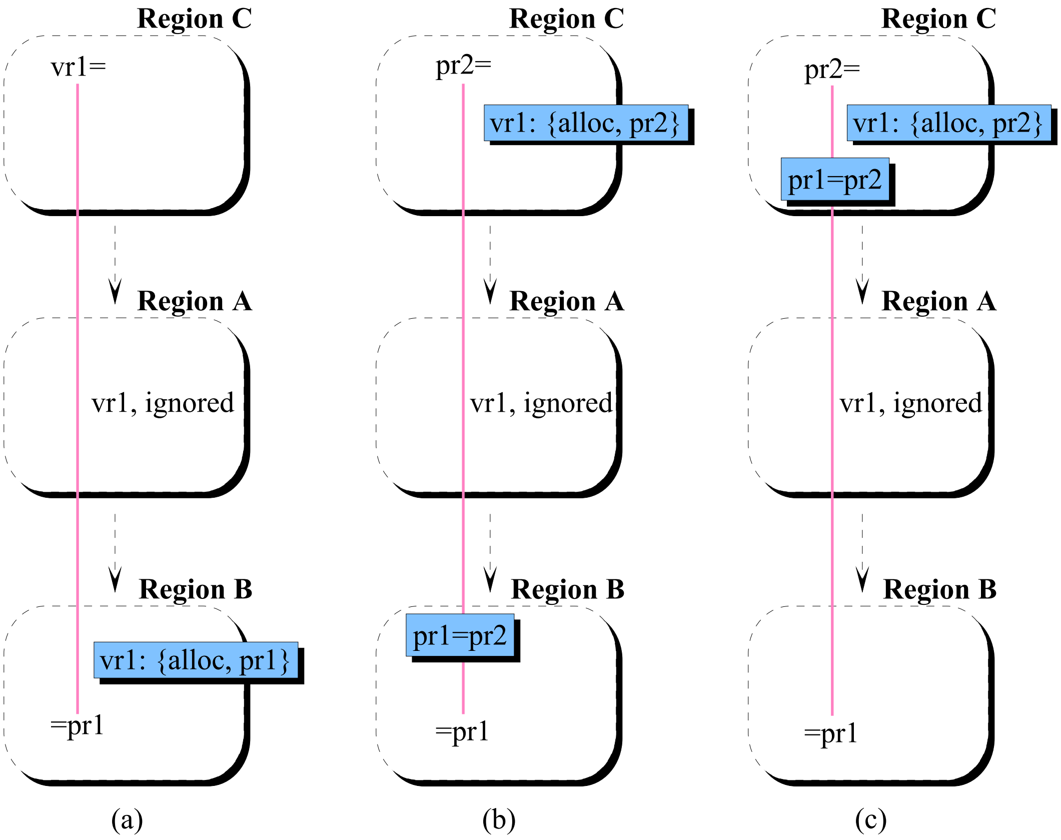


Figure 6.4 Ignoring virtual registers: (a) vr1 ignored in region A and assigned to pr1 in region B, (b) vr1 assigned to pr2 in region B with copy in region B, (c) copy in region C.

Allowing the register allocator to ignore a virtual register during the allocation of a region poses an interesting problem for the compilation manager, which must provide register binding information at region entry and exit points. Once an ignored virtual register has been bound to a physical register, that information must be propagated to subsequent regions. Consider the following example shown in Figure 6.4. If virtual register **vr1** is ignored in region **A** and later allocated to physical register **pr1** in region **B**, as shown in Figure 6.4(a), care must be taken during the allocation of region **C**. If the register allocator assigns the **vr1** to **pr1**, there is no problem. If the register allocator assigns the virtual register to **pr2**, then some correction code is required. This correction code could be inserted within region **B** assuming **vr1** will reside in

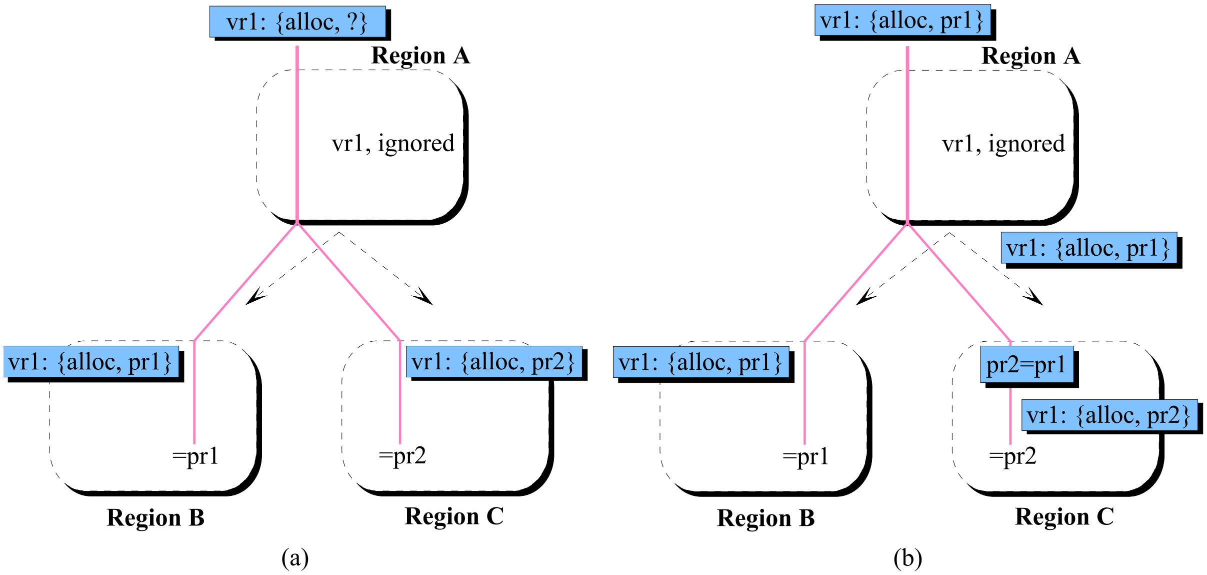


Figure 6.5 Ignoring virtual registers: (a) unknown boundary condition at entry to region A, (b) boundary condition set by allocation in region B.

pr2 in region **A**, as shown in Figure 6.4(b), or the correction code could be placed in region **C** assuming that **vr1** will reside in **pr1** in region **A**, as shown in Figure 6.4(c).

Figure 6.5 contains a slightly more complicated example. Consider the case in which the register allocator ignores virtual register **vr1** in region **A** and allocates it to **pr1** in region **B**, as indicated in Figure 6.5(a). If the register allocator assigns **vr1** to **pr2** in region **C** a question arises. What physical register contains **vr1** in region **A**? The register allocator must be provided the location of **vr1** in region **A** while region **C** and any other regions adjacent to **A** are being allocated. Given that regions are being compiled in order of importance, the compilation manager applies the following constraint. The first region to which a virtual register is bound, to either a physical register or a stack location, determines the location of that virtual register in all regions where it was ignored. Under this assumption, the register binding conditions for ignored virtual registers are well-defined. Thus, during the allocation of region

C in Figure 6.5(b) the compilation manager provides binding information telling the register allocator that **vr1** resides in **pr1** at the entry point to region **C**. If the register allocator does not allocate **vr1** to **pr1**, then a copy operation is required. From the literature it is not clear what assumptions were made within the Multiflow compiler to handle this situation [8], [69].

In a sense, the register allocator is not aware that a virtual register was ignored in an adjacent region. The register allocator is simply told which physical register a previously ignored virtual register resides in upon entry to the region or must reside in upon exit from the region. This also implies that an outward-exposed virtual register that has been bound to a physical register in an adjacent region **cannot** be ignored during the allocation of that region. The register allocator must ensure that the virtual register resides in the appropriate physical register at the region boundary points.

Value-location mappings

As previously mentioned, the Multiflow compiler conveyed register binding information by inserting explicit *Value-Location Mappings*(VLMs) into the control flow graph. These mappings are treated exactly like other operations during the scheduling and register allocation of a trace. Figure 6.6(a) shows the VLMs resulting from separate allocation of regions **A** and **B** from Figure 6.2. The VLMs indicate that **vr1** was ignored, or its binding is delayed, **vr2** was allocated to **pr3**, and **vr4** was allocated to **pr2** in region **A**. In region **B**, **vr1** was also ignored, **vr2** was allocated to **pr4**, and **vr4** was allocated to **pr5**. To reconcile the regions, a new trace region, **C**, is formed containing the VLMs. This region is then scheduled as any other region, resulting in the code shown in Figure 6.6(b). Note that copy operations have been inserted to compensate for the differing register bindings for **vr2** and **vr4**. The approach in this thesis

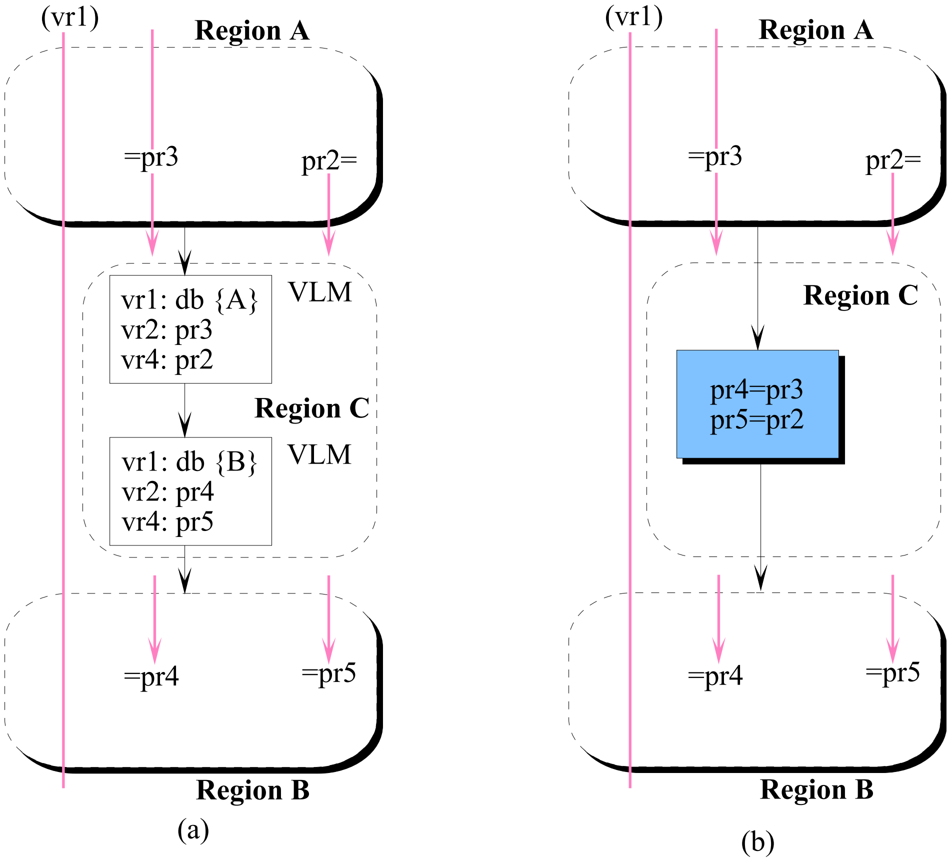


Figure 6.6 Value-location mappings: (a) VLMs after scheduling of regions A and B, (b) after scheduling of new region C.

differs in that the register allocator is responsible for inserting compensation directly into the current region after allocation is complete. This allows the compensation code to be scheduled in with the region entry or exit code and prevents mixing of register allocated and unregister allocated operations during the optimization of subsequent regions. Compensation code will be discussed in more detail in Section 6.3.

6.2.3 Unavailable registers

Delaying the assignment of a virtual register also imposes constraints upon the physical register that the virtual register may be assigned to during allocation of subsequent regions.

Consider **vr1** in Figure 6.3. There are no references of **vr1** within region **A** so it was ignored during the allocation of that region. However, since **vr1** is used within region **B**, the register allocator is required to either allocate **vr1** or spill it. To ensure a correct allocation, the register allocator cannot use any physical registers assigned to virtual registers in region **A** that interfere with **vr1**. Thus, as indicated in Figure 6.3, the compilation manager provides the register allocator with a set of physical registers that cannot be used for this virtual register. In this case, they are **pr1**, **pr2**, and **pr3**.

Under the assumption that the compiler cannot place compensation code within a compiled region, there is an additional constraint. If the live range of an ignored virtual register contains a subroutine call within the region where it was ignored, the register allocator cannot allocate that virtual register to a caller-saved register. Doing so would require the compiler to insert caller-save code around the subroutine calls in the regions where the virtual register was ignored. As each region is register allocated, the compilation manager maintains a set of unavailable registers for each virtual register that is ignored within a region and provides this set to each region where a previously ignored virtual register is outward-exposed.

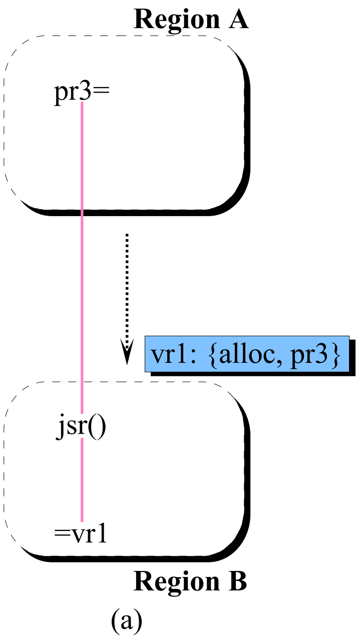
During graph coloring, the register allocator examines the interference graph to construct a set of physical registers that are already in use by interfering virtual registers and are, therefore, unavailable to the current virtual register. In addition, if the current virtual register is outward-exposed, but not yet assigned to a physical register in an adjacent region, the set of globally unavailable registers provided by the compilation manager is merged in with the results of the interference graph search. The register allocator then attempts to find a physical register that is not located within this set of unavailable registers for the current virtual register.

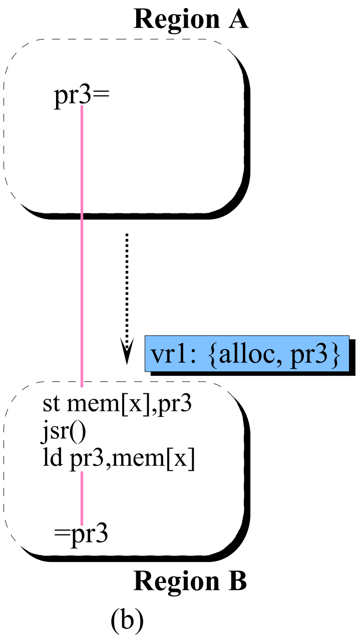
6.2.4 Stack locations

During the register allocation of each region, some virtual register will invariably be spilled and assigned a stack location. The chosen stack location must be provided to all subsequent regions where that same virtual register is outward-exposed. Again, the compilation manager maintains a table of the stack locations assigned to outward-exposed virtual registers that were spilled within a region. As shown in Figure 6.3, **vr3** was spilled during the allocation of region **A**. Thus the compilation manager provides information to the register allocator indicating that **vr3** is located in memory location **mem[0]** at the entry point to region **B**. Requiring that all regions use the same stack location for a particular virtual register ensures that all regions needing to access memory to retrieve this virtual register will access the correct value. This eliminates the need for compensation code between regions to relocate virtual registers on the stack.

6.2.5 Beneficial global information

Recall that the caller-saved/callee-saved benefit functions, Equations (6.3) and (6.4), assume that the register allocator has visibility to the entire live range of a virtual register. This is not the case during region-based register allocation. Consider the example in Figure 6.7(a) containing two regions **A** and **B** and an outward-exposed virtual register **vr1**. During allocation of region **A**, the register allocator selects a caller-saved register, say **pr3**, since the portion of the live range within the region contains no subroutine calls. Within region **B**, the live range of **vr1** contains a subroutine call. If the register allocator were to use the same register, **pr3**, within region **B**, load and store operations must be inserted around the subroutine call, as shown in Figure 6.7(b). If the register allocator were to select a callee-saved register, say **pr15**, then a





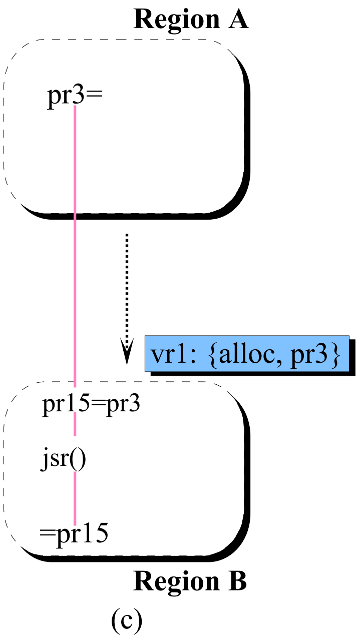


Figure 6.7 Subroutine calls: (a) caller-saved register selected in region A, (b) use of caller-saved register in region B requires spill code, (c) use of callee-saved register in region B requires copy.

copy operation would be required at the entry point of region **B** to correct for the different allocation, as shown in Figure 6.7(c).

By providing the register allocator with an estimate of the number of subroutine calls contained within the global live range of a virtual register, a callee-saved register may be selected instead, requiring no correction code if the same register is used in both regions **A** and **B**. This information is readily available to the compilation manager. After the initial pass of global live-variable information, the number of subroutine calls within a lifetime can be determined by examining the live-in sets of the basic blocks that contain subroutine calls. The compilation manager can then provide these global subroutine call counts for each outward-exposed virtual register of a region. In addition, it may be beneficial to provide the function execution weight for the same reason.

Both are provided to the IMPACT region-based register allocator by the compilation manager and used to determine the caller-saved and callee-saved benefits of outward-exposed virtual registers.

6.3 Reconciliation Code

After register allocation is complete, the register allocator may have to insert reconciliation code between the current region and previously allocated regions to compensate for the different allocation in each region. Since reconciliation code is only required between adjacent, allocated regions and those previously allocated regions are of higher weight and are potentially more frequently executed, all reconciliation codes are absorbed within the current region being allocated. Not only does this tend to keep reconciliation code in the less frequently executed areas of the function, but under a vertical compilation model, there is no benefit to pushing this compensation code outside the current region, since all adjacent, register allocated regions are completely compiled.

The required reconciliation code is determined by examining the register binding information at the region entry and exit points as provided by the compilation manager and comparing it to the allocation result in the current region. In all, there are five possible scenarios. Figure 6.8 shows the two most desirable situations. In these cases, the register allocator made the same decision in both regions for an outward-exposed virtual register, i.e. either the virtual register was allocated to the same register in both regions or it was spilled to the same memory location in both regions. In either case, no code compensation code is required. Recall, from Section 6.2.4 that the register allocator is not allowed to spill a virtual register to two different stack locations.

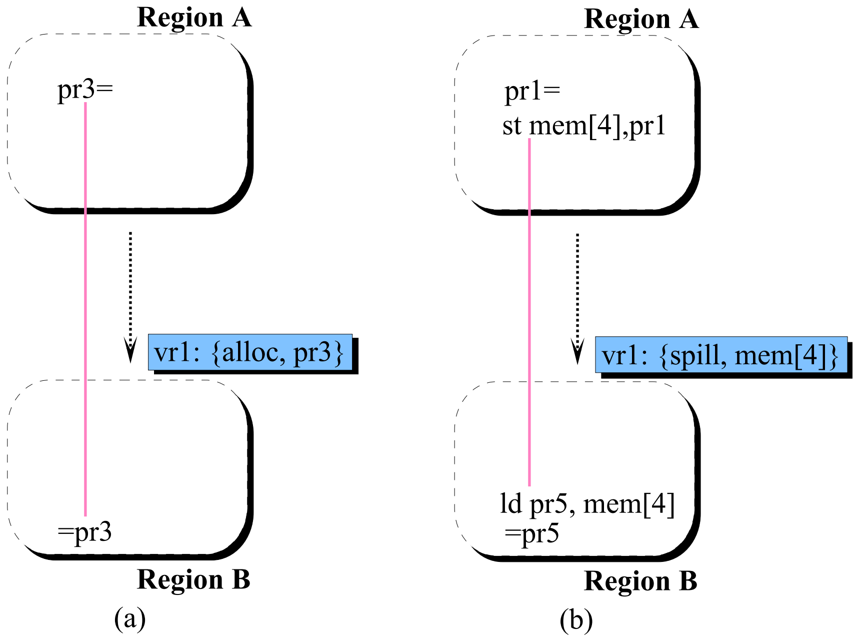


Figure 6.8 Region reconciliation: no code required.

Figure 6.9 shows the two cases in which a virtual register is spilled in one region and allocated in the other. In Figure 6.9(a) virtual register **vr1** is allocated to **pr3** in region **A**, but is spilled in the current region. Since all references of **vr1** in the current region, **B**, will be accessing its assigned stack location, a store operation is required to place the contents of **pr1** on the stack upon entry to the region. Figure 6.9(b) shows the opposite case. In region **A**, **vr1** was spilled; however, it is allocated to physical register **pr3** in the current region. In this instance a load operation is required to place the proper value into **pr3** upon entry to the current region.

Finally, Figure 6.10 shows the case in which a virtual register is allocated to two different registers in different regions. In region **A**, **vr1** is allocated to physical register **pr3**, but in the current region it is allocated to **pr10**. This requires a copy operation at the entry point of the current region to move the contents of **pr3** into **pr10**. If the register binding information is not taken into account during the allocation of the current region, separate allocation can quickly

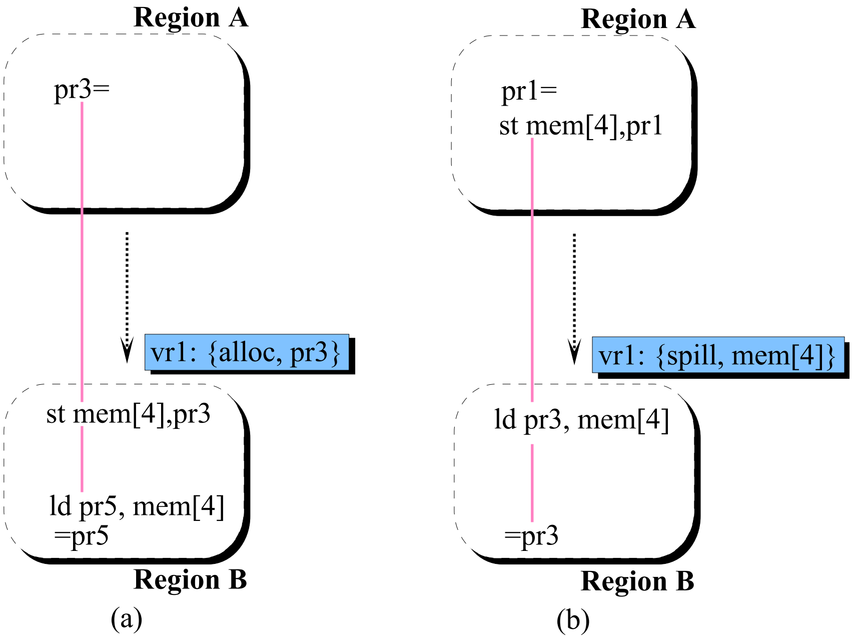


Figure 6.9 Region reconciliation: (a) store required in current region, (b) load required in current region.

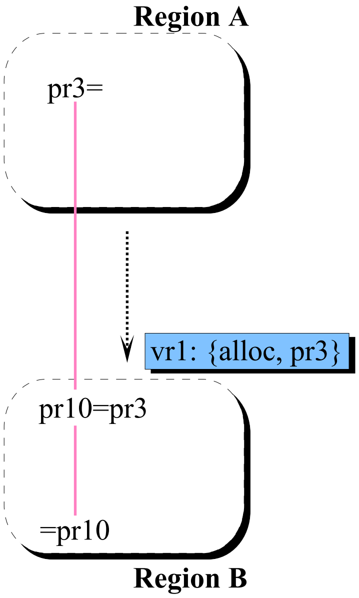


Figure 6.10 Region reconciliation: copy operation required current region.

produce an explosion of copy operations at region boundaries. To avoid an excessive amount of copy operations, the IMPACT region-based register allocator first attempts to assign an outward-exposed virtual register to the same register used by an adjacent region. If this cannot be done, the virtual register will be spilled. This approach has two effects. First, reconciliation code is limited to the first four cases shown in Figures 6.8 and 6.9; no copy operations are required. Second, this will restrict the register allocator and result in additional spill code. However, as shown in Section 6.4 it does not adversely affect output code performance.

6.4 Allocation Performance

This section will evaluate region-based register allocation in terms of register allocation quality, register allocation time, and memory usage. Function-based register allocation and region-based register allocation are performed on the same superblock ILP optimized code, for each of the 11 benchmarks at the desired inlining level. Superblock ILP optimized code is used to provide a register allocation environment with higher register pressure as is typically found after aggressive ILP optimization and scheduling. The assumed processor model is an identical four-issue processor with 64 integer and 64 double-precision registers rather than the previously assumed infinite number of registers.

6.4.1 Allocation quality

There are two ways to compare the quality of register allocation. The first is to compare the amount of dynamic spill operations introduced; the second is to compare the additional execution cycles added as a result of register allocation. Figure 6.11 compares the dynamic percentage of additional load and store operations introduced by the function-based and region-

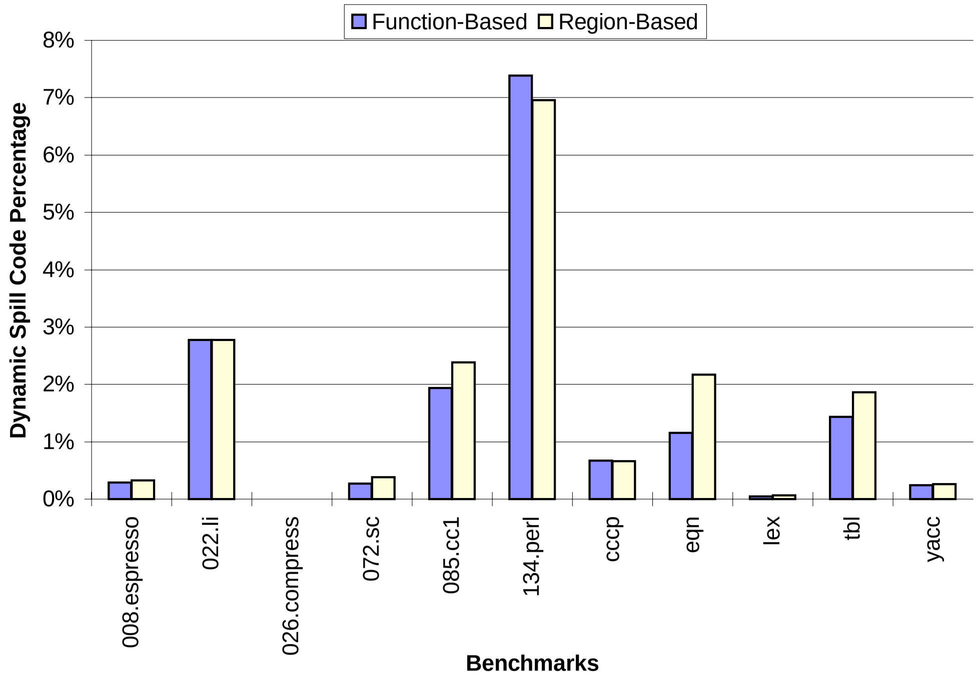


Figure 6.11 Comparison of function-based and region-based spill code for superblock ILP optimized code on a four-issue processor.

based register allocators. On average, 1.5% of the dynamic operations executed after function-based register allocation are memory access operations introduced by the register allocator. After region-based register allocation, this number increases slightly to 1.6%. This is not unexpected due to the reduced register allocation scope within the region-based compiler.

There are several factors that may increase the amount of spill code introduced by region-based register allocation. The region partitioning of the function will force the register allocator to make local decisions regarding the selection of caller-saved vs. callee-saved registers. In addition, the requirement that the register allocator use the same physical register for outward-exposed virtual registers may cause unnecessary load and store operations rather than copy operations. Providing the register allocator with a hint of the global subroutine call counts for a virtual register allows the register allocator to compensate very effectively for the reduction in scope for most of the benchmark programs. In spite of this, the noticeable increase in the number of spill code operations in the benchmarks **085.cc1**, **eqn**, and **tbl** is due to caller-saved load and store operations around subroutine calls. For these benchmarks, the global subroutine call estimates are invalidated by later transformation, namely, code scheduling. This problem does not occur during function-based register allocation since all code scheduling is completed prior to performing register allocation.

Consider the following example. If the register allocator is told that a virtual register **vr1** in region **A** contains no subroutine calls, that virtual register will most likely be allocated to a caller-saved physical register. However, during the compilation of region **B** an operation defining **vr1** is moved above a subroutine call. During register allocation of region **B**, the register allocator may prefer the use of a callee-saved register because the live range of **vr1** now contains a subroutine call contrary to the information provided by the compilation manager. The register allocator has two choices. First, use the caller-saved register selected by region **A** or spill the virtual register. In either case, some spill code is introduced. This is the source of the additional caller-saved spill code in the mentioned three benchmarks. In general, this is the only situation in which the restriction that the register allocator use the same physical register in all regions for an outward-exposed virtual register results in a significant amount of spill code.

Several of the benchmarks **022.li**, **134.perl**, and **cccp** experience a decrease in the amount of dynamic spill code operations, because of the natural live range splitting that the region partitioning provides. The most noticeable difference is in the benchmark **134.perl**, where the region-based register allocator was able to use three fewer callee-saved registers during the allocation of a frequently called, self-recursive function. This phenomenon can be illustrated

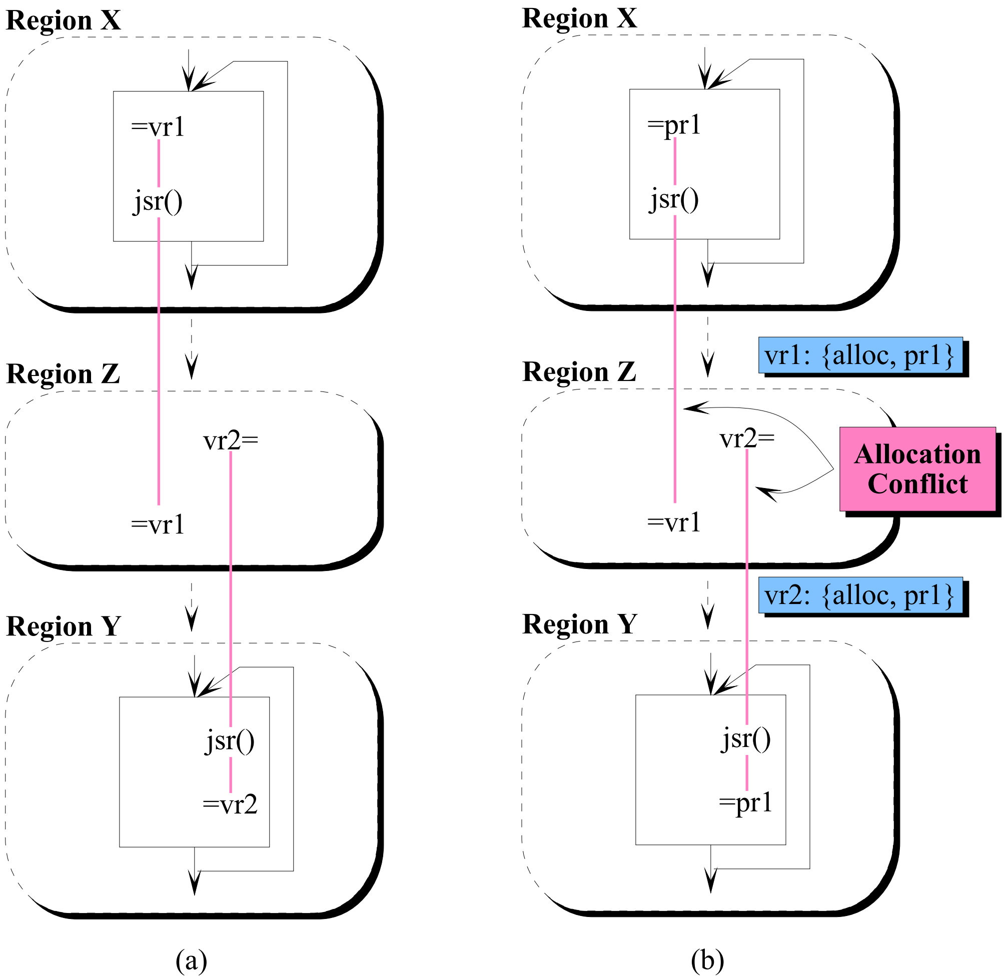


Figure 6.12 Natural live range splitting: (a) vr1 and vr2 interfere globally, but not in regions X and Y, (b) vr1 and vr2 allocated to same physical register resulting in conflict in region Z.

with the example shown in Figure 6.12(a). The figure contains three regions **X**, **Y**, and **Z**. Region **X** contains a frequently iterated loop with a subroutine call within it and references a virtual register **vr1**. Region **Y** holds another frequently iterated loop containing a subroutine call that references a virtual register **vr2**. These two virtual registers interfere since they are both simultaneously live in region **Z**. A function-based register allocator is required to use two registers because of this interference and prefers callee-saved registers because of the frequent subroutine calls contained in the live ranges. During allocation of region **X** and **Y**, the register

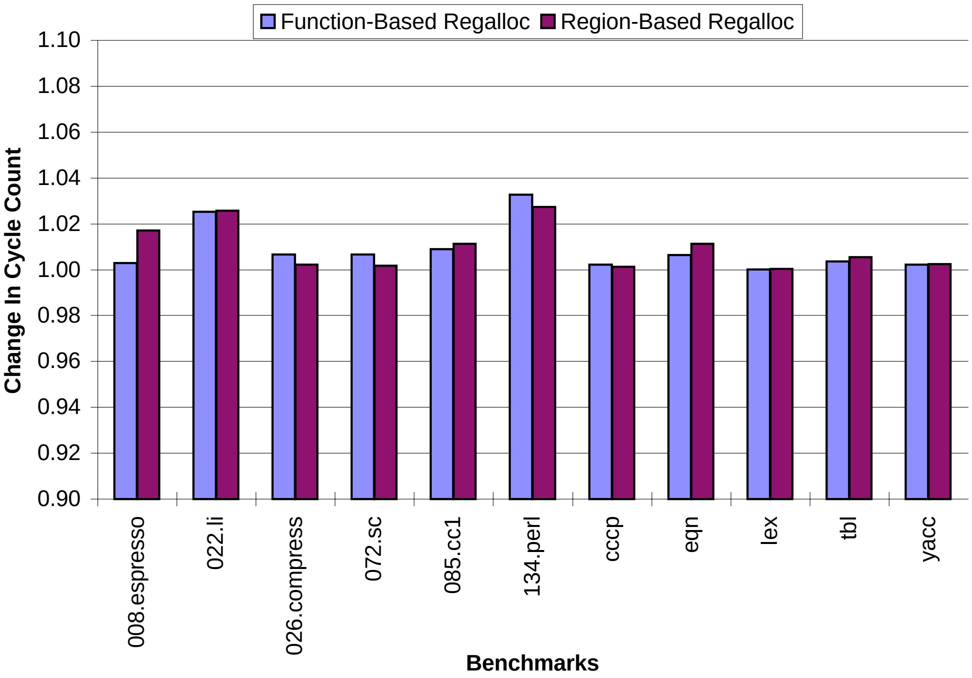


Figure 6.13 Comparison of function-based and region-based cycle count after register allocation of superblock ILP optimized code on a four-issue processor.

allocator may use the same callee-saved register for both **vr1** and **vr2** because these virtual registers do not interfere in these regions. In doing so, the register allocator will discover a conflict during the allocation of region **Z**, because the two interfering, outward-exposed virtual registers have been allocated to the same register, as shown in Figure 6.12(b). As a result, one of these two virtual registers will be spilled within region **Z**; however, for **134.perl** this spill code is much less frequently executed than the callee-saved spill code in the function prologue and epilogue. In this example, the region partition has reduced the register pressure from that seen during function-based allocation in two of the three most important regions.

The overall performance difference between the function-based and region-based codes on a four-issue processor is shown in Figure 6.13 by including the effect of the load and store operations introduced by register allocation. The two bars represent the change in cycle count after function-based and region-based register allocations. As expected, both register allocation environments result in a slightly increased cycle count due to the additional memory operations introduced by the register allocator. The benchmarks **134.perl** and **cccp** have slightly better cycle counts after region-based register allocation because of the smaller amount of spill code added during register allocation. However, the benchmarks **026.compress** and **072.sc** also show a lower cycle count even though the region-based register allocator required more spill code. In these benchmarks, the region-based allocation inserted fewer artificial dependences giving the postpass code scheduler more freedom to perform code motion than after function-based allocation. Also, as expected the larger amount of spill code inserted by the region-based register allocator in **008.espresso**, **eqn**, etc. results in a slightly larger cycle count. Overall, the region-based register allocator is able to produce comparable code in a considerably shorter amount of time.

6.4.2 Allocation time and memory usage

Register allocation time is dominated by the time required to build the virtual register interference graph. Given the $O(n^2)$ nature of interference graph construction, the reduction in problem size provided by the region partitioning has a significant impact on register allocation time. To quantify the effect of the region partitioning on register allocation time, the same register allocator is applied to both function-based and region-based compilation units. The amount of time required for region-based register allocation is the sum of the times required to allocate each region within the program. Figure 6.14 shows the speedup for region-based register allocation, i.e., the function-based allocation time over the region-based allocation time, for each benchmark. Region-based register allocation is on average three times faster than function-

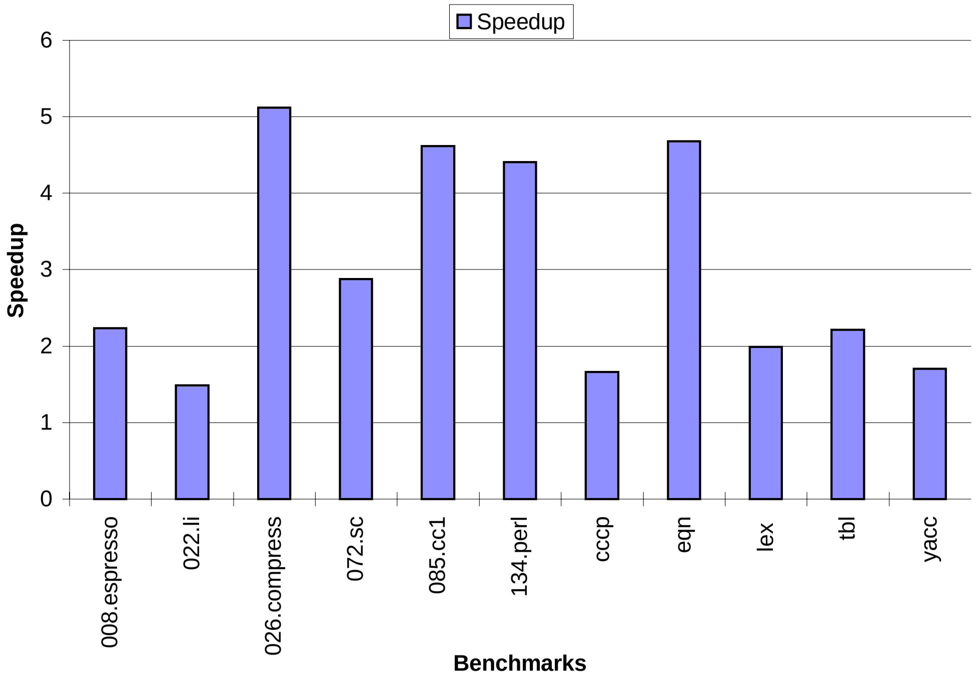


Figure 6.14 Speedup of region-based register allocation over function-based register allocation on superblock ILP optimized code.

based register allocation with most benchmarks achieving a speedup of at least a factor of two. The most significant speedups for the benchmarks such as **026.compress**, **085.cc1**, **134.perl**, and **eqn** result from the partitioning of large function bodies that contain a correspondingly large number of virtual registers. In these programs interference graph construction clearly dominates register allocation time. The benchmarks **022.li** and **cccp** do not benefit as much from region partitioning because the number of virtual registers within the functions of these programs is not high enough to cause interference graph construction to overshadow the linear nature of the rest of the register allocation process.

In addition to the significant register allocation time improvements, the region partition also provides memory usage improvements. Figure 6.15 shows the memory size reduction for region-based register allocation for each benchmark. This graph contains the ratio of the maximum

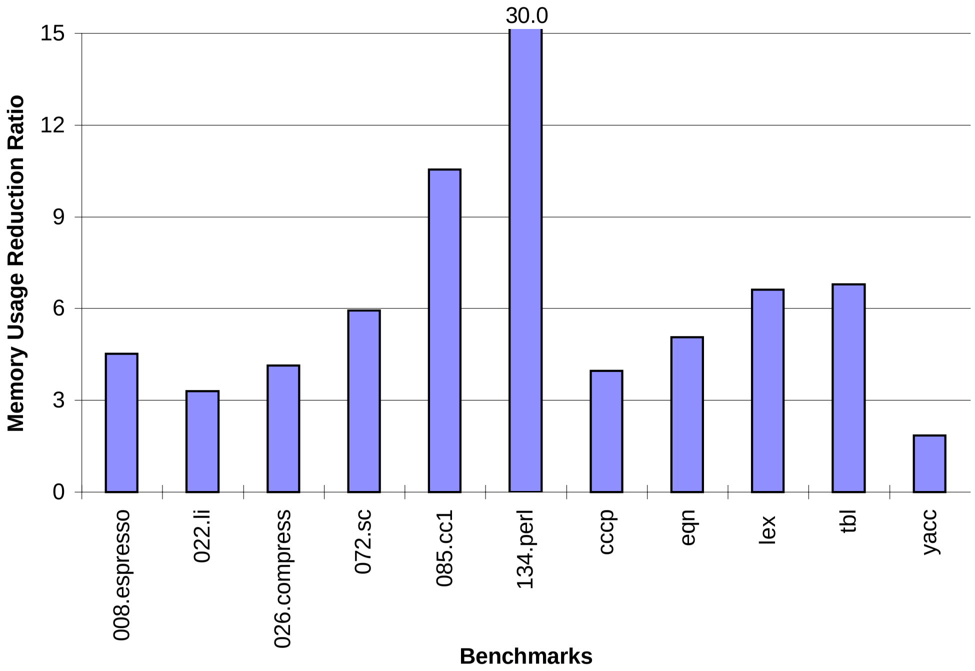


Figure 6.15 Memory usage reduction of region-based register allocation over function-based register allocation on superblock ILP optimized code.

amount of memory required for function-based register allocation to the maximum amount of memory required by region-based register allocation. Overall region-based register allocation required 7.5 times less memory than function-based register allocation with all benchmarks achieving a reduction of at least a factor of 3. Again, the extremely large functions within **134.perl** are responsible for the memory usage explosion within the function-based register allocator. The region-partition allows the region-based register allocator to reduce the amount of memory required by a factor of 30. The principal source of memory savings for register allocation is the fact that the register allocator does not have to construct the entire interference graph, which has a memory complexity of $O(n^2)$ in the number of virtual registers. Additional saving comes from the reduction in memory requirements for dataflow information and other information that must be collected for each virtual register within the compilation unit.

Table 6.1 Summary of function-based and region-based register allocation times and memory usages at desired inlining levels.

	Function-Based		Region-Based	
Benchmark	Time (s)	Memory (K)	Time (s)	Memory (K)
008.espresso	36.9	3748	16.5	828
022.li	18.1	1848	12.2	560
026.compress	6.3	2796	1.2	676
072.sc	48.6	9460	16.9	1592
085.cc1	613.1	46084	132.8	4368
134.perl	192.0	37528	43.5	1252
cccp	6.4	1172	3.8	296
eqn	38.5	11464	8.2	2264
lex	8.5	1800	4.3	272
tbl	28.7	7264	12.9	1068
yacc	9.4	836	5.5	452

Finally, Table 6.1 contains the data used to create the the allocation time and memory reduction graphs in this section. The register allocation times, in seconds, are shown for both function-based and region-based compilations on a HP 9000/735, 125MHz workstation. Note, the table contains no overhead for region-based register allocation. Maintenance of the boundary condition information required for region-based register allocation does not noticeably increase the overhead introduced by the compilation manager. The maximum amount of memory required during function-based and region-based register allocations is shown in increments of 1024 bytes(K).

CHAPTER 7

CONCLUSIONS

7.1 Summary

The traditional function-based approach to compilation is not suitable within an aggressive ILP compiler. The function-based partition of a program, albeit providing a convenient way to break up the task of compiling the program, presents the compiler designer with several problems. As shown in this dissertation, the function bodies present the compiler with an environment that hides valuable information and optimization opportunities. In addition, the use of aggressive transformation, such as inlining, to improve the situation must be carefully controlled to contain the increase in compile time and memory usage that results from assembling larger function bodies.

This dissertation has proposed region-based compilation, a technique to improve the feasibility of applying aggressive compilation techniques within a production quality ILP compiler while potentially improving the quality of the code generated by the compiler. Under this framework, the compiler is allowed to repartition the program into regions and compile each region as a separate compilation unit. This provides the compiler with several new capabilities. First, the compiler gains control over the compilation unit size, which improves the compilation time and memory usage behavior of the compiler, reducing the importance of algorithmic complexity of the applied transformations. This in turn makes the compiler easier to design. Second, the compiler can focus the application of aggressive transformations on important regions of the program and apply only necessary transformations to regions that are less important. Third, region-based compilation introduces vertical compilation within the body of a function, allowing any compensation code generated during the compilation of a region to benefit from optimization as part of subsequently formed regions. In order to study the implications of these capabilities within an ILP compiler, a region-based compiler has been implemented within the framework of the IMPACT compiler.

Application of classical optimization within the region framework requires the compiler to accurately maintain live variable information at region boundaries and to optionally maintain available-expression and available-definition information. Without available-expression and available-definition information, interregion global optimizations are lost; however, the region partition also tends to expose additional loop optimization opportunities. Essentially, without available-expression and definition information, region-based optimization trades global optimization efficiency for improved optimization within the region. Providing available-expression and available-definition information allows the optimizer to take advantage of interregion optimization opportunities. In both cases, the results indicate that region-based classical optimization produces better quality code than function-based optimization.

Application of global register allocation within the proposed region framework requires the compiler to maintain register binding as well as additional information to ensure correct reconciliation of separately register allocated regions. The region partition reduces the ability of the register allocator to make effective global decisions, such as selecting between caller-saved and callee-saved registers, resulting in slightly higher spill code than is introduced during function-based global register allocation. However, the natural live range splitting capability provided by the region partition does reduce register pressure in important regions allowing spill code to be placed in less frequently executed areas of the program. As the data indicate the result is a better quality allocation than that provided by function-based global register allocation in some instances. As with optimization, the reduced scope of a region provides a significant reduction in the amount of time and memory required to perform global register allocation.

This thesis shows that region-based classical optimization produces better quality code and region-based register allocation produces a comparable register allocation, both in much less time and using much less memory. The time and memory saved during classical optimization and register allocation will allow the designer of an ILP compiler to apply more aggressive ILP techniques than are feasible in a production compiler that assumes functions as the unit of compilation. Thus, a region-based compiler has the potential to produce superior quality code in a production environment where compilation time and memory requirements must be contained.

7.2 Future Work

Although the investigations in this dissertation provide a great deal of understanding of the behavior and potential of region-based compilation, there are still several open issues that can lead to promising areas for future research.

The investigation of classical optimization within a region context examined varying levels of dataflow information at region boundaries. Maintenance of available-expression and available-definition information was beneficial in several instances by allowing the region-based optimizer to perform critical interregion global optimizations. Availability information may be more crucial in a region-based optimizer that applies more aggressive transformations, such as partial redundancy elimination. The region-based compiler as implemented for this thesis maintained only live-variable information at region boundaries. The ideal region-based results where availability information is provided were generated by performing function-based optimization. An efficient method of providing and maintaining availability information to the global optimizer as well as the feasibility of interregion transformation within a vertical compilation model warrants further study.

In this dissertation, aggressive function inline expansion was used to assemble dynamically coupled functions into the same function body. Regions were then selected on the enlarged function bodies. Inline expansion of the entire function body introduces a great deal of dynamically unimportant code into the calling function, resulting in unnecessary code expansion. Rather than perform aggressive inlining a priori, the inline expansion decisions can be incorporated into the region formation process. If the compiler were allowed to cross function boundaries and grow regions interprocedurally, only the portion of the called function within the selected region has to be inline expanded into the call site. This reduces the code expansion that results from aggressive inline expansion prior to region formation. In this manner the region-based compilation framework may provide a directed method for performing partial inlining.

Recall from the discussion of function inline expansion in Chapter 3 that inline expansion does not expose an interprocedural call cycle to the compiler. Through application of aggressive inline expansion, a recursive call cycle will degenerate to a self-recursive function; however, the back edge of the cycle remains implicit in the subroutine call. Performing interprocedural region formation, followed by partial inlining as described above, will result in a self-recursive region. A technique that can transform such a region into an iterative cycle has enough merit to warrant further study. With this capability, the region-based compilation framework provides a uniform method for exposing all cycles to the compiler while attempting to minimize code expansion.

APPENDIX A

BENCHMARKS

All evaluations presented in this thesis use the set of eleven benchmarks shown in Table A.1. The benchmarks consist of five of the six programs from the SPEC CINT92 suite, and another benchmark from the SPEC CINT95 suite, **134.perl**. The remaining five benchmarks are common Unix utility programs. A short description of each benchmark is also presented in Table A.1. These benchmarks were chosen because of their control-intensive nature and traditional lack of exploitable ILP.

The IMPACT compiler makes extensive use of execution profile information during the compilation procedure. A description of the input files used to generate the profile information is presented in Table A.2. For the benchmarks in which inputs were readily available, a set of 20

Table A.1 Benchmark set.

Benchmark	Description
008.espresso	truth table minimization (SPEC CINT92)
022.li	lisp interpreter (SPEC CINT92)
026.compress	file compression (SPEC CINT92)
072.sc	spreadsheet (SPEC CINT92)
085.cc1	GNU C compiler (SPEC CINT92)
134.perl	shell interpreter (SPEC CINT95)
cccp	GNU C preprocessor, version 1.35
eqn	format math formulas for troff
lex	lexical analyzer generator
tbl	format table for troff
yacc	parser generator

Table A.2 Benchmark inputs used for profiling.

Benchmark	Description of Profiling Inputs
008.espresso	20 truth tables
022.li	4 lisp files (8-queens, 3 gabriel benchmarks)
026.compress	20 files of varying size
072.sc	3 spread sheets (3 SPEC reference inputs)
085.cc1	20 C files of varying size
134.perl	20 perl scripts of varying size
cccp	20 C files of varying size
eqn	20 technical papers containing equations
lex	5 lexers for C, Lisp, Pascal, awk and pic
tbl	20 technical papers containing tables
yacc	10 grammars

random files were selected to provide a wide range of training data. For the other benchmarks, as many inputs as typically could be obtained were used for profiling.

The input file used for each benchmark to collect performance data is presented in Table A.3. Overall, a large amount of care was taken to select a suitable input for measurement purposes. Also, the measured input was chosen to be different than all of the inputs on which the program was profiled to provide a more realistic evaluation. However, due to lack of available inputs, **072.sc**, was profiled and measured on a common input. For the SPEC benchmarks, one of the SPEC reference inputs was chosen. The one exception was for **022.li**, which used a scaled-down version of the SPEC reference input to reduce the simulation time.

Table A.3 Benchmark input used for measurements.

Benchmark	Description of Measured Input
008.espresso	one of the SPEC reference inputs (bca.in)
022.li	7-queens
026.compress	SPEC reference input
072.sc	one of the SPEC reference inputs (loada2)
085.cc1	one of the SPEC reference inputs (insn-recog)
134.perl	one of the SPEC referene inputs (jumble)
cccp	the file cccp.c from the GNU C Compiler Version 1.35
eqn	one large technical paper
lex	IMPACT compiler's C lexer
tbl	one large technical paper
yacc	the grammar in c-parse.y from the GNU C Compiler Version 1.35

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

[6] 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.

[7] D. Callahan and B. Koblenz, "Register allocation via hierarchical graph coloring," in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 192–203, June 1991.

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

[9] G. E. Haab, "Data dependence analysis for Fortran programs in the IMPACT compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[10] D. M. Gallagher, "Memory disambiguation to facilitate instruction-level parallelism compilation," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[11] K. Subramanian, "Loop transformations for parallel compilers," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[12] S. Anik, "Architectural and software support for executing numerical applications on high performance computers," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[13] Y. Yamada, "Data relocation and prefetching for programs with large data sets," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[14] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 242–251, May 1989.

[15] P. P. Chang, "Compiler support for multiple instruction issue architectures," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[16] 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*, vol. 22, pp. 349–370, May 1992.

[17] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[18] R. A. Bringmann, "Compiler-controlled speculation," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[19] 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.

[20] S. A. Mahlke, "Exploiting instruction-level parallelism in the presence of conditional branches," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[21] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," *IEEE Transactions on Computers*, vol. 44, pp. 353–370, March 1995.

[22] N. J. Warter, "Modulo scheduling with isomorphic control transformations," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[23] D. M. Lavery and W. W. Hwu, "Unrolling-based optimizations for modulo scheduling," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 327–337, Nov. 1995.

[24] W. Y. Chen, "Data preload for superscalar and VLIW processors," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[25] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, pp. 183–193, October 1994.

[26] R. E. Hank, "Machine independent register allocation for the IMPACT-I C compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[27] J. C. Gyllenhaal, "A machine description language for compilation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[28] R. A. Bringmann, "Template for code generation development using the IMPACT-I C compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1992.

[29] W. Y. Chen, "An optimizing compiler code generator: A platform for RISC performance analysis," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[30] R. G. Ouellette, "Compiler support for SPARC architecture processors," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[31] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.

[32] R. S. Pressman, *Software Engineering: A Practitioner's Approach.* New York, NY: McGraw-Hill, 1988.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] G. Kildall, "A unified approach to global program optimization," in *1st ACM Symposium on the Principles of Programming Languages*, pp. 194–206, 1973.

[49] P. Anklam, D. Cutler, J. R. Heinen, and M. D. MacLaren, *Engineering a Compiler*. Bedford, MA: Digital Press, 1982.

[50] M. A. Auslander and M. E. Hopkins, "An overview of the pl.8 compiler," *ACM SIGPLAN Notices*, vol. 17, pp. 22–31, June 1982.

[51] F. C. Chow, "A portable machine-independent global optimizer - design and measurments," Ph.D. dissertation, Department of Electrical Engineering, Stanford University, Sanford, CA., 1983.

[52] E. Morel and C. Renviose, "Global optimization by suppression of partial redundancies," *Communications of the ACM*, pp. 96–103, February 1979.

[53] K.-H. Drechsler and M. P. Stadel, "A solution to a problem with Morel and Revoise's 'Global optimization by suppression of partial redundancies'," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 635–640, April 1988.

[54] J. Knoop, O. Ruthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementaton*, June 1994.

[55] P. Briggs and K. D. Cooper, "Effective partial redundancy elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pp. 159–170, June 1994.

[56] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice and Experience*, vol. 21, pp. 1301–1321, December 1991.

[57] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, pp. 137–147, March 1976.

[58] K. Kennedy, "A comparision of two algorithms for global flow analysis," *SIAM Journal of Computing*, pp. 158–180, January 1976.

[59] M. S. Hecht, *Flow Analysis of Computer Programs.* New York: North-Holland, 1977.

[60] L. L. Pollock and M. L. Soffa, "An incremental version of iterative data flow analysis," *IEE Transactions on Software Engineering*, vol. 15, pp. 1537–1549, December 1989.

[61] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," in *Computer Languages*, vol. 6, pp. 47–57, January 1981.

[62] F. Chow and J. Hennessy, "Register allocation by priority-based coloring," in *Proceedings of the SIGPLAN 1984 Symposium on Compiler Construction*, pp. 222–232, June 1984.

[63] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon, "Coloring heuristics for register allocation," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, July 1989.

[64] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.*, pp. 40–52, Apr. 1991.

[65] P. Briggs, K. D. Cooper, and L. Torczon, "Rematerialization," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, June 1992.

[66] C. Norris and L. L. Pollock, "Register allocation over the program dependence graph," in *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 266–677, June 1994.

[67] J. Ferrante, M. Mace, and B. Simmons, "Generating sequential code from parallel code," in *International Conference on Supercomputing*, pp. 582–592, July 1988.

[68] 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.

[69] 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," *The Journal of Supercomputing*, vol. 7, pp. 51–142, January 1993.

[70] P. Briggs, K. D. Cooper, and L. Torczon, "Aggressive live range splitting," Tech. Rep., Rice University, Houston, TX, 1991.

VITA

Richard Eugene Hank was born in Peoria, Illinois, in 1968. He pursued his undergraduate studies at the University of Illinois in Urbana, Illinois, where he received the B.S. degree in Computer Engineering in 1990. After receiving the B.S. degree, he continued his studies in Electrical Engineering at the University of Illinois at Urbana, Illinois. In the fall of 1991 he joined the Center for Reliable and High-Performance Computing as a member of the IMPACT project directed by Professor Wen-mei Hwu. He completed his M.S. degree in Electrical Engineering in 1993. After completing his Ph.D. work, he will join Hewlett Packard's California Language Laboratory in Cupertino, California.