DYNAMIC CONTROL OF COMPILE TIME USING VERTICAL REGION-BASED COMPILATION

BY

JAYMIE LYNN BRAUN

B.S., University of Iowa, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Wen-Mei Hwu, for providing me with the resources, support, and guidance necessary to complete my master's thesis. He is an extremely caring teacher and a positive and motivating influence on the students he teaches and advises.

I would like to thank Richard Hank, upon whose doctoral work this thesis is built. He always had time to answer my questions and to talk about possible research directions.

I would also like to thank the members of the IMPACT research group for their assistance. This includes Dan Connors, Teresa Johnson, David August, Brian Deitrich, Ben Chung, and John Gyllenhaal. All of these people assisted me whenever I need help.

Finally, I would like to thank my family and friends, without whom I could never have made it to the place I am today. Thank you to my mother and father for always letting me make my own decisions, and for supporting me and believing in me no matter what those decisions were. Dan Connors and Robb Shimon were always there to talk with or just to listen during my time at Illinois, and both are wonderful friends. Finally, thank you to Tom, for caring about me, hanging on through the good and bad, loving me unconditionally, and providing me with the balance I so desperately need in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

In an effort to extract higher levels of instruction-level parallelism (ILP) from programs and meet the needs of today's wide-issue machines, compilers are employing aggressive global optimizations and scheduling techniques. The application of these transformations to a program is a time-intensive task. Compilers also often employ aggressive procedure inlining in order to obtain a more global view of a program and expose cyclic code. This process increases the size of the functions in a program. In a traditional compiler, this may cause the optimization and scheduling of a function to become intractable as the memory and time compilation requirements increase.

A viable alternative to the traditional compilation method is region-based compilation. In region-based compilation, the basic compilation unit is changed from the function, which is defined by the software developer, to a section of the program's control flow graph chosen by the compiler. This section of the graph is referred to as a region. Allowing the compiler to repartition the program into regions provides it with the freedom to control the content of the collection of basic blocks which it will compile together. The compiler can choose to exclude certain hazards from a region, such as a subroutine call, or infrequently executed basic blocks which may limit optimization. Region-based compilation also allows dynamically related portions of the control flow graph to be compiled together. Finally, region-based compilation allows the compiler to control the size of the region it selects, and therefore the time and memory requirements of the algorithm used to apply aggressive optimizations to the region.

This thesis utilizes vertical region-based compilation in a method for dynamically controlling the compile time of a program. Under this model, the compiler attempts to extract a high level of ILP from a program by taking a completely global view of its structure and then selecting the most important portions of the program to be compiled first. As it compiles the regions in order of descending profile frequency, it reduces the aggressiveness of the optimizations applied to the regions in order to meet a user-specified target compilation time. The goal of the technique is to extract nearly all of the performance from the program, which is possible when aggressive optimizations are applied over its entire CFG, while controlling the amount of time spent compiling the program. Under this model, the greatest percentage of time is spent compiling the most important regions, as indicated by the profile information, and very little time is spent compiling the remainder of the program. With this approach, we hope to find the point where increasing the compile time further provides little or no performance benefit to the compiled program. In taking a dynamic approach to this, the compiler is able to adjust the level of optimization applied to each region as it compiles the program and gauges the amount of time required to compile the more important regions.

Chapter 2 presents region-based compilation and some of its benefits, and also gives an overview of the Illinois Microarchitecture Project Utilizing Advanced Compiler Technology (IMPACT) compiler which is used through out this thesis. Chapter 3 illustrates the dynamic-based compilation procedure developed for this thesis and presents the different compilation options which can be exercised under this model. Chapter 4 illustrates the results of using compile-time control and presents a viable setting for the module. Finally, Chapter 5 contains a conclusion and proposes directions for future work.

2. BACKGROUND

In a traditional compiler, the unit of compilation is the function. This unit is a direct reflection of the way in which the developer of the program decides to divide its functionality in order to facilitate the reuse and readability of code. The function provides a convenient way to partition a program for compilation because it clearly establishes self-contained segments with well defined calling conventions between them. These conventions allow the compiler to process each function separately without maintaining any state information between them. However, it does not necessarily follow that the partition chosen by the developer in the interest of code clarity is the best choice over which the compiler should apply optimizations, scheduling, and register allocation. Often, these divisions force the compiler to consider too much code at once, causing the aggressiveness of optimizations to be scaled back in the interest of time and memory conservation.

This chapter will provide background on a method of compilation in which the compiler is able to repartition the code within a program. These newly defined segments, called regions, replace the function as the primary unit of compilation. The framework of region-based compilation utilized in this thesis was developed by Richard Hank upon the IMPACT compiler platform. For a more comprehensive explanation of the process, the reader is referred to [1], [2]. Prior to the presentation of the region selection algorithm, a brief overview of the IMPACT compiler is presented to facilitate understanding of how region-based compilation works within the IMPACT framework.

2.1 Overview of the IMPACT Compiler

The IMPACT compiler is an retargetable optimizing C-compiler developed at the University of Illinois. A block diagram of the IMPACT compiler is presented in Figure 2.1. The compiler is divided primarily into two sections, distinguished by the form of intermediate representation (IR) used in each. The level of IR closest to the source code is called *Pcode*. It is a parallel C code representation with its loop constructs intact. The following functions are performed within *Pcode*: memory dependence analysis [3], [4], statement-level profiling and function inlining [5], [6], [7], [8], loop-level transformations [9], and memory system optimizations [10].

The lower level of IR in IMPACT is referred to as *Lcode*. This is a machine-independent instruction set implemented as a generalized register transfer language similar to most load/store architecture instruction sets. At this level, all machine independent optimizations [11] are applied. These include constant propagation, forward copy propagation, backward copy propagation, common subexpression elimination, redundant

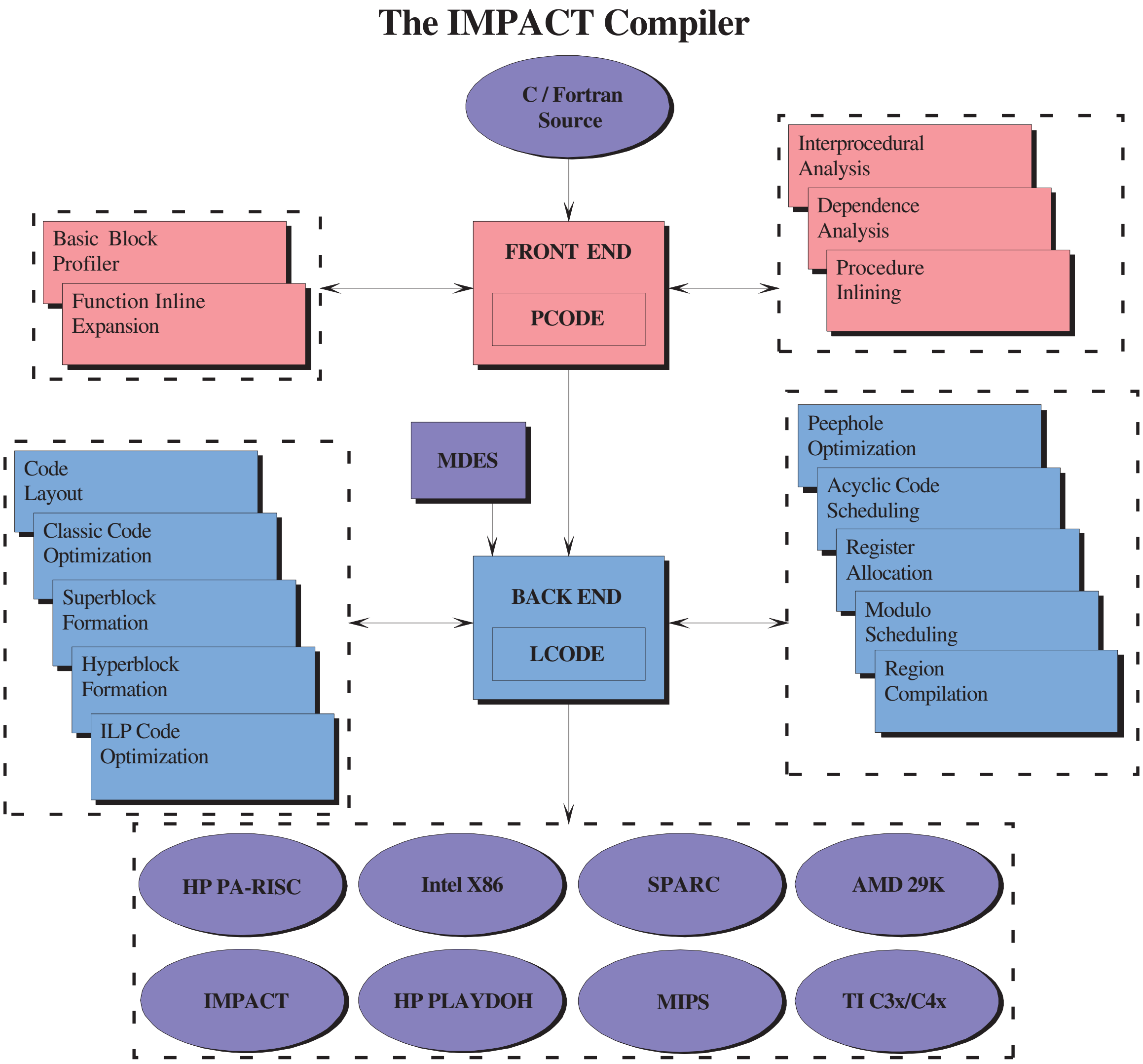

Figure 2.1: The IMPACT compiler.

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 reassociation. Additionally, advanced ILP compilation techniques, such as superblock formation and optimization [12] are performed on the *Lcode* representation of the program.

Region-based compilation, as utilized in this thesis, is also performed at the *Lcode* level. Prior to processing, each region is encapsulated in such a way as to allow the existing IMPACT transformations, originally designed to be applied to functions, to be applied directly to a region. A detailed explanation of region selection, extraction, optimization and reintegration is presented in Section 2.2.

Once the *Lcode* is optimized, it is translated into assembly language. IMPACT supports the generation of code for several architectures through distinct code generators. The most actively supported architectures are the Sun SPARC [13], the HP PA-RISC, and the Intel X86. There are also two experimental ILP architectures supported, IMPACT and HPL Playdoh [14]. 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 this thesis, all experiments utilize the IMPACT architecture.

The two most significant components of the code generators are the register allocator and the instruction scheduler. These modules are common to all of the code generators in IMPACT. Register allocation is performed using graph coloring [15], [2]. Several different code scheduling models exist, including acyclic global scheduling [16], [17], software pipelining using modulo scheduling [18], [19], and sentinel scheduling [20].

A detailed machine description database, *Mdes*, is referenced throughout the compilation process by various IMPACT modules [21]. This database contains information such as the number and type of functional units available, the size and width of register files, instruction latencies, instruction input/output constraints, addressing modes, and pipeline constraints. The information is used to guide optimization, scheduling, register allocation, and code generation.

2.2 Region-Based Compilation

In any compilation framework, there are three basic components. These include the compilation unit selection, transformation, and state maintenance. The maintenance of state insures that separately compiled portions of the program can be reconciled into a program which is functionally correct.

In a traditional compiler, the unit of compilation is the function. A predefined suite of phase-ordered transformations is applied to each function. The maintenance of state between the functions is trivial due to the well defined calling conventions between function boundaries.

In a region-based compiler, the selection and state maintenance components become more complex. A region is defined as an arbitrary subgraph of the global control flow graph chosen by the compiler. Under this definition, a function can be considered a region, while it does not necessarily follow that a region is a function. In choosing a region, the compiler is able to take into consideration and control the region size and characteristics. Allowing the compiler to limit the size of the compilation unit reduces the importance of the time and memory complexity of the transformations applied to a region. Under this framework, the compiler is often able to apply more aggressive transformations to the regions than would be possible in a traditional compiler where the compilation unit is larger.

Another benefit of region-based compilation is that the compiler is able to select compilation units which reflect the dynamic behavior of the program using profile information. Finally, the compiler is able to exclude basic blocks from a region which may contain hazards. A function call is such a hazard in that it may present an obstacle to aggressive optimization and scheduling due to the unknown memory footprint of the function call.

The problems of a function-based compiler are further aggravated in the presence of aggressive inlining. Profile-base inlining is often employed by a compiler to gain a broader global view of a program and to reduce the effects of interprocedural coupling [1], [2], [5], [6], [7], [8]. Inlining often exposes memory aliasing, optimization opportunities, or cyclic control flow structures which were hidden in the original program by procedure calls. The price of aggressive inlining, however, is an increase in the size of a program's functions. In a traditional, function-based compiler, this increase in function size may cause the time and memory requirements of aggressive transformations to become intractable as both these resource usages are related in a nonlinear fashion to the problem size. If this is the case, the benefit which was exposed by inlining can't be taken advantage of as intended [22], [23], [24].

In a region-based compiler, much of the adverse effect of inlining is alleviated by the compiler's ability to control the region size.

2.2.1 Region-based optimization

In a conventional compilation environment, each program function is completely processed before the compiler proceeds to the next function. A phase-ordered suite of optimizations, register allocation, and scheduling are all applied in a horizontal fashion, as illustrated in Figure 2.2. There is very little freedom to adjust the level of optimization applied to different parts of a function because distinct segments are difficult to identify and categorize.

Region-based compilation provides the compiler with more freedom. Compilation can proceed in a horizontal fashion, as in function-based compilation, illustrated in Figure 2.3, or a vertical component can be added to the compilation processes, as in Figure 2.4. In this model, the compiler is allowed to select a region, apply optimizations to it, allocate its registers, and schedule it before proceeding to the selection of the next region. The

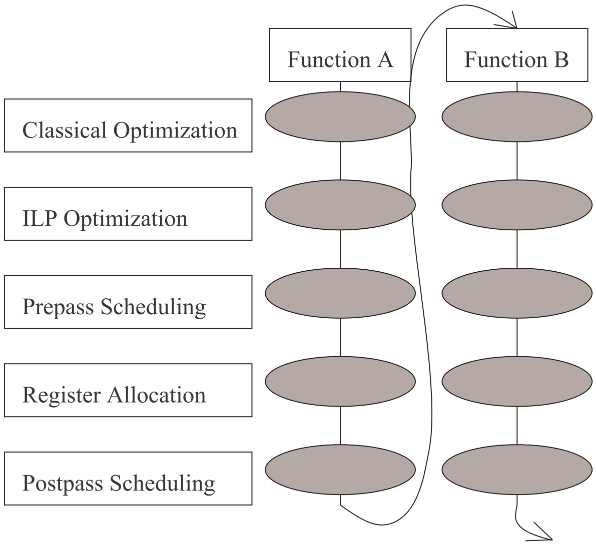

Figure 2.2: Horizontal function-based compilation of a function.

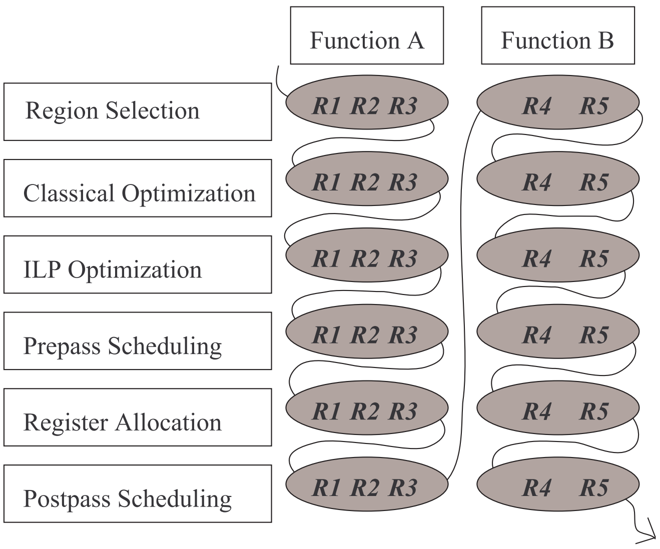

Figure 2.3: Horizontal region-based compilation of a function.

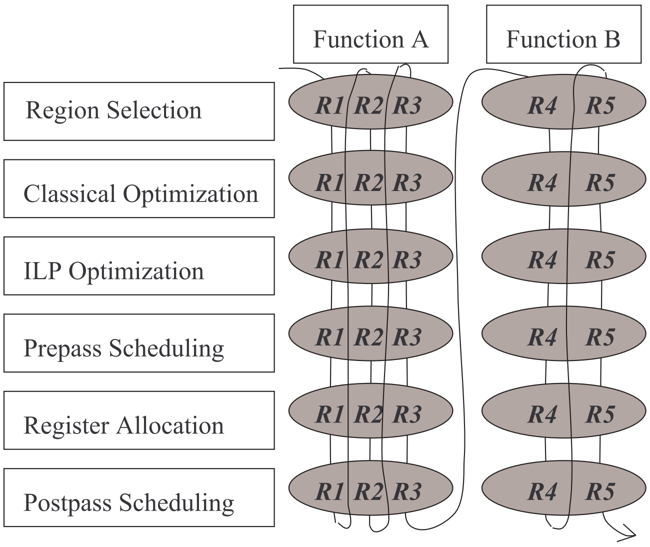

Figure 2.4: Vertical region-based compilation of a function.

level of optimization applied to each region can also be adjusted depending on the region's characteristics, as shown in Figure 2.5. This allows the compiler to focus on aggressively optimizing the most frequently executed portions of the program and less time on the remainder of the code. Note that under this compilation model, basic blocks in different regions but within the same function may be in different phases of compilation at any given time. This allows the compiler the freedom to push any compensation code it generates outside the current region to be optimized into a different region later. Finally, the region compilation process can be taken to a completely global scope as in Figure 2.6. Under this model, seeds, which are the root of region selection, can be chosen from any function at any time during the compilation process. The compiler need not complete the compilation of one function before continuing onto the next. This process will be

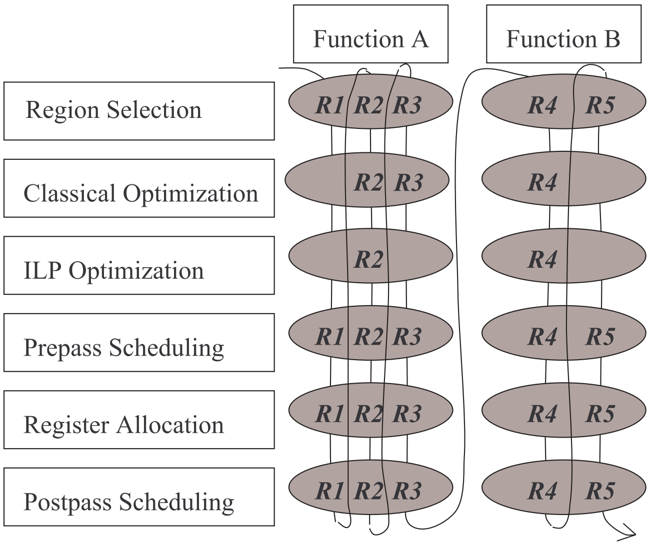

Figure 2.5: Specialized vertical region-based compilation of a function.

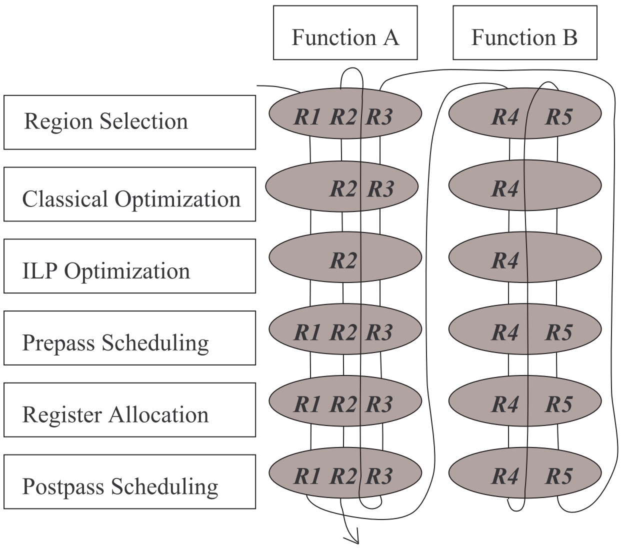

Figure 2.6: Global specialized vertical region-based compilation of a function.

discussed further in Chapter 3 as it is employed in the process of dynamic compile-time control.

2.2.2 Region-based compilation management

The implementation of region-based compilation requires the addition of a distinct phase to the compilation process: region selection. Under this framework, illustrated in Figure 2.7, a compilation manager chooses and applies the appropriate transformations to a region before reintegrating the region back into the function. The goal of region selection is to divide a function into the best possible compilation units to which to apply aggressive transformation. The utility of a region can be measured in two ways. First, the region should be able to be effectively compiled. The main region characteristic which affects this measure is size. By making sure that the region is reasonably sized, we

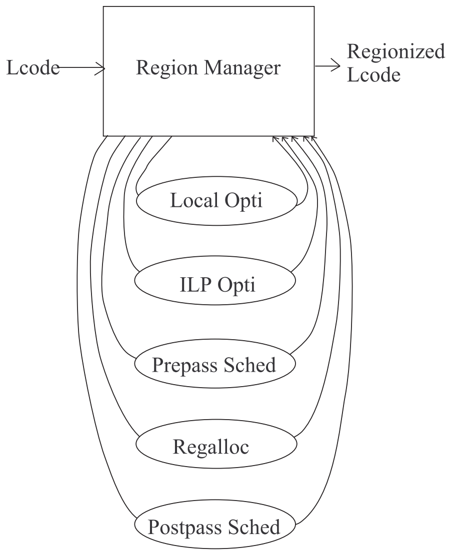

Figure 2.7: Vertical region-based compilation of a function.

can ensure that aggressive transformations can be applied to the region when desired. The second measure of the region is the quality of code which the compiler produces when operating over the regions. This can be controlled by taking into consideration the program's dynamic behavior, hazards, and control flow structure, as well as the dependence height of the operations within the region. Exclusion of basic blocks which are infrequently executed and which may preclude the application of certain transformations is often beneficial. The use of profile information in the region selection process allows the compiler to apply transformations over compilation units which are more representative of the dynamic behavior of the program.

The algorithm used for region selection is a generalization of the profile-based trace selection algorithm used within IMPACT [25]. The difference between the two algorithms is that the region selection algorithm is able to expand a region along multiple control flow paths, while the trace selection algorithm is limited to a single path.

There are four basic steps to the region selection algorithm. First, a seed basic block is selected. This is the most frequently executed block in the program, based on profile weight, which is not already in a region. Once this block is chosen, a path of desirable successors is chosen from the seed. A basic block is deemed desirable for inclusion in the region based on the execution frequency of the basic block and the size of the region. A basic block y is a desirable successor of basic block x if two conditions are met. First, block y is likely to be executed when the flow of control leaves block x. This is considered likely if the control flow transition from block x to block y, $W(x \rightarrow y)$, is at least $(T \times 100)\%$ of the weight of block x, $W(x)$, where T is a threshold value defined by the compiler. Secondly, the execution frequency of y must be at least $(T_s \times 100)\%$ of the execution frequency of s, the seed basic block. This condition prevents the inclusion of an irrelevant block to the region. These conditions are summarized by

$$Succ(x,y) = \left(\frac{\mathcal{W}(x \rightarrow y)}{\mathcal{W}(x)} \geq T\right) \&\& \left(\frac{\mathcal{W}(y)}{\mathcal{W}(s)} \geq T_s\right) \tag{2.1}$$

After the path of desirable successors is found, a path of desirable predecessors is found in the same way growing backward from the seed basic block. The equation which governs a predecessor's inclusion in the path is

$$Pred(x,y) = \left(\frac{\mathcal{W}(y \rightarrow x)}{\mathcal{W}(y)} \geq T\right) \&\& \left(\frac{\mathcal{W}(y)}{\mathcal{W}(s)} \geq T_s\right) \tag{2.2}$$

The final step in the region selection algorithm is to expand the region along multiple paths of control. This is done by considering the successors of every basic block already in the region for inclusion until no more basic blocks meet the conditions stipulated in Equation 2.1. The four steps of the region selection algorithm are summarized in Figure 2.8.

After a region is selected, the compilation manager chooses the level of transformations to be performed based upon the characteristics of the region.

The final issue that the compilation manager must contend with is the maintenance of states between regions. Recall that the region is an arbitrary subgraph of the program

```
1. Select the most frequently executed block not yet in a region
2. Select a path of desirable successors of the seed block.
   while ( y ∋ R && Succ(x,y) ) {
         R = R ∪ {y}
         x = y
         y = most frequent successor of x
   }
3. Select a path of desirable predecessors of the seed block.
   x = seed
   y = most frequent predecessor of x
   while ( y ∋ R && Pred(x,y) ) {
         R = R ∪ {y}
         x = y
         y = most frequent predecessor of x
   }
4. Select all desirable successors of blocks within the region.
   stack = R
   while ( stack ≠ ∅ ) {
         x = Pop(stack)
         for each successor of x, y ∋ R {
               if ( Succ(x,y) ) {
                     R = R ∪ {y}
                     Push(stack,y)
               }
         }
   }
```

Figure 2.8: Profile-sensitive region formation algorithm.

control flow graph. In contrast to a function, which has a clearly defined calling convention at its entrance and exit, a region has live variables which span its *multiple* entry and exit points. Figure 2.9 illustrates a possible region chosen from a portion of a function's CFG. Live variable information across the entrance and exit boundaries of the region is dynamic. As transformations are applied to a region, the dataflow information for each of the basic blocks and its neighbors has the potential to change. When the region is reintegrated into the function, the modified dataflow information must be updated accordingly. The region manager must also store information about register allocation and scheduling decisions made at the boundaries to ensure the correct reintegration of the region into the function. Finally, the compiler needs to know if there is potential for an ambiguous store across a boundary so that it can preclude the application of an illegal transformation if the possibility of memory aliasing exists.

The compilation manager, as implemented in the IMPACT compiler, encapsulates a region in such a way that it has the same properties as a function and thus can have transformations originally designed for functions directly applied to it. This encapsulation captures the live-variable information, the control flow of the region, and possible ambiguous memory references.

To encapsulate a region, boundary control blocks are added at every entry and exit point into and out of the region. A prologue and epilogue are also added to each region. The encapsulation of the region presented in Figure 2.9 is shown in Figure 2.10. Control flow is added to the encapsulated region, taking advantage of the control flow relationships

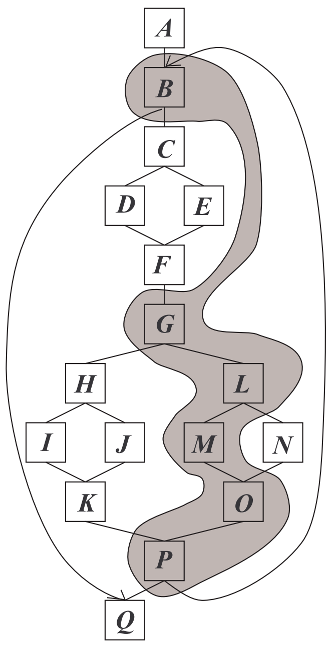

Figure 2.9: Selected basic blocks of region in program CFG.

between the entry and exit boundary blocks. If an exit block dominates an entry block, a control flow arc may be added between them. Any remaining exit blocks which do not dominate an entry are connected to the epilogue. Finally, any entry blocks which are reachable from outside the region are connected to the prologue. Note in Figure 2.10 that entry block **K** need not be connected to the prologue because it is dominated by exit block **H**. The same condition holds true for entry block **F**, exit block **C**, and block **N**, which is both an exit and an entry block. In order to encapsulate this region, the only thing that remains to be done is to connect entry block **A** to the prologue, and exit block **Q** to the epilogue.

Live-variable information is conveyed to each transformation through the use of dummy operations placed in the boundary condition blocks. An explicit reference is

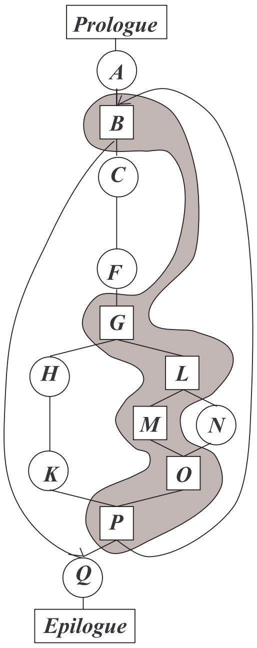

Figure 2.10: Region boundary conditions.

made to all variables which are live across the boundary. Finally, if there is potential for an ambiguous store across a boundary, this is placed as an attribute to the boundary to preclude any illegal memory transformation.

The final point to note about the boundary condition blocks is that they may be used as place holders for any compensation code that is generated during transformation and pushed out into an uncompiled region during reintegration.

3. DYNAMIC COMPILE-TIME CONTROL

To extract a greater degree of ILP from programs, compilers are forced to take a more global view of the program. This includes the application of global optimizations and scheduling. In addition, the compiler attempts to reduce the interprocedural coupling in a program in order to benefit from interprocedural optimization opportunities. Reduction in the coupling between procedures may also expose the true cyclic nature of the program providing more opportunities for the application of loop transformations. Inlining is a tool commonly used to achieve the benefits noted above. However, as inlining is applied and global transformations are utilized, the time and memory requirements of a conventional compiler increase in a fashion nonlinear to function size. This often forces the compiler to scale back optimizations it wishes to apply to a function in the interest of time and memory conservation.

When the compiler applies less aggressive transformations to a function, the benefit of inlining is often lost. The compiler may even produce poorer quality code than if the original source code had been used. Ideally, the compiler would be able to extract all the performance out of a program, as would be possible with aggressive compilation, while containing the compilation time.

The goal of this thesis is to explore this possibility. By focusing the majority of the compilation time on aggressively optimizing the most frequently executed portions of a program, and the remainder on less important portions, code can be produced in a reasonable compilation time which extracts nearly all the performance of the aggressively optimized functions. Region-based compilation is a key element in this process because it allows the compiler to focus on arbitrary portions of the control flow graph and apply varying levels of optimizations to these segments based upon the characteristics of the region, the time which has already been spent compiling the program, and the percentage of the program which still needs to be processed.

3.1 Algorithm

The region compilation manager within IMPACT has been extended to perform dynamic compile-time control of vertical region-based compilation. Under this model, the manager is responsible for selecting a region, extracting the region, guiding it through the appropriate transformations and then reintegrating the transformed region into the function from which it was extracted. This algorithm is based on two user-specified parameters: first, a target compilation time, and second, a pivot point. For instance, one can specify that $x\%$ of the program be compiled in $y\%$ of the target compile time and the rest of the program compiled in the remaining time. As the compilation of the program progresses, the compile time manager evaluates how much of the program has been compiled and what percentage of the target compile time has been expended, and uses this information to choose a level of optimization for each selected region.

The first task of the compile time manager is to prescan all of a functions in the program. It does this to obtain a global portrait of the program and to locate its most important portions. For the purposes of this thesis, importance is measured solely by the execution frequency of a block. Once this is done, the manager selects what it considers to be the most important region in the program. The level of optimization to be applied to this region is chosen in the following way. The user specifies a maximum compilation time for the compile-time manager. This is the time this program would take to compile if the greatest level of optimizations was applied uniformly to it. If the target compile time is more than 60% of the maximum compilation time, then the most aggressive level of optimizations available is applied to this region. If this percentage is less than 60%, a sliding scale is used to determine the level of optimizations to be applied to this region. This scale is included in Figure 3.1. The reason for this step is to prevent the first region from always being compiled at the highest level of optimization and potentially shooting the actual compile time over the target compile time before the dynamic compile-time algorithm has a chance to be applied. The levels of optimizations available to the manager will be more thoroughly explained in Section 3.1.1. When this region has been completely compiled and reintegrated, the manager chooses the next most important region to be compiled. At this point, the compile-time manager evaluates the percentage

1. If this is the first region compiled

    $s_{target} = s_1$

    $p_{maxtime} = t_{target}/t_{max}$

        $l_{new} = 0$

    If $p_{maxtime} < 10\%$

        $l_{new} = 0$

    If $p_{maxtime} < 20\%$

        $l_{new} = 1$

    If $p_{maxtime} < 30\%$

        $l_{new} = 2$

    If $p_{maxtime} < 40\%$

        $l_{new} = 3$

    If $p_{maxtime} < 50\%$

        $l_{new} = 4$

    If $p_{maxtime} < 60\%$

        $l_{new} = 5$

        $l_{new} = 6$

2. $s_{cur} = p_{time}/p_{compiled}$

3. If $p_{time} > p_{percenttime}$

    $s_{target} = s_2$

    $s_{adjtarget} = s_{target}$

4. else $s_{adjtarget} = p_{percenttime}/p_{time} * s_{target}/2$

5. $p_{greater} = (s_{cur} - s_{adjtarget})/s_{adjtarget}$

6. If $p_{greater} > 100\%$

    $l_{new} = 0$

7. If $100\% < p_{greater} < 75\%$

    $l_{new} = l_{new} - 4$

8. If $75\% < p_{greater} < 50\%$

    $l_{new} = l_{new} - 3$

9. If $50\% < p_{greater} < 25\%$

    $l_{new} = l_{new} - 2$

10If $25\% < p_{greater} < 0\%$

    $l_{new} = l_{new} - 1$

11return l_{new}

Figure 3.1: Dynamic compile-time, level-determination algorithm.

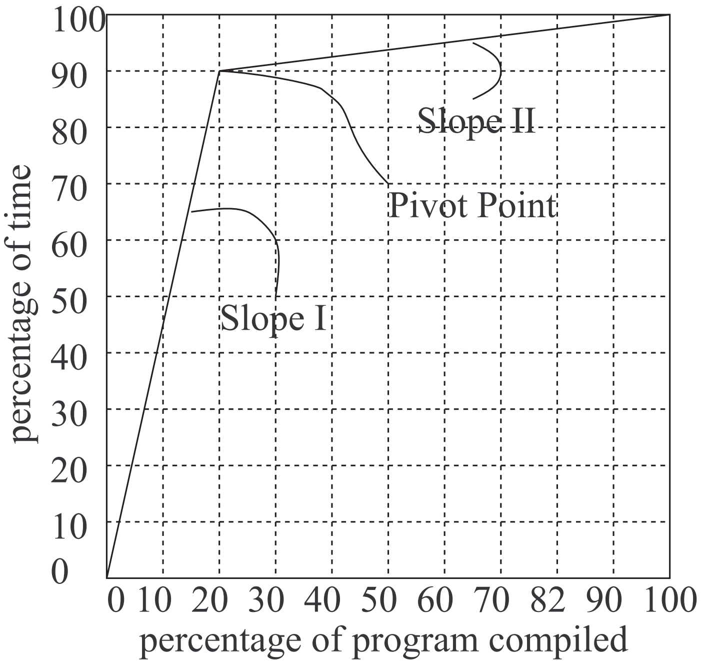

Figure 3.2: Piecewise linear function used to control compile time.

of the program compiled, the percentage of the target time expended, and the current optimization level to determine whether to apply the same level of transformation to this region, or to apply a lower level of transformation. The manager makes this decision in the following manner. The pivot point and the target compilation time, as specified by the user, are used to construct a piecewise linear function as in Figure 3.2. This function is composed of two lines with two respective slopes. The first slope is the target slope of compilation during the first compilation portion, and the second is the target slope of compilation for the remainder of the program. The goal of the compilation manager is to follow the slopes of these lines as closely as possible.

The algorithm used to dynamically track the piecewise function is outlined in Figure 3.1. The algorithm evaluates its position in an x-y graph after the compilation of each region is complete. In this coordinate system, the x axis is the percentage of the program compiled. An estimate of this is made by dividing the number of basic blocks which have been compiled by the number of basic blocks in the program. The y coordinate of the graph is the percentage of the target compile time which has been expended. Once this point has been located in the graph, a line is drawn from the origin through the point, and the slope of this line in calculated. At this point, an adjusted target slope is calculated. This slope takes into consideration how far along we are in the compilation process and allows for a greater variation from the target slope the further we are from the pivot point. The adjusted slope is calculated by taking the target percent time and dividing it by the current percent time and then taking this ratio and multiplying it by the target slope. This creates a threshold cushion as illustrated in Figure 3.3. If the slope of the line is more than 100% greater than the adjusted target slope, then the compilation level is dropped to level zero. A sliding scale is again used in this case to decide how many notches the current optimization level should be lowered by.

3.1.1 Levels of optimization

The optimization process within the IMPACT compiler is iterative. It involves the process of searching for optimization opportunities, the application of the transformation,

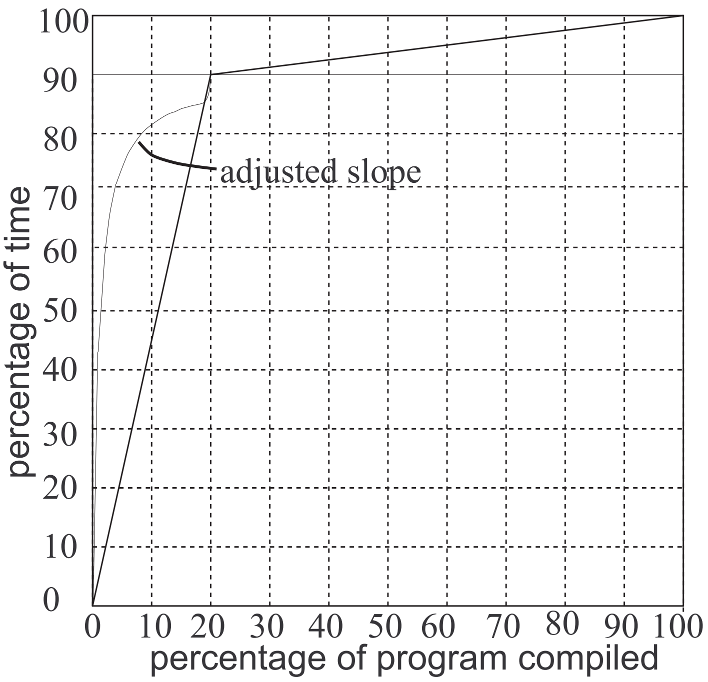

Figure 3.3: Adjusted slope cushion used with piecewise linear function.

and the upkeep of the dataflow information in light of these transformations. The suite of optimizations include local, global [11], jump, and loop transformations.

The following local optimizations are performed: 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, and code reordering. The following global optimizations are performed: constant propagation, forward copy propagation, backward copy propagation, memory copy propagation, common subexpression elimination, redundant load elimination, redundant store elimination, and dead-code elimination. The following loop optimizations are performed: invariant code motion, global variable migration, branch simplification, induction variable strength reduction, and induction variable elimination. The following jump optimizations are performed: dead block removal, branch elimination, branch to jump optimization, merge always successive blocks, branch target expansion, branch prediction, and combine labels.

The IMPACT compiler also has the ability to perform superblock formation and optimization [26]. A superblock is a single-entry, multiple-exit path of basic blocks through a function which are combined into a single control flow block. This combination has the potential to produce optimization and code motion opportunities. Once a superblock has been formed, a specialized suite of transformations is applied to it to take advantage of these new opportunities.

In the context of superblock formation, the process of tail duplication has been employed to eliminate side entrances from a superblock so that the optimizer need not take them into consideration [27]. This same technique can also be applied to eliminate side entrances from arbitrary regions. Tail duplication duplicates all blocks within a region that are reachable from a side entrance, thus eliminating the entrance.

There are seven levels of optimization available to the compile time compilation manager. At Level 0, no optimizations are applied to the region. At Level 1, dead block removal, local optimizations, and dead code removal are performed. Level 2 adds global optimizations to the suite. Level 3 applies jump optimizations, while Level 4 adds loop optimizations to the suite of transformations. Level 5 applies tail duplication to the regions to remove any side entrances from the region. Finally, in Level 6, the highest level of optimization, superblock formation and optimization are applied after the optimizations in Level 4 are complete. Table 3.1 summarizes the optimizations performed at each level.

3.1.2 Region selection scope

An additional option of the compile-time manager is control of the region seed selection scope. A program can be compiled using either a global or a function scope. If the program is compiled with a global scope, a seed basic block from which to grow a region can be chosen from any function in the program. Under this model, a region is selected based upon that seed. The region is then compiled and reintegrated back into its parent function. At this point, the manager is again able to select a seed from any function in the program. Conversely, if the manager is limited to a function scope, the function with the greatest execution weight not already compiled is selected. Seed selection then occurs exclusively within this function until it is completely compiled. This limits the granularity of the compilation process, as illustrated in Figure 3.4.

Within the context of both function and global seed selection scope, Region 1 is chosen as the most important portion of the program. However, if the dynamic compilation time controller is operating under the function seed selection scope, then the selection

Table 3.1: Levels of Optimization.

Level	**Optimizations**
0	no optimizations
1	dead block removal local optimization dead code removal
2	dead block removal local optimization global optimization
3	dead block removal local optimization global optimization jump optimization
4	dead block removal local optimization global optimization jump optimization loop optimization
5	dead block removal local optimization global optimization jump optimization loop optimization tail duplication
6	dead block removal local optimization global optimization jump optimization loop optimization superblock formation and optimization

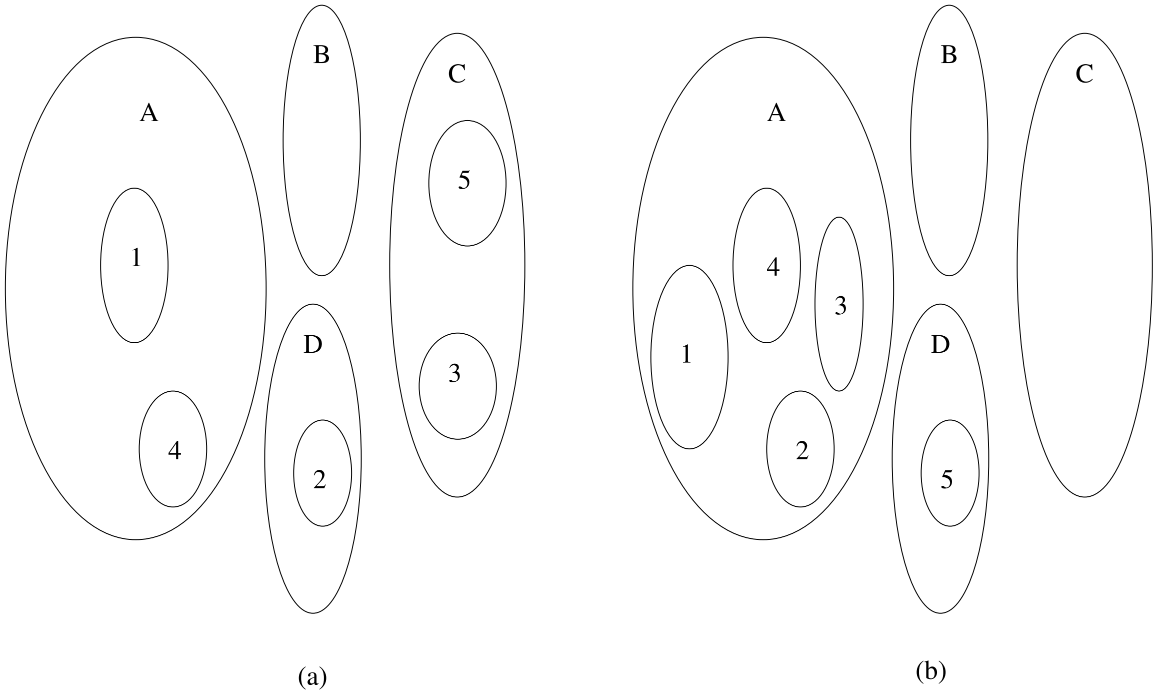

Figure 3.4: (a) Global versus (b) function seed selection scope.

of the next seed must continue in function **A** until the function is completely compiled. Conversely, with global seed selection scope, the compiler is able to choose the second region from function **D** and the third from function **C**. Its selection scope is not limited as with function-based seed selection scope. The disadvantage of utilizing global selection scope is that is can be a memory-intensive functionality. This issue is addressed in Section 3.1.3. The effects of these two seed selection scopes will be will be further studied in Chapter 4.

3.1.3 Memory requirements

As mentioned in Section 3.1, the compile-time manager must access all of the files in a program at initialization in order to perform a prescan of the functions and their execution weights. This obviously provides the potential for dynamic compile-time management to require a great deal of memory. Under ideal circumstances, we would like to see every function remain in memory from the point when it is loaded for prescanning until it is completely compiled. However, with a large program, this may obviously become too memory intensive. To avoid this problem, a memory management algorithm is implemented within the compile-time manager which maintains an LRU stack. When the memory usage exceeds a user-defined maximum value, the manager closes the least recently accessed file. This will be reopened when and if necessary. This option will not be fully explored in this thesis, but merits mention as a memory control feature.

4. EXPERIMENTAL PERFORMANCE EVALUATION

This chapter evaluates the effectiveness of the dynamic compile-time algorithm proposed in Section 3.1. First, in Section 4.1, an investigation is performed on how different pivot points affect the performance of the algorithm. In Section 4.2, the point is identified at which increasing the compile time further has a negligible effect on the performance. The chapter then studies the accuracy of the algorithm in Section 4.3. In Section 4.4, a comparison between benchmarks compiled using dynamic compile time control and traditional compilation is performed. Finally, in Section 4.5, the relative merits of global and function seed selection scope are studied.

In this thesis, select integer benchmarks from SPEC95 and several UNIX utilities are utilized. Static performance numbers are used to measure performance.

4.1 Choosing a Pivot Point

As mentioned in Section 3.1, a pivot point is one of the key parameters supplied by the user. This point will specify, for instance, that $x\%$ of the program should be compiled in $y\%$ of the target compile time. To determine an appropriate value at which to set this pivot point for the remainder of the tests performed in the thesis, the performance benchmarks were compile at several different pivot points. The target compile time was set to be the time used during traditional compilation of this program at a level of optimization comparable to Level 3.

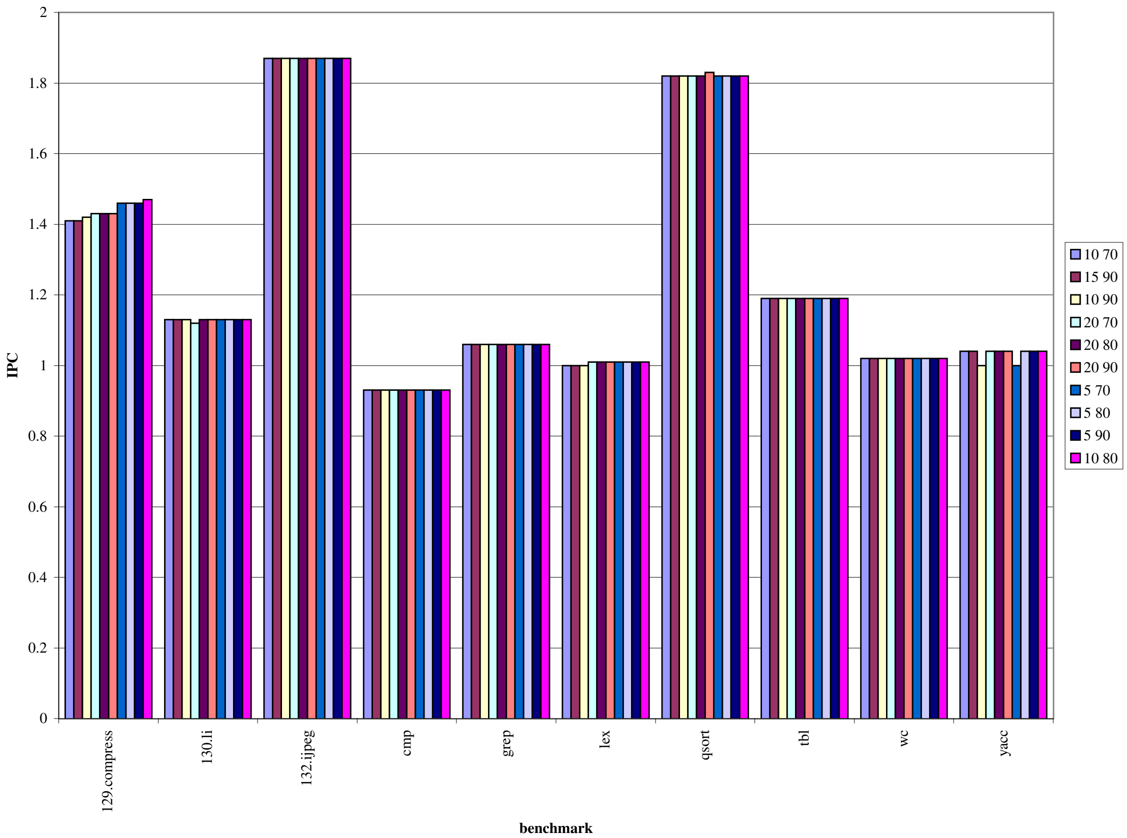

Figure 4.1: Comparison of pivot points.

Figure 4.1 shows the results of these tests. The x axis of this graph is the benchmark compiled. The y axis is the CPI of each benchmark at a specific pivot point. The value of the pivot point is represented by the shade of the bar. There are ten pivot points shown in this graph. In the legend, the first number is the percentage of the program compiled and the second is the percentage of the total compilation time. It appears that for the pivot points chosen there is no difference in performance for some of the benchmarks shown and a marginal increase in performance for *129.compress*, *qsort*, and *yacc* with a 20/90 pivot point. This pivot point specifies that the goal of the compiler was to compile 20% of the program in 90% of the total compilation time. This was the pivot point chosen to be utilized for the remainder of the thesis.

4.2 Finding Appropriate Compile Time

Using the pivot point chosen in Section 4.1, each benchmark is again compiled. In this experiment, the target compile time is incremented from minimum to maximum and the progress is charted. Figure 4.2 shows the results of this for the benchmark *qsort*. The x axis on this graph is the percentage of maximum compile time, and the y axis is the CPI of the program when compiled at target time. The minimum time used for this experiment is the time taken to compile the function using region-based compilation if there are no optimizations done on the program. The maximum compile time, conversely, is the time required to compile the program applying optimizations comparable to Level 7 in the traditional framework. It can be seen from Figure 4.2 that there is an elbow in the graph. This elbow is the point where increasing the compile time of the program further has little or no effect on its performance. For the benchmark *qsort*, this point appears to be when the target compile time is about 37% of the maximum compile time. To increase the compile time beyond this point gains no performance and actually decreases performance slightly in this case. The same elbow can be found for the benchmarks *129.compress*, *cmp*, *grep*, *yacc*, *tbl*, and *wc* in Figures 4.3-4.7.

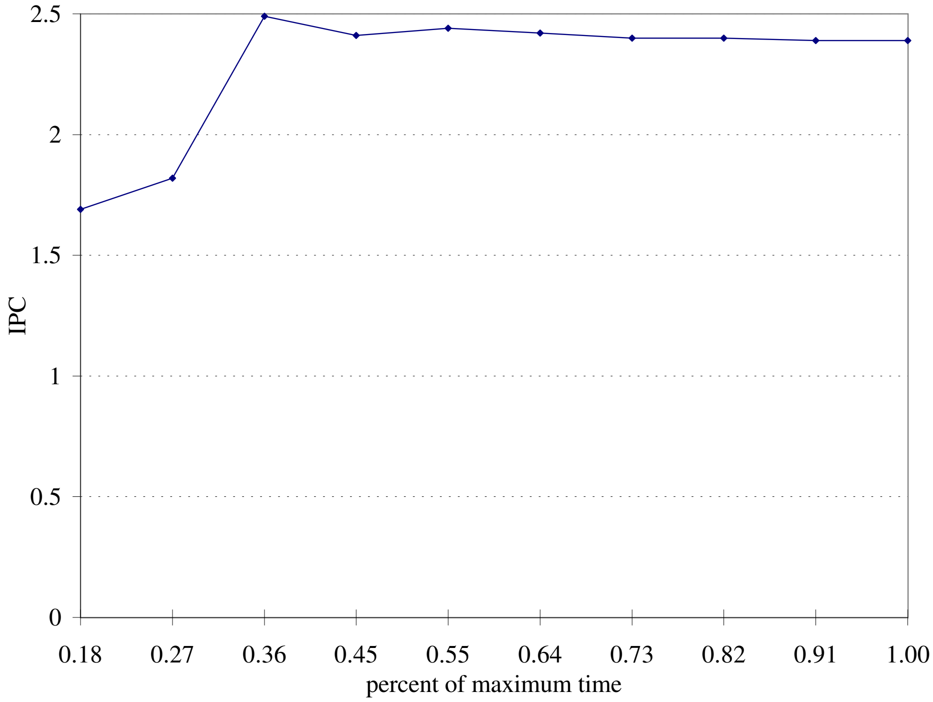

Figure 4.2: IPC of *qsort*.

4.3 Dynamic Compile Time Algorithm Evaluation

This section evaluates the algorithm used to dynamically control the compile time of a benchmark. The algorithm is judged primarily in two ways. First, the distribution of the number of regions compiled at each optimization level is presented in Section 4.3.1. Second, in Section 4.3.2, the accuracy of the algorithm in meeting the target compile time is explored.

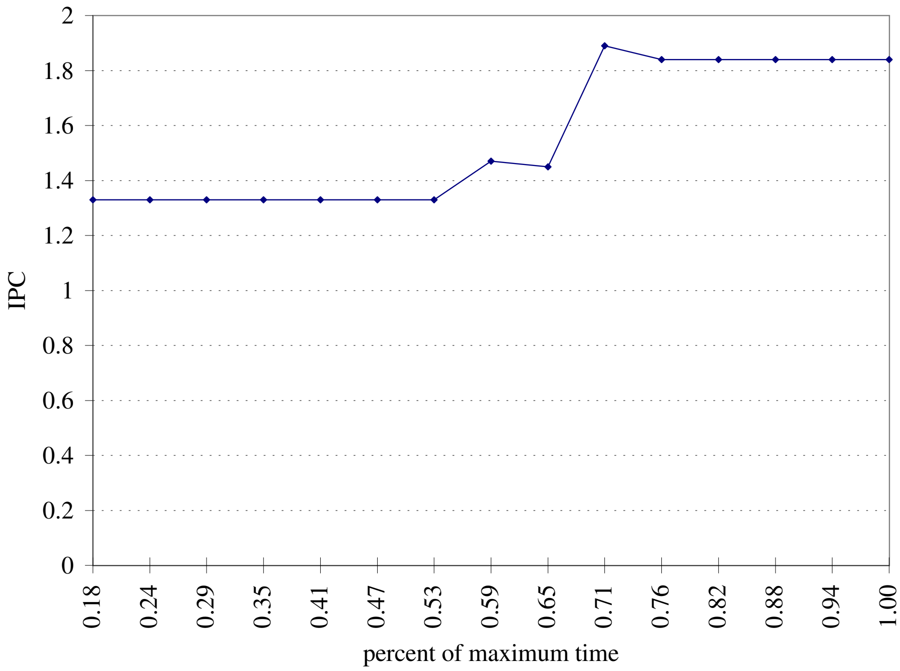

Figure 4.3: IPC of *129.compress*.

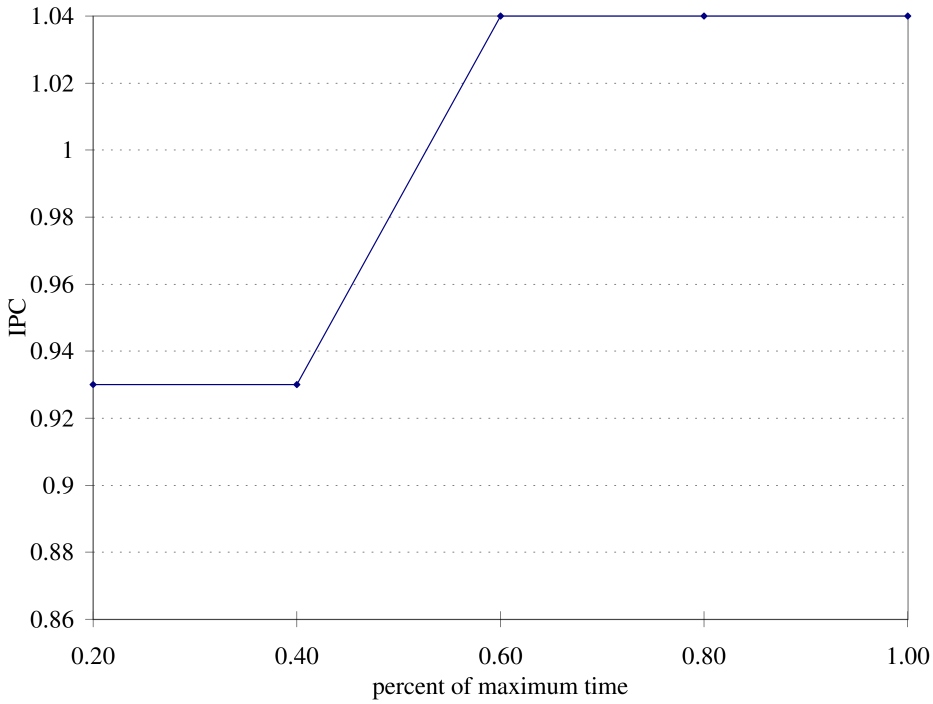

Figure 4.4: IPC of *cmp*.

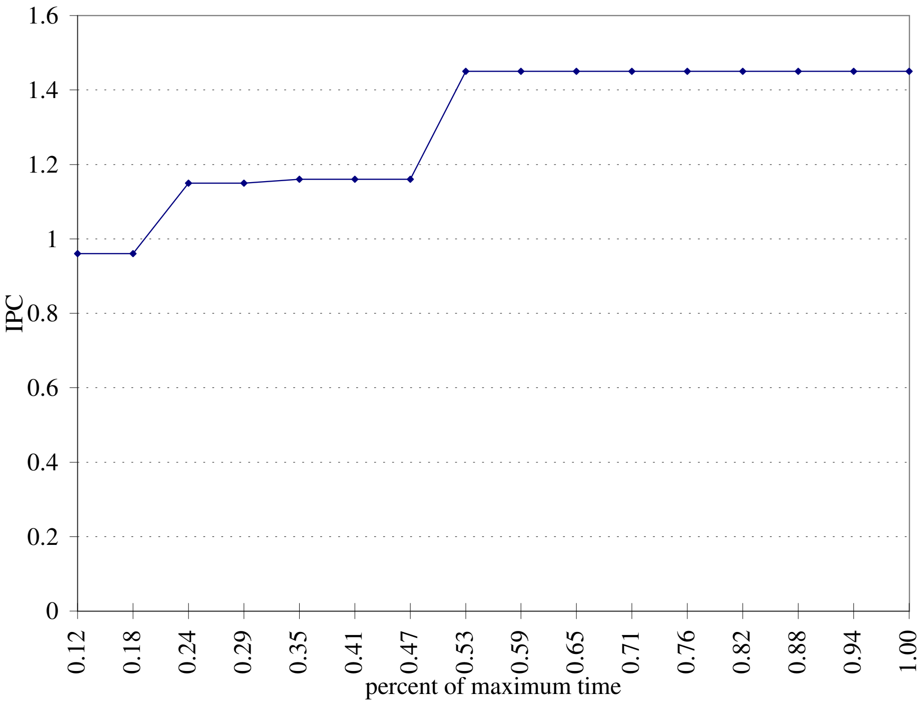

Figure 4.5: IPC of *grep*.

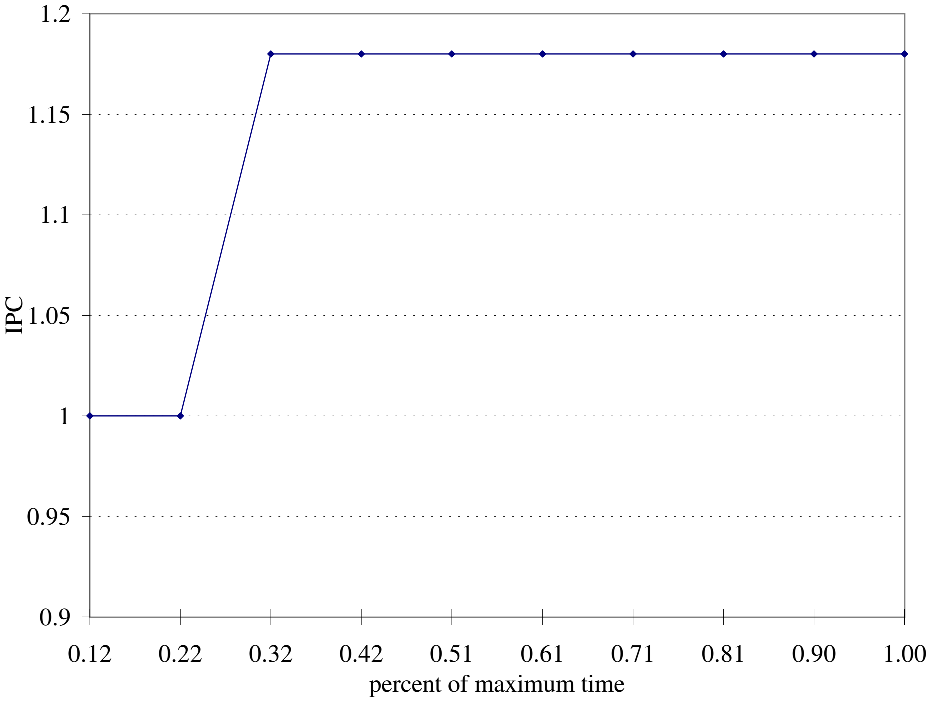

Figure 4.6: IPC of *tbl*.

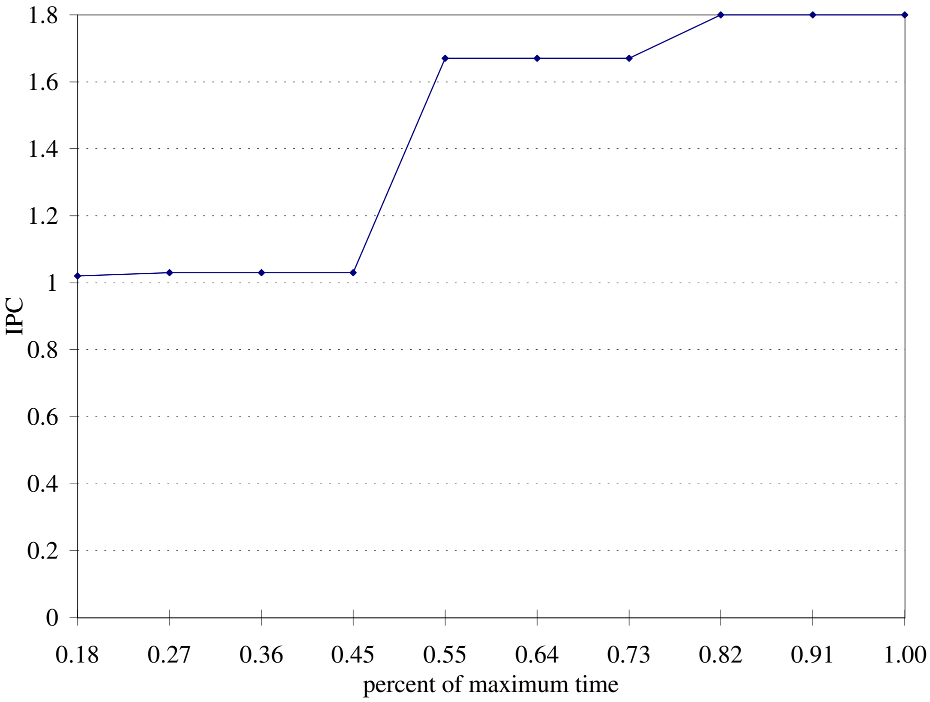

Figure 4.7: IPC of *wc*.

4.3.1 Distribution of optimization level over program

In order to ensure that the algorithm is evenly distributing the number of regions compiled at each level over the target compilation time, Figure 4.8 is introduced. This graph presents the number of regions compiled at each optimization level for *qsort*. Each bar represents the distribution of regions compiled at a given target time. Note that as the compilation time increases, so does the level of optimizations applied to the program. This is the behavior we would expect to occur when the algorithm was functioning correctly. This same quality can be observed in the rest of the benchmarks in Figures 4.9-4.14.

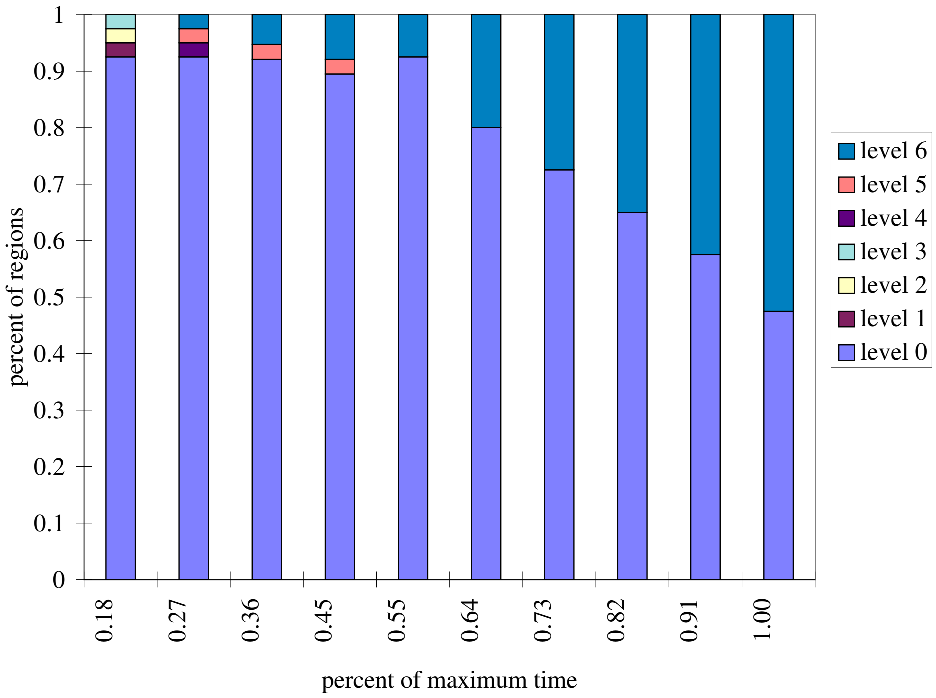

Figure 4.8: Distribution of optimization levels over *qsort*.

4.3.2 Accuracy of algorithm

The success of the algorithm at meeting the target compile time is presented in Figure 4.15. On this graph, the x axis contains each benchmark and the y axis is a ratio of the actual compile time to the target compile time. Different shades of bars represent the increasing target compile time, so the leftmost bar for each benchmark is at the minimum target compile time, while the rightmost bar is at the maximum compile time. Some of the benchmarks have fewer bars because they were compiled over a smaller span of minimum compile time to maximum compile time, allowing fewer data points to be gathered. It can be seen that in most cases the dynamic compile-time algorithm comes very close to the target time. The ability of the algorithm to meet the target compile

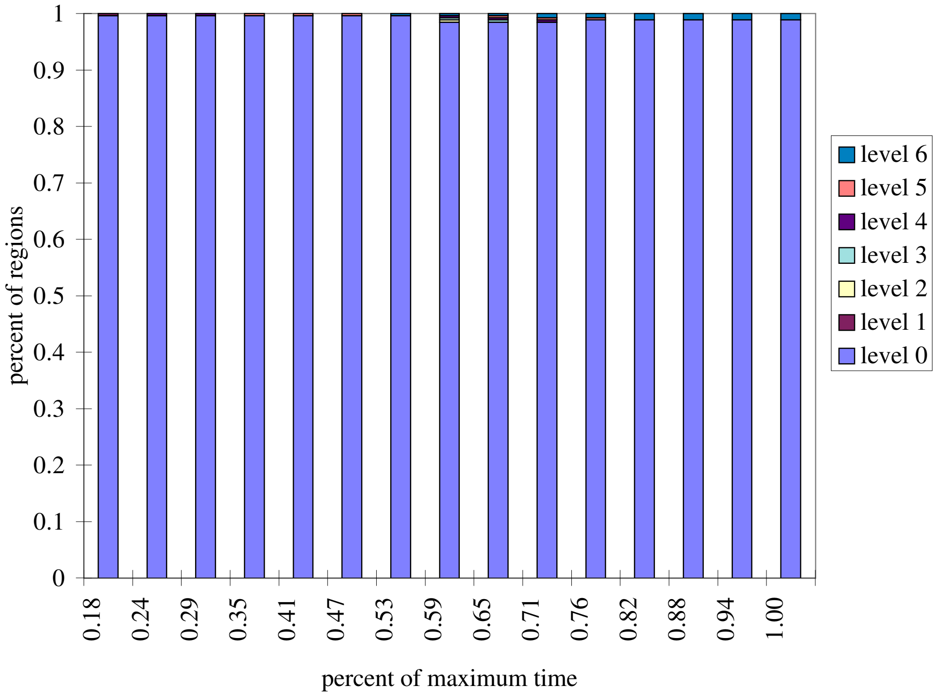

Figure 4.9: Distribution of optimization levels over *129.compress*.

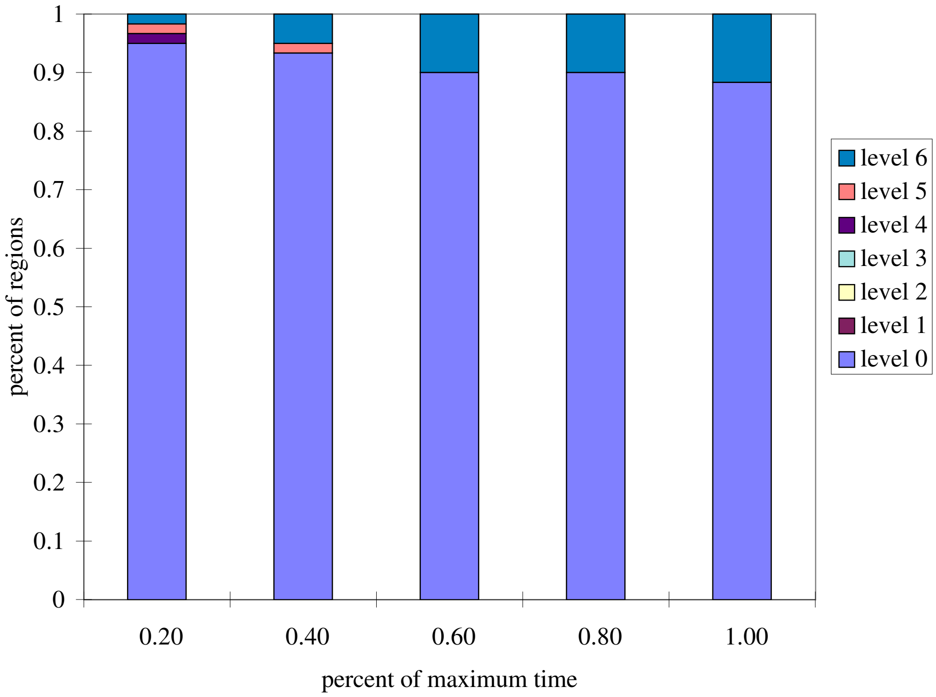

Figure 4.10: Distribution of optimization levels over *cmp*.

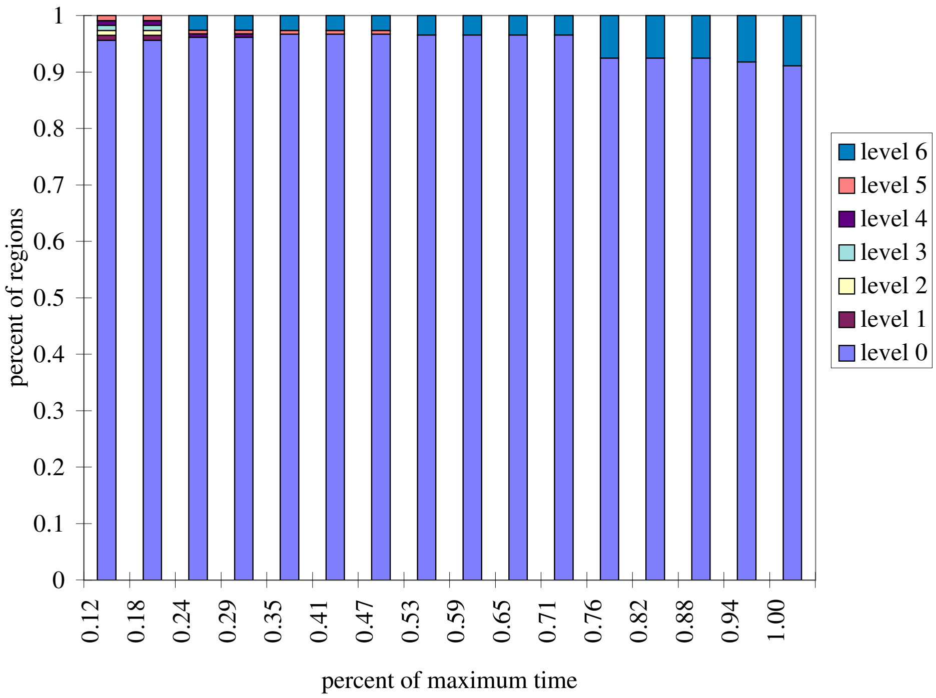

Figure 4.11: Distribution of optimization levels over *grep*.

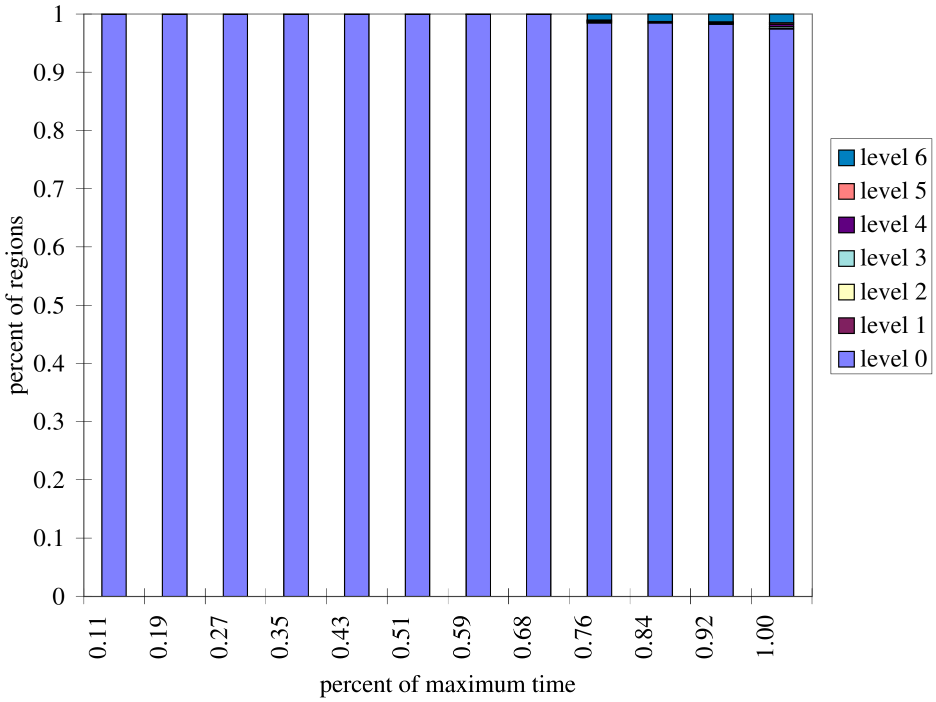

Figure 4.12: Distribution of optimization levels over *yacc*.

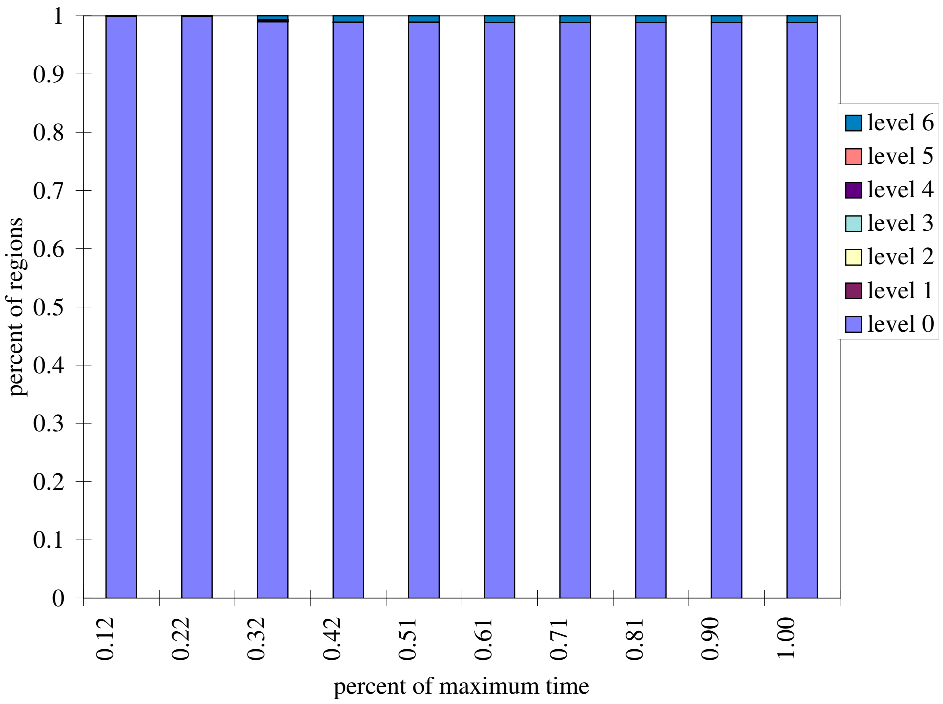

Figure 4.13: Distribution of optimization levels over *tbl*.

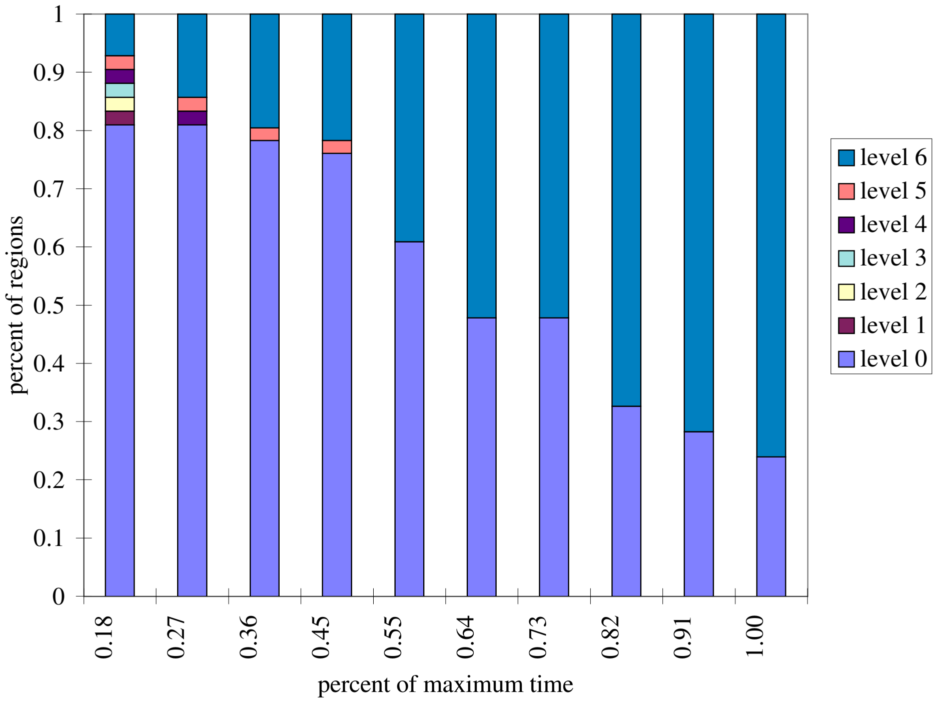

Figure 4.14: Distribution of optimization levels over *wc*.

time is limited by the granularity of the regions. It is impossible for the compile-time control algorithm, when it applies an optimization to a region, to predict how long that optimization will require The cases where the target and actual compile time differ by a great deal are typically due to a large region which the compiler deemed very important and optimized heavily, expending a lot of the target compile time. When this is the case, the dynamic compile time algorithm is often unable to meet the target compile time because too much of the time has been expended on one region, leaving too little time to compile the rest of the regions, even at optimization Level 0. This is most obviously the case for the benchmark *tbl*.

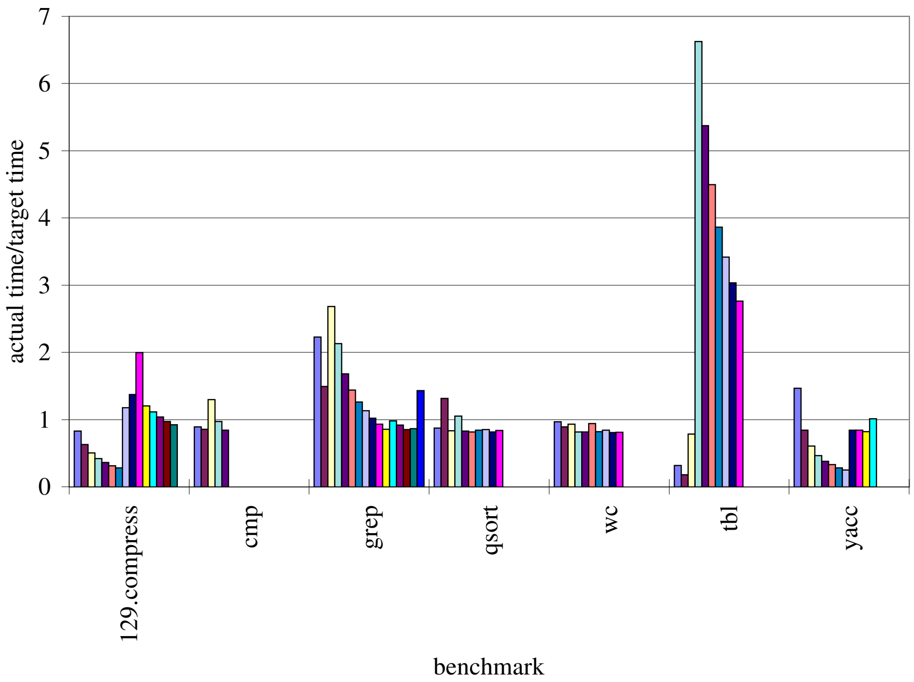

Figure 4.15: Accuracy of dynamic compile-time control.

4.4 Comparison of Performance

The goal of the dynamic compile time algorithm is to extract nearly all the ILP from a program, which is possible with aggressive optimizations in a more reasonable compile time. Figure 4.16 exhibits the effectiveness of the algorithm in doing this. In this figure, four sets of bars are present for each benchmark. Each set contains two bars. The first set represents compilation at optimization Level 3 under the traditional framework. The second represents compilation at optimization Level 4 under the traditional framework. The third is the performance of dynamic compile-time control, while the fourth is optimization at Level 4 with superscalar optimizations applied under the traditional framework. The first bar in each set represents the percentage of the time taken at the respective optimization level to compile the benchmark at optimization Level 4 and apply superscalar optimizations under the traditional compilation framework. The second bar in each set represents the percentage of the ILP for the program when compiled at the respective level of the ILP when compiled at optimization Level 4 with superscalar optimizations applied under the traditional framework. The x axis contains each benchmark and the y axis contains the percent. It can be seen from this graph that in many cases the dynamic compile-time algorithm was successful in extracting a high level of ILP in less time than was required to apply both superscalar optimizations and Level 4 optimizations.

From this figure, we can conclude that the dynamic compile-time control provides the flexibility to specify a reasonable compile time and extract as much performance at that

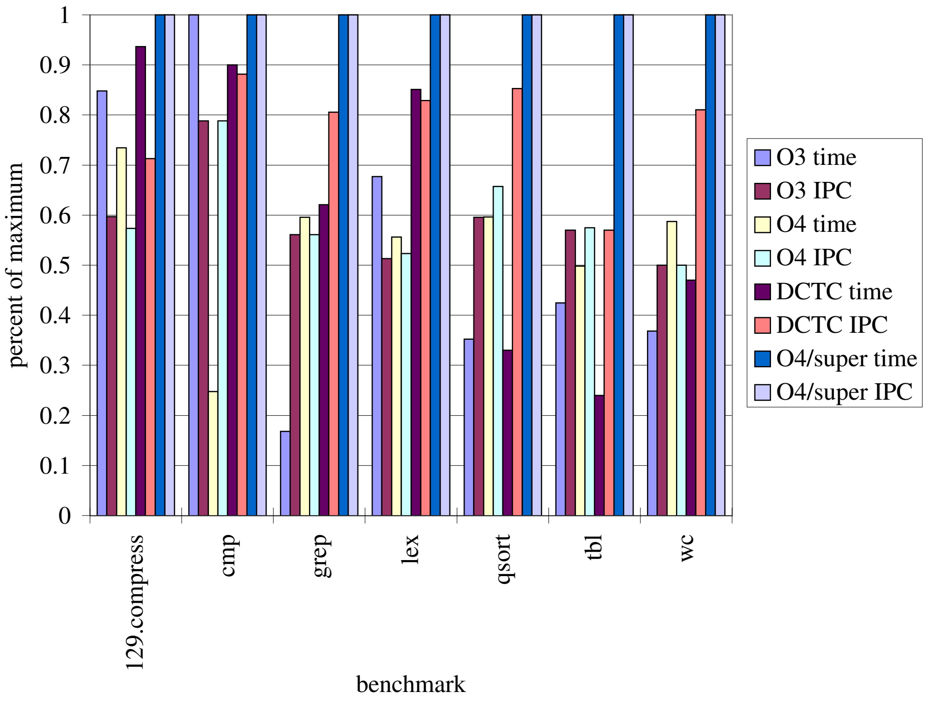

Figure 4.16: Performance of dynamic compile-time control.

compile time as possible. Consider again the benchmark *qsort*. It can be seen that the dynamic compile-time algorithm was able to extract 85% of the maximum ILP in only 33% of the time it took to compile the program at optimization Level 4 with superscalar optimizations under the traditional framework. Other benchmarks which performed well were *wc*, *tbl*, and *grep*.

4.5 Global versus Function Scope

As mentioned in Section 3.1.2, the global view the compiler takes during the dynamic compile time process can be limited to a function scope. Figure 4.17 shows the results of imposing this limitation. In most cases, the IPC that is extracted by the dynamic compile-time algorithm when limited to the function scope is less than that produced with a global seed selection scope. This is due to the inability of the algorithm to evaluate all the functions in the program at the same time. In some cases, however, the function scope dynamic compile-time algorithm does out perform the global scope algorithm. This is a phenomenon which needs to be explored further.

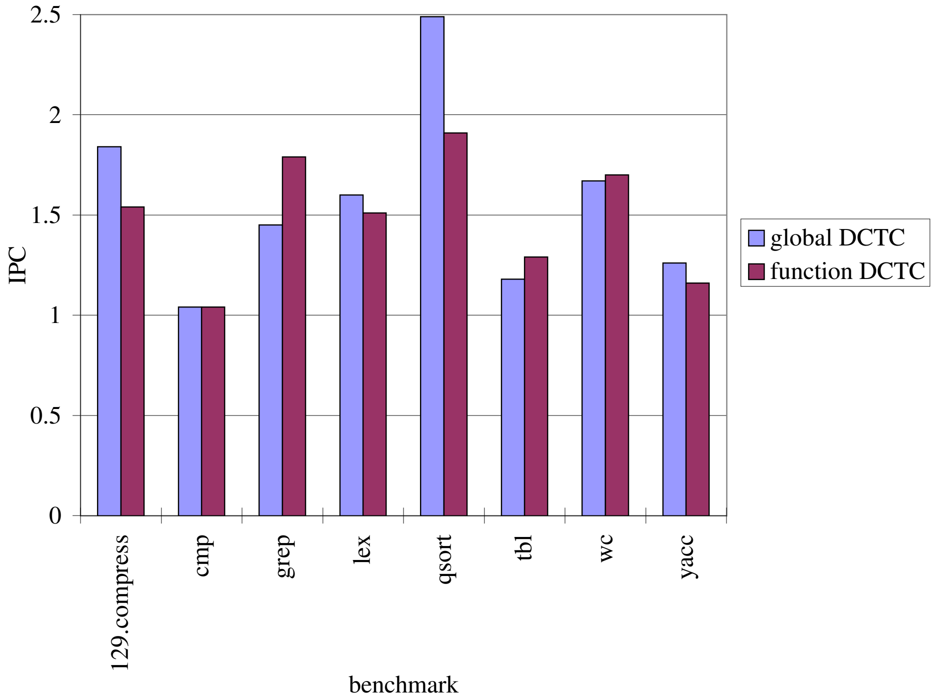

Figure 4.17: Global versus function seed selection scope.

5. CONCLUSIONS AND FUTURE WORK

As stated in Chapter 3, the goal of this thesis is to focus the majority of the compilation time on aggressively optimizing the most frequently executed portions of a program while spending less time on the remainder of the program. By doing this and dynamically adjusting the optimizations applied to meet a target compilation time, it was hoped that nearly optimal-quality code could be produced in a more reasonable time. Under the traditional compilation model, the same suite of transformations is applied to an entire function, or even an entire program. This increases the compilation time of a program in fairly large intervals. The dynamic compile-time algorithm introduces an alternative to this model. Using region-based compilation, the dynamic compile time controller is able to apply varying levels of optimization to different regions of the program, processing the sections of a program from most to least important and scaling back the optimizations it applies based on how long compilation has taken thus far.

In this regard, the dynamic compile-time controller has had some success. It is able to produce code in less time than it would take the traditional compiler to apply Level

4 and superscalar optimizations while extracting nearly all the ILP. It should be noted here that while this algorithm has evolved significantly from its initial implementation, it should be studied further to gain a better understanding of why it performs extremely well on some benchmarks and not as well on others. This knowledge would allow the dynamic compile-time algorithm to be fine-tuned to perform well on a wider span of programs.

Overall, the concept has shown potential as a tool to control the increasing compile time of programs while maintaining the quality of the code produced. Further investigation of the algorithm itself, the memory requirements of the procedure, and global versus function seed selection scope is in order.

REFERENCES

[1] R. E. Hank, "Region based compilation," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1996.

[2] R. E. Hank, W. W. Hwu, and B. R. Rau, "Region-based compilation: An introduction and motivation," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 158–168, December 1995.

[3] 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.

[4] 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.

[5] B.-C. Cheng, "Pinline: A profile-driven automatic inliner for the impact compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1997.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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, 1995.

[16] R. A. Bringmann, "Compiler-Controlled Speculation," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[17] 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.

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

[19] 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, November 1995.

[20] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for VLIW and supercalar processors," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 238–247, October 1992.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.

[27] 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.