ELIMINATING DYNAMIC COMPUTATION REDUNDANCY

BY

DANIEL ALEXANDER CONNORS

B.S., Purdue University, 1994
M.S., University of Illinois, 1997

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, 2000

Urbana, Illinois

ELIMINATING DYNAMIC COMPUTATION REDUNDANCY

Daniel Alexander Connors, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 2000
Wen-mei W. Hwu, Advisor

The traditional method of extracting performance from programs is based on scaling processor resources to execute multiple independent instructions per cycle. In order to enable their cost-effective performance potential, these processors demand that increasing levels of instruction-level parallelism (ILP) be exposed in programs. As a result, compilers must perform increasingly more aggressive analysis and global optimization. Nevertheless, current state-of-the-art compilers cannot expose the level of ILP necessary to overcome the diminishing performance returns of high-issue processors. Ultimately performance becomes limited by the dependences of programs, and not the machine resources. Thus, one of the major challenges to increasing processor performance is overcoming the fundamental dataflow limitation imposed by data dependences. By reusing previous computation results, the dataflow limit can be surpassed for sequences of operations that are otherwise redundantly executed. Effective reuse of previous computation results requires coordinating compiler and hardware techniques in an integrated framework.

Many traditional compiler techniques eliminate program redundancy and optimize the effectiveness of the program. These optimization techniques rely on the detection of static redundancy, which requires computations to be completely redundant for all possible executions. Currently, compiler techniques have no mechanism for capturing dynamic redundancy, redundancy occurring over a temporal set of definitions. As a result, empirical program behavior studies indicate that many instruction traces are dynamically executed with the same inputs, a form of redundancy known as *value locality*. Such sequences of redundant operations reduce

the effectiveness of ILP processors by wastefully utilizing the multiple functional unit resources. Consequently, the available silicon resources are more effectively allocated to exploiting program redundancy than instruction-level parallelism. This dissertation proposes and investigates architectural and compilation techniques that eliminate dynamic redundancy to improve the resource utilization and performance.

The four key technologies here presented work in a coordinated fashion to eliminate dynamic redundancy from program execution. The Reusable Computation Region Framework (RCRF) provides a compilation framework to accurately determine the regions, called Reusable Computation Regions (RCRs), of a program in which reuse is likely to occur. Generally, these regions are recurring tasks and invariant computations that represent the redundant execution behavior occurring within general purpose programs. By locating and designating these regions at compile time, effective run-time techniques for directing the reuse of computation results can be achieved. To exploit the execution behavior of these regions, this dissertation introduces a reuse approach, called Compiler-directed Computation Reuse (CCR). The CCR approach integrates compiler and architecture techniques to exploit value locality for large regions of code. In this approach, the Instruction Set Architecture (ISA) provides a simple interface for the compiler to communicate the run-time execution of each region into a hardware structure.

The RCRF techniques can identify redundantly executing regions within programs based on analyzing value-profile execution information gathered using training input evaluations. However, since most modern software environments do not support value profiling, static region formation techniques are also developed. To support static region formation in the compile-time techniques, it becomes necessary to add run-time management support to the CCR approach. The Dynamic Computation Management System (DCMS) enhances the effectiveness of compiler-directed computation reuse by dynamically controlling deployment of statically-selected computation regions by gathering run-time program execution behavior. In addition, program invariance and computation redundancy occur in phases during run-time execution, and compile-time reuse decisions need to adapt to program behavior and contention for the reuse resources. The DCMS uses hardware tables to gather run-time execution behavior to enable accurate allocation policies of the computation reuse buffer. Within the given framework, the DCMS can determine which compiler-selected computations are currently best exploited using a processor's reuse resources.

Finally, a class of computation regions formed by the RCRF can be exploited using compiler transformation and existing architecture support. In the compiler-based Value Optimization Framework (VOF), program reformulation and predicated execution-based value transformations are used to exploit the redundancy of regions. These techniques offer alternatives for the compiler to synthesize codes based on data distribution rather than traditional practice of basic program conversion.

The Reusable Computation Region Framework, the Compiler-directed Computation Reuse Approach, and the Dynamic Computation Management System represent a fundamentally new method of controlling the microarchitecture execution engine of processors to improve program performance. The compiler-based Value Optimization Framework illustrates a new compiler methodology for synthesizing code based on data distribution. Systematically coordinating these compiler techniques and hardware technologies can eliminate significant amounts of the dynamic computation redundancy in program execution. These techniques are new methods of improving modern processor utilization and performance by exploiting readily available program value locality characteristics.

DEDICATION

Dedicated to my mother, father, sister, and brother.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Wen-mei Hwu, for his guidance throughout my graduate studies. Most importantly, I would like to thank him for his patience and understanding. Several occasions I know I truly depended on his advice and his insight. I would also like to thank him for providing me with every opportunity to achieve my dream of becoming a professor. I am grateful for his kindness and hope I can continue to illustrate the positive influence he has had on my life.

Next, I would like to extend my gratitude to the members of my dissertation committee, Professor Nicholas Carter, Professor Steven Lumetta, Dr. Scott Mahlke, and Professor David Padua. Their numerous insights, questions, and suggestions improved the quality of this work immensely. Also, I would like to thank Jim Pierce, Allan Knies, and Dan Lavery at the Intel Corporation; their suggestions had a strong influence on the directions of this work. Mateo Valero provided me with both insight on my research and many opportunities to advance my experience in the computer architecture field. Yale Patt was a great influence on my career. I have been fortunate to learn on many different levels from all of Yale's numerous experiences.

I feel extremely fortunate to have been part of the IMPACT research group. My thesis research truly would not have been possible without the support, hard work of the members of the IMPACT research group. Members of the group were always there to discuss ideas, debate solutions, practice talks, and develop software. I feel extremely fortunate to have been a part of this group. There are several group members that I wish to thank. David August, Ben-Cheng Chung, Rick Hank, Teresa Johnson, Brian Deitrich, John Sias, and John Gyllenhaal provided invaluable feedback on ideas, papers, and talks. Kevin Crozier, Jeff Monks, Ron Barnes, and Matt Merten made office life enjoyable with their thoughts and discussions. I would also like to specially thank David August and John Sias for developing the predicate-based optimization framework used within this thesis. Lastly, I would like to thank Rick Hank for being my mentor when I first entered the IMPACT research group. Rick served as a great model for me and I wish I had more time to learn from his experience and abilities.

Most importantly, I would like to acknowledge the support of my friends and family. Tammy Rohrbaugh Knoblock helped inspire me to begin the long but worthwhile journey of my development as a professor. Steve Gemelos, Larry Chambers, Marty McKenna, and Krishna Bandaru provided many good times and friendship throughout my life. Jenny Carman Johnson was an incredible influence on who I am today and who I will try to continue to be in the future. I would like to thank Margaret Carns Manuel, a dear friend who always supported me in everything I did. I share incredible connections with Jaymie Braun Oehler, Jake Janovetz, Qudus Olaniran, and Teresa Johnson, all of whom are close friends with whom I have shared some of the best experiences. I thank Jean-Michel Puiatti for simply being himself, a wonderful and amazing person with great insight on how to live life, and a person who continues to teach me about what is important in life. I would like to thank Gigi Gogliotti for being a great friend and always listening to me whenever I needed to talk. Likewise, Hillery Hunter helped restore my balance whenever I started to wander off the path of reason. Similarly, at those times, the charm and kindness of Marie-Pierre Lassiva-Moulin gave me a reason everyday to come into the office and work. Sanjay Patel restored my appreciation for intelligent well-rounded people in the engineering field, and it is near tragic that I did not get to meet him until late in my graduate career. The friendship I share with Scott Mahlke and David August is incredibly unique and has provided me with great experiences and laughter. I am incredibly thankful for the connections I share with Scott and David. One of the best parts of my experience at the University of Illinois was getting to know Sabrina Hwu and being part of the entire Hwu family. My life would not be the same without her support and friendship.

Finally, I would like to express my heart-felt thanks to my family. My mother, Ann, my sister, Melody, and my brother, Mel, gave me the encouragement and consistent support that I needed to make it through graduate school. My thesis is dedicated in part to the life I have completely shared with them. Lastly, I would like to acknowledge the love and memory of my father, Melvin Francis Connors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The performance of an application on a modern processor is dominated by the level of compiler optimization. Generally, optimizing compilers are successful at eliminating inefficiencies and redundancies within programs by performing iterations of analysis and optimization. Static analysis techniques allow a large scope of a program to be considered and increase the available optimization opportunities. The elimination of redundancies in programs at compile time can dramatically improve a program's execution time. Traditional compiler techniques such as constant propagation, common subexpression elimination, loop invariant code removal, conditional branch elimination, and partial redundancy elimination [1] eliminate program redundancy and improve the efficiency of a program. However, compiler optimization techniques rely on the detection of static redundancy, which requires the complete assertion that the computations be definitely redundant for all executions. As such, compiler techniques have no mechanism for capturing dynamic redundancy, redundancy occurring over a temporal set of definitions. As a result of the deficiencies of compiler optimization, based on various types of applications, empirical program behavior studies indicate that many instruction traces are dynamically executed with the same inputs, a form of redundancy known as *value locality* [2], [3], [4], [5]. Overall, optimizing compilers are only able to make existing programs operate more efficiently on processors, and they fail to reformulate the program and transform the original algorithm to the most effective form. As such, optimizing compilers eliminate only a small portion of the overall redundancy of program execution.

The performance of an application on modern processors is also highly dependent on the ability to execute multiple instructions per cycle. In order to enable their cost-effective performance potential, these processors demand that increasing levels of instruction-level parallelism (ILP) be exposed in programs. As a result, compilers must perform increasingly more aggressive analysis and global optimization. Nevertheless, current state-of-the-art compilers cannot expose the level of ILP necessary to eliminate the diminishing performance returns of high-issue processors. Ultimately performance becomes limited by the dependences of programs, and not the machine resources. Thus, one of the major challenges to increasing processor performance is overcoming the fundamental dataflow limitation imposed by data dependences. By reusing previous computation results, the dataflow limit can be surpassed for sequences of operations that are otherwise redundantly executed.

As the availability of silicon increases, it makes sense to develop techniques to intelligently reduce the amount of redundancy or value locality in program execution. Several other fundamental architectural mechanisms are based on exploiting similar aspects of program behavior. Branch predictors [6] often occupy a fairly significant portion of the chip area and are critical for reducing the branch misprediction stalls. Generally, branch predictors are developed to exploit the predictability and correlation of branch execution behavior. Similarly, cache memory systems [7], [8] seek to exploit the locality characteristics found in memory references. By storing a referenced item, caches exploit *temporal locality*, the tendency for that item to be re-referenced soon. Additionally, by storing multiple items adjacent to the referenced item, caches exploit *spatial locality*, the tendency for neighboring items to be referenced soon. The exploitation of value locality of the program execution may provide substantial improvement to application performance on modern processors.

This dissertation introduces compiler-directed computation redundancy elimination, an approach that allows the compiler to identify code regions whose computation can be reused during dynamic execution. In a compiler-directed approach, the compiler essentially partitions the program into a set of units, called reusable computation regions. The compiler may determine the regions based on profiling information or static formation techniques. The use of profiling information to determine region contents allows the compiler to select units that accurately reflect the dynamic behavior of the program and are more likely to benefit from reuse. As a result, the compiler may be able to detect the most likely sequences of reuse. Unfortunately, there are several drawbacks to profiling. As such, for the compiler-directed mechanism to be successful, it is necessary to statically estimate the reuse potential of program regions. This dissertation deals with both concepts of profile-directed and static-based computation region formation.

In the compiler-directed mechanism proposed by this dissertation, the instruction set architecture provides a simple interface for the compiler to communicate the scope of each reuse region and its live-out register information to the hardware. During run time, the microarchitectural components of the approach record the execution results of the regions for potential reuse. Explicit designation of computation reuse allows a large number of dynamic control, memory, and arithmetic instructions to be jointly considered for reuse in the compiler-directed framework. The fundamental responsibility for detecting reusable sequences of code is placed on the compiler and the basic action of reusing the results of the sequence of the code is enabled through hardware. However, variations in program behavior cause the static designation of computation regions to reduce the effectiveness of the compiler-directed hardware approach. For instance, program invariance and computation redundancy occur in phases during run-time execution, and compile-time reuse decisions need to adapt to program behavior and contention for the reuse resources. Similarly, program invariance is highly related to input sets and the modes of particular applications, thus requiring methods of controlling the deployment of the compiler selected regions. Dynamic management hardware structures can gather run-time execution behavior and enable allocation policies of the computation reuse buffer. Within the dynamic framework, run-time decisions can determine which compiler selected computations are currently best exploited using the processor's reuse resources.

Many of the computation regions exploited through the compiler-directed hardware have persistent value locality characteristics that can be exploited using compiler transformation and existing architecture support. In the compiler-based value optimization framework, predicated-execution-based program transformations are used to exploit the redundancy of regions. These techniques offer venues for the compiler to synthesize codes based on value distribution within processors that do not offer hardware support for computation reuse.

This dissertation proposes and investigates the coordinated efforts of architectural and compilation techniques to eliminate dynamic computation redundancy of program execution on an ILP processor. An integrated compiler and architecture redundancy elimination framework has been implemented within the IMPACT compiler framework to quantify the benefits of exploiting value locality characteristics using both compiler techniques and hardware technologies.

1.1 Contributions

To integrate both compiler and hardware techniques in a coordinated framework to eliminate large amounts of dynamically redundant computation, four key technologies were developed. These four technologies, discussed below, together form the contribution of this work.

- **The Reusable Computation Region Framework (RCRF)**

  In order to apply compiler-directed dynamic elimination of redundant computation, the primary issue of determining the regions of a program that should be selected for reuse at run time must be addressed. Under this framework, the compiler partitions the program into *reusable computation regions*, which serve as the fundamental unit of execution in the integrated approach. Effective exploitation of dynamic redundancy requires that the compiler either be aware of the execution behavior of the program or estimate the program behavior. As such, two region formation approaches, *profile-based* and *static*, are developed. A quantitative analysis is performed to assess the characteristics of the reusable partitions selected using static heuristics and profile-based heuristics.

- **Compiler-directed Computation Reuse Approach (CCR)**

  A new approach to computation reuse is introduced and investigated. Previous approaches to exploiting value locality have relied strictly on hardware support to detect, exploit, and adapt to program value locality opportunities. Hardware-only techniques generally illustrate limited performance improvement because run-time techniques have limited scope in exploiting dynamic redundancy between program instructions, and hardware structures cannot cost effectively detect program redundancy. In the compiler-directed computation reuse approach, the compiler divides the program into regions, which eliminates reuse detection at run time and implements efficient reuse. The rationale for compiler-directed computation reuse is demonstrated, and the hardware framework for implementing compiler-directed reuse is presented.

- **Dynamic Computation Management System (DCMS)**

  An adaptive mechanism for bypassing computation regions based on the detected reuse patterns and utilization is developed. Run-time monitoring prevents regions with less likelihood of computation reuse from displacing computation regions with a greater likelihood of reuse in their use of hardware support resources. Both quantitative and analytical evaluations of the performance of reuse bypassing using the framework developed in this dissertation are presented.

- **Compiler-based Value Optimization Framework (VOF)**

  Traditional compiler techniques for eliminating program redundancy rely on the detection of static redundancy, which requires the complete assertion that the computations be definitely redundant for all executions. Generally such techniques focus on removing the redundancy of computations already performed by other instances of computations in the control flow graph. In compiler-based value locality techniques, program reformulation, control flow transformation, and predicate-based value transformations are used to exploit redundancy. These techniques offer alternatives for the compiler to synthesize value-based codes rather than the traditional generation of program. The concept of each technique is explored, and the compiler infrastructure for implementing compiler-based value locality exploitations is presented.

1.2 Overview

This dissertation is composed of seven chapters. Chapter 2 presents an overview of the empirical redundancy characteristics of program execution and discusses related approaches to exploiting computation redundancy.

The Reusable Computation Region Framework (RCRF) is detailed in Chapter 3. Region selection provides an effective framework to investigate the exploitation of computation redundancy in an integrated compiler and architecture approach. This chapter discusses the complexities of identifying reusable computation with profiling information and with static program heuristics. Chapter 4 presents the essential hardware design and instruction set architecture extensions for the compiler-directed approach. Compiler-directed computation reuse provides an effective framework to convey static analysis of dynamic redundancy to the hardware. This chapter discusses the implications of this concept for the design of an ILP microarchitecture.

Chapter 5 investigates the application of run-time management system to the base model of compiler-directed computation reuse. Dynamic management provides an enhancement to the adaptability of the base model, and the necessary requirements of the system will be investigated. Chapter 6 examines the exploitation of recurrence behavior of computation regions using compiler-based value locality transformations. Chapter 6 provides an overview and experimental evaluation of the systematic coordination of the proposed compiler and hardware techniques to eliminate dynamic computation redundancy. A quantitative comparison is done to investigate the effectiveness of value locality exploitation in compiler-based, hardware-based, and compiler-directed approaches. Finally, in Chapter 7, conclusions and directions for future research are given.

CHAPTER 2

REDUNDANCY OF PROGRAM EXECUTION

Modern microprocessor architectures are increasingly designed to exploit aspects of program behavior. Since the introduction of virtual memory and cache memory systems, architectures have been exploiting spatial and temporal locality of memory access patterns to reduce the average latency of a memory reference. Similarly, the designs of branch prediction techniques focus on extracting branch predictability and correlation for improving the handling of branches within pipelined processors. Another program behavior is computation redundancy, or value locality, which describes the recurrence of a previously seen computation instance.

The importance of computation reuse is not limited to the frequent existence of computation redundancy in real-world applications. Researchers in many different scientific domains have recognized the importance of computation reuse. General techniques to code algorithms to use memoization of computation results to achieve efficient processing have been studied. Nevertheless, development of automatic computation redundancy elimination techniques has not been integrated into modern programming and computing environments. This thesis provides a initial method of establishing the automatic exploitation of computation redundancy in modern high-performance computing systems. To provide sufficient background for the techniques proposed in this thesis, this chapter outlines the general elements contributing to computation redundancy in modern computer systems. Then, an empirical analysis of the value locality occurring in general-purpose C programs evaluated using the IMPACT compiler is presented to demonstrate the applicability and potential of this work. This chapter concludes with a brief survey of proposed approaches to exploit computation redundancy.

2.1 Dynamic Redundancy

The term *value locality* is used to describe the behavior of computations that execute with frequently recurring sets of values. Several empirical studies indicate the presence of significant amounts of value locality, or redundancy, in programs [5], [9], [10]. Traditional compiler techniques such as constant propagation, common subexpression elimination, loop invariant code removal, conditional branch elimination, and partial redundancy elimination reduce program redundancy and improve the efficiency of a program. These techniques are based on the detection of redundancy between computations of the program. As a result, current compiler optimization techniques can merely eliminate inefficiencies in a program by systematically detecting patterns within dataflow analysis of computations. Overall, compiler techniques are conservative and inflexible due to complexities associated with global program analysis [11].

Dynamic redundancy is a form of redundancy that occurs when input operands being evaluated are identical to previously evaluated inputs in an arbitrary temporal set of executions for the same computation. Figure 2.1 illustrates the occurrence of dynamic redundancy for a given history of a particular computation. Generally, the degree of dynamic redundancy occurring is determined by recording a history of unique evaluations and calculating the number of repetitions divided by the total number of observed executions. Compiler techniques have no general mechanism for capturing dynamic redundancy and as a result, even highly optimized programs execute half of their instructions redundantly, only to recompute previously computed values. Ideally, these values should be recorded and later reused, removing unnecessary computations.

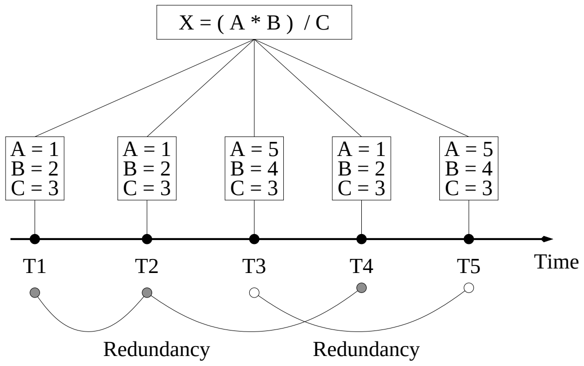


Figure 2.1 Illustration of dynamic computation redundancy.

2.2 Sources of Value Locality

There are several sources of computation redundancy within programs. Sodani and Sohi [5] characterized the sources by four categories. The fundamental sources are based on aspects of the input data, programming model, application domain, and the software distribution model. The work of this thesis does not specifically target any one source of redundancy; instead many elements of the proposed techniques are capable of eliminating redundancy from all of the base value locality sources. Naturally, many of these sources are subject to the particulars of an instruction set, compiler, and run-time environment being employed.

2.2.1 Data redundancy

Frequently, the input sets for real-world programs contain data that has little variation. Examples of this are sparse matrices, low frequency images, text files with white space, and repeated data entries in spreadsheets. Other aspects of data redundancy contributing to computation redundancy relate to the quantization and scaling of the input data at different points within a program.

2.2.2 Programming models

Studies [12] indicate that redundancy may exist primarily for the same reason that *partial evaluation* is such an effective compile-time optimization: real-world programs, run-time environments, and operating systems incur severe performance penalties because they are general by design. Most modern programming environments are implemented to handle numerous contingencies, exceptional conditions, and erroneous inputs, all of which occur relatively rarely and contribute to redundant run-time execution.

2.2.3 Application domain

Various types of applications by the nature of their design include run-time constants and semi-invariants. Examples are interpreters, simulators, and graphics processing. For interpreters, the program being interpreted is a run-time constant. For simulators, circuit layout parameters, and architecture resource constraints are examples of run-time invariants. Finally, for graphics processing, the scene and viewing parameters are semi-invariant.

In addition to the methods of programming a particular application, each application domain can be characterized by a degree of recurrence for certain overlapping subtasks within its implementation. Program coding recurrence of this form has been studied in its relation to the principles of dynamic programming [13]. Dynamic programming, similar to divide-and-conquer methodologies, solves problems by combining solutions to subproblems. However divide-and-conquer algorithms partition problems into independent subproblems, solve the subproblems recursively, and combine each solution to solve the original problem. In contrast, a dynamic-programming algorithm solves every subproblem exactly once and then saves the answer in an effort to avoid the work of recomputing the answer every time the identical subproblem is encountered. Whenever a problem exhibits overlapping subproblems, it meets a primary requirement of being solvable with a dynamic-programming algorithm. As such, since many applications created in modern programs exhibit repetition of subproblems, there exist significant amounts of redundant computation and a tremendous opportunity to exploit the principles of dynamic programming of the implicit but uncoded underlying algorithm.

Another aspect of application domains that contributes to dynamic redundancy is the different behaviors observed during execution. Virtually all programs go through a series of stages and each application domain can be described by how redundancy exists when the working set changes, either for the code or for the data, and sometimes for both. Generally an application may have different modes which operate on the data set in different ways and for different periods of time.

2.2.4 Software distribution model

Although current trends in compiler technology help reduce the amount of redundancy within programs, even code that is aggressively optimized by modern, state-of-the-art compilers exhibits redundancy. In short, substantial barriers to compiler optimization occur due to the limited scope of program analysis and optimization. The compiler must often be conservative and the result is code that includes large amounts of redundant computation. For instance, when disambiguating stores and potentially aliasing loads, a compiler must frequently generate redundant loads to resolve a potential alias. Overall, enhancing the level of program analysis appears to be a fundamental theme in overcoming the barriers to removing redundant instruction execution with compiler and hardware techniques.

Other aspects of the high performance computing market add to the occurrence of redundancy, the most dominant being a trend in software distribution. Proliferation of ubiquitous computing and interchangeable software is driving the use of Dynamically Linked Libraries (DLLs), downloadable updates, and downloadable net applications. Such models of distribution act as fundamental barriers to traditional compilation by eliminating some availability of program analysis and decreasing the ability estimate program behavior. The distribution methods increase the potential of the underlying processor executing invariant and semi-invariant codes. Ultimately in these models, performance is sacrificed for maintainability.

Related to software distribution are the inherent software portability aspects such as procedure calling conventions, stack frame orientation, and caller/callee register assignment. These items provide standardized software portability features but frequently sacrifice overall performance by contributing to program redundancy [5].

2.3 Value Locality Characteristics in Programs

There are several important metrics that can be used to describe the inherent value locality of a program. Value locality characteristics are important as they can determine which optimization technique can most effectively be applied to eliminate the redundant execution. For compiler optimizations, the invariance of a variable, the most frequent values of the variables, or the frequent set of values for a variable over the lifetime of the program are all important execution characteristics. In this section, a straightforward approach to value profiling is discussed. The study concentrates on instruction-level profiling to motivate the observed redundancy occurring in large regions of program execution.

2.3.1 Value profiling

Value profiling [2] can be used to automatically guide compilation and optimization. To determine code regions that have variables with semi-invariant or recurrent behavior, three types of information are important: (i) how an instruction's result varies over the lifetime of the program, (ii) the top N results values for an instruction, and (iii) the reuse history of instruction execution.

The invariance of an instruction's resulting value can be calculated in many different ways. The *Most Recent Value-M* (MRV-M) metric, defined by Calder et al., tracks the number of times an instruction's destination register was assigned a value that was one of the most recent M values. Where M is the history depth of the most recent unique values executed and defines the value profiling environment in which an instruction's invariance is being studied. The *Invariance-X* of an instruction-level locality characteristic indicates the percentage of execution time that the top X values occurred during profiling. The MRV characteristic differs from the invariance because it does not have state associated with each value indicating the execution occurrence.

The value profiling information required for determining exploitable semi-invariant code region behavior also requires the repetition of instruction computation instances. A history depth of unique instances of register source operands encountered during an instruction's execution can be used to calculate the overall redundancy of arithmetic operations in the program execution. Similarly, the repetition of values in the memory state can be tracked by maintaining the history of accesses for each memory instruction and a database of all accesses to memory.

2.3.2 Reuse Profiling System (RPS)

Although value profiling provides insight about the values occurring during execution, the nature of estimating a repeatable sequence requires further collection of the run-time behavior of a program. As such, the Reuse Profiling System (RPS) was developed for this thesis. Figure 2.2 shows a diagram of the information maintained by the profiling infrastructure developed in this thesis. The foundation of the reuse profiling infrastructure is the value profiling of source and destination register operands. The infrastructure generally performs reuse profiling of program execution by enabling three individual profiling methodologies: value profiling, computation reuse profiling, and memory reuse profiling. In addition, region reuse profiling can be enabling by annotating regions within code prior to profiling. Region reuse profiling determines the value, computation reuse, and memory reuse execution behavior for a region of code rather than a single instruction.

The value profiler keeps a top$N$ value (TNV) table for each register operand. The TNV table records (V-value, E-execution) pairs for each entry with a least frequently used (LFU) replacement policy. When inserting a value into the table, existing entries are incremented. If a value is not found, the least frequently used entry is replaced. A least recently used (LRU) value replacement policy does not account for the execution occurrence of a particular value. A modified LFU policy with fixed and temporary entries is used. The temporary entries are used to contain the most recent unrecorded values and are periodically cleared to allow new entries to be collected.

To determine the repetition of computation execution, the computation reuse profiler records instances of all the source operand values of a profiled instruction within a history buffer with$N$ entries. The entries maintain (CI-computation instance, E-execution) pairs with an LRU

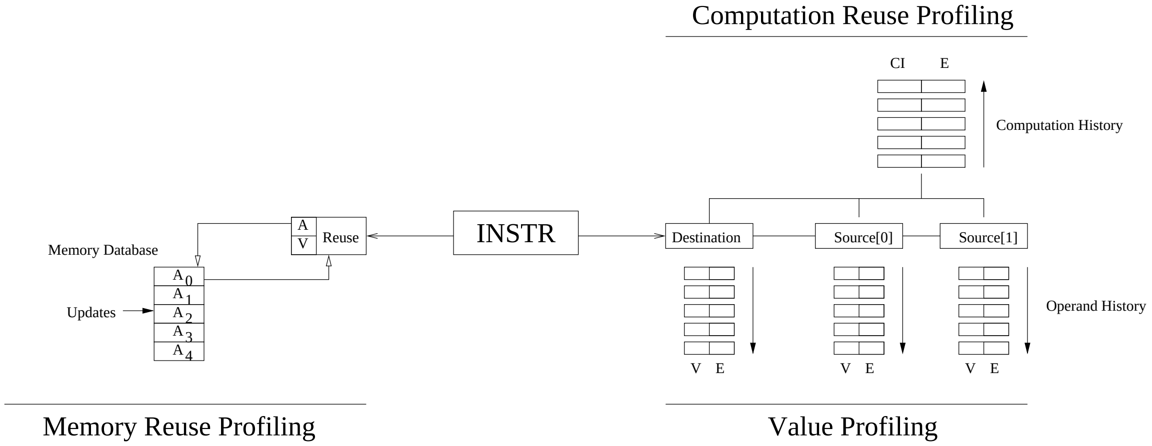


Figure 2.2 Reuse profiling: value, computation reuse, and memory reuse profiling.

replacement policy. When accounting for matching computation instances, the specific values of the source operands are not recorded for final determination. Instead, only the reuse statistics observed within the computation history are maintained. In addition to the history buffer, an additional computation instance entry records the last computation entry set. This entry accounts for redundant computation that matches identically with the very last computation execution, allowing value locality that occurs in streaks or phases to be detected. Finally, to determine the activity of the history buffer, for every period of 1,000 computation instances, the number of buffer entries attributing more than 5% of the observed execution instances is recorded. This measure indicates whether a dominant number of computation instances exists for a computation instruction.

Conceptually, the memory reuse profiling approach is based on an address-based hash table system that records a database of references for each accessed datum. The memory addresses of load and store operations in a profiled program are extracted and used as the hash keys for a single global hash table. At the same time, information is maintained locally for the instruction references memory. The local information consists of A-address, V-value of data, and the *Reuse* time. The addresses of the hash table entries correspond to specific data items, such as variables or array elements, in the executing program. Since the goal of memory profiling is to associate conflicts between load and store operations, it is necessary for each hash entry to maintain the unique ID for the last store operation that modified the respective program address. When load operations execute, their memory addresses access the hash table and find the respective hash entry. This entry contains the store ID of the last conflicting store operation, and is used to construct a conflict between the load and store operations. Conflicts between operations are maintained in the local instruction structure and are updated for each occurring conflict. Previous work [14] developed a similar address hash structure for evaluating data dependences to direct data speculative optimization and scheduling algorithms.

The region reuse profiling can be intuitively described as emulation of a designated region of code as a single instruction. Live-in source register values, computations instances, and memory reuse can be collected for the annotated region. Region annotations can be obtained using profile-guided techniques or static analysis. The value profiler can designate a trace of instructions with high individual instruction reuse for region reuse profiling. Further description of the computation region formation process is given in the next chapter.

Static analysis derives regions consisting of simple loops, capturing potential redundancy for linked-list traversals and array scans. When statically annotating regions, a defined set of control blocks composes a region. Formally, a function can be represented by a control flow graph $G(N, E)$, where N is the set of basic blocks in the function and E is the set of all control flow edges between blocks in N. The graph can be divided further to represent the individual instructions that define basic blocks by the graph $GI(I, E)$, where I is the set of all instructions in the function and E is the set of all control edges. The difference between graphs G and GI is that GI uses implied edges between the sequential instructions of a basic block N. The graph GI is trivially created from G by designating every instruction as a basic block.

For each reuse region, profiling instrumentation is inserted to verify reuse at the start of each region. However, this process involves executing the region to initiate the path of executions and check the register values and memory database. Thus in validating region reuse, both the memory region reuse rate and rate that an individual load causes the invalidation of a loop execution can be determined. During stateless region detection, loops representing memory regions are annotated for the profiling system to gather the memory region reuse information.

2.3.2.1 Evaluation methodology

The benchmarks used in the experiments of this evaluation and the entire thesis consist of 13 nonnumeric programs: two of the SPECINT92 benchmarks, *008.espresso*, *072.sc*; seven of the SPECINT95 benchmarks, *099.go*, *124.m88ksim*, *126.gcc*, *129.compress*, *130.li*, *132.ijpeg*, *147.vortex*; two Unix utilities, *lex*, *yacc*; and two media utilities, *mpeg2*, *pgp*. These benchmarks were selected to represent the characteristics of a wide range of engineering, communication, and desktop applications. Table 2.1 describes the functionality of the applications studied and is cited to develop comparisons between the program behavior and reuse characteristics of the different application types.

To throughly study the value locality characteristics of programs, profile information was collected for two series of inputs, training and reference. Generally, a training input is selected to indicate the program elements that are most characteristic of the application behavior. The data input sets used for all experimental evaluations in this thesis, and the respective number of dynamic instructions (in millions) executed for each data set are described in Table 2.2.

Table 2.1 Benchmarks descriptions.

Benchmark	Description
008.espresso	Optimization program for Programmable Logic Arrays.
072.sc	A mathematical spreadsheet program.
099.go	An internationally ranked go-playing program.
124.m88ksim	A chip simulator for the Motorola 88100 microprocessor.
126.gcc	The GNU C compiler, version 2.5.3.
129.compress	An in-memory compression application.
130.li	Xlisp interpreter.
132.ijpeg	Image compression/decompression on in-memory images.
147.vortex	An object oriented database.
lex	A lexical analyzer generator.
yacc	A parser generator for context free grammars.
mpeg2	Standard for digital video transmission.
pgp	128-bit digital signature cryptography.

Table 2.2 Data input sets used in evaluating locality.

	Data Input Set 1 : Training		Data Input Set 2 : Reference	
Benchmark	Name	Exec M-instr	Name	Exec M-instr
008.espresso	bca.in	307.8	cps.in	369.7
072.sc	loada1	86.8	loada2	64.7
099.go	2stone9.in	292.9	5stone21.in	11122.6
124.m88ksim	dcrand.big	67.8	dhry.big	46550.2
126.gcc	amptjp.i	855.0	varasm.i	16301.2
129.compress	test.in	23.8	bigtest.in	27835.4
130.li	train.lsp	114.3	reference	35611.2
132.ijpeg	vigo.ppm	1072.3	penguin.ppm	22000.3
147.vortex	persons.250	1082.1	persons.1k	32806.5
lex	lex.l	37.8	awk.lx.l	35.5
yacc	c-parse.y	33.8	awk.g.y	15.0
mpeg2	mei16v2.m2v	94.7	reference	974.2
pgp	pgptest.plain	43.8	pgptext.pgp	54.4

The amount of time for a reuse profile to be generated plays an important role in the integration of an architecture mechanism or compiler technique into a traditional high-performance computing system. Table 2.3 shows the time of the value profiling infrastructure. The second column indicates the amount of user and system time spent performing control profiling a program's execution. The third column indicates both the time spent performing value profiling and the factor of evaluation time increase relative to the time of control profiling. The fourth column shows the both the time spent performing reuse value profiling with memory reuse profiling and the factor of evaluation increase relative to the time of control profiling.

The amount of time spent in developing, verifying, and releasing software will vary depending on the level of the potential benefits achievable. Although the times for reuse profiling in Table 2.3 indicate considerable expense to the program development time, the estimated benefits in program execution significantly outweigh the costs. Methods of reducing value profiling time by taking advantage of value locality properties have been considered [2].

Table 2.3 Control profiling, value profiling, and reuse profiling times.

	Control Profiling (CP)	Value Profiling (VP)		Reuse Profiling (RP)	
Benchmark	Time (s)	Time (s)	(VP/CP)	Time (s)	(RP/CP)
008.espresso	156.5	231.4	1.47	295.6	1.88
072.sc	78.5	114.6	1.46	120.9	1.54
099.go	202.2	306.4	1.51	280.8	1.38
124.m88ksim	134.5	184.9	1.37	172.5	1.28
126.gcc	944.4	1156.7	1.22	1395.4	1.47
129.compress	29.2	72.3	2.47	181.9	6.23
130.li	50.7	89.0	1.75	165.8	3.26
132.ijpeg	152.0	193.9	1.27	319.8	2.10
147.vortex	326.0	378.9	1.16	801.9	2.46
lex	22.1	66.7	3.01	163.4	7.39
yacc	18.8	47.4	2.51	158.2	8.41
mpeg2	28.7	111.7	3.89	323.0	11.25
pgp	68.8	167.1	2.42	204.6	2.97

2.3.3 Invariance

This section examines the invariance of values for instruction types and procedure parameters. For the results, a TNV table size of 10 values was used for profiling. When an instruction or parameter is said to have an *Invariance-M* of X, this is calculated by taking the number of times the top M values occurred during profiling, as found in the TNV table after profiling, and dividing the number of times the instruction was executed. Table 2.4 shows the percentage of execution matching the *Inv-10* invariance characteristic and the percentage of dynamic execution frequency for different individual instruction types.

Table 2.4 includes instruction categories: integer loads (I-Load), integer stores (I-Store), integer arithmetic (I-Arith), integer multiplication (I-Mul), integer logical (I-Logic), floating-point loads (F-Load), floating-point arithmetic (F-Arith), and all other floating-point operations (F-Op). The results show that integer invariance is a common behavior in the programs evaluated. For all programs, the integer loads and the integer arithmetic instructions consistently have a high degree of invariance and account for the major portion of program execution. The floating-point instructions, although having a high degree of invariance, are not frequently executed. These results indicate that dynamic computation redundancy is concentrated in integer operations and motivates the development of a mechanism to eliminate redundant code execution in nonnumeric programs.

Many compiler optimizations exploit the invariance commonly exhibited by variables during program execution. Load instructions represent the mechanism by which variables that reside in the memory system are transfered to the processor. Due to the growing disparity between processor and memory performance, load instructions occasionally incur increasingly long latencies. These latencies delay the execution of dependent instructions, which can significantly

Table 2.4 Percentage execution invariance and percentage dynamic program execution for individual instruction types.

Benchmark	I-Load	I-Store	I-Arith	I-Mul	I-Logic	F-Load	F-Arith	F-Op
008.espresso	55 (27)	73 (14)	48 (32)	28 (0)	57 (0)	80 (0)	50 (0)	98 (0)
072.sc	42 (26)	48 (14)	49 (26)	34 (0)	63 (6)	78 (0)	0 (0)	50 (0)
099.go	35 (32)	70 (15)	30 (35)	20 (0)	45 (0)	100 (0)	0 (0)	97 (0)
124.m88ksim	75 (22)	76 (12)	65 (30)	35 (0)	69 (0)	60 (0)	100 (0)	100 (0)
126.gcc	43 (21)	36 (12)	43 (26)	39 (0)	56 (9)	67 (0)	0 (0)	0 (0)
129.compress	42 (22)	14 (6)	14 (34)	10 (0)	16 (11)	0 (0)	0 (0)	0 (0)
130.li	40 (30)	59 (15)	55 (25)	30 (0)	70 (3)	90 (1)	0 (0)	100 (0)
132.ijpeg	25 (20)	22 (5)	22 (35)	35 (3)	18 (22)	75 (0)	0 (0)	90 (0)
147.vortex	55 (27)	60 (12)	60 (30)	10 (0)	35 (5)	79 (0)	0 (0)	0 (0)
lex	38 (27)	42 (10)	22 (34)	15 (0)	30 (6)	50 (0)	0 (0)	48 (0)
yacc	52 (25)	50 (11)	45 (30)	25 (0)	52 (5)	50 (0)	33 (0)	67 (0)
mpeg2	32 (19)	40 (9)	38 (18)	13 (0)	35 (3)	56 (0)	0 (0)	33 (0)
pgp	55 (22)	58 (10)	51 (31)	27 (1)	51 (9)	65 (1)	45 (0)	82 (2)

affect system performance. For these reasons, the invariance of load values is an important characteristic to exploit in both compiler-based and hardware-based techniques. In order to look at the overall invariance for an instruction, several degrees of invariance were examined. For *Inv-1*, the frequency count of the most frequently occurring value in the TNV table is divided by the number of instruction executions. For *Inv-X*, the occurrences of the top X values in the final TNV table are added together and divided by the number of instruction executions to generate the percentage invariant execution. Figure 2.3 shows the invariance for integer loads for the top five values. The difference in *Inv-1* and *Inv-5* invariance of load instructions for most programs is about 10% of the load execution. The results indicate that benchmarks *124.m88ksim*, *147.vortex*, and *pgp* have an *Inv-5* invariance that accounts for upwards of 65% of the overall instruction execution. Overall the high percentage of load execution accounted by a relatively small number of values indicates that potentially recording a limited set of values for each instruction can amount to significant performance improvement.

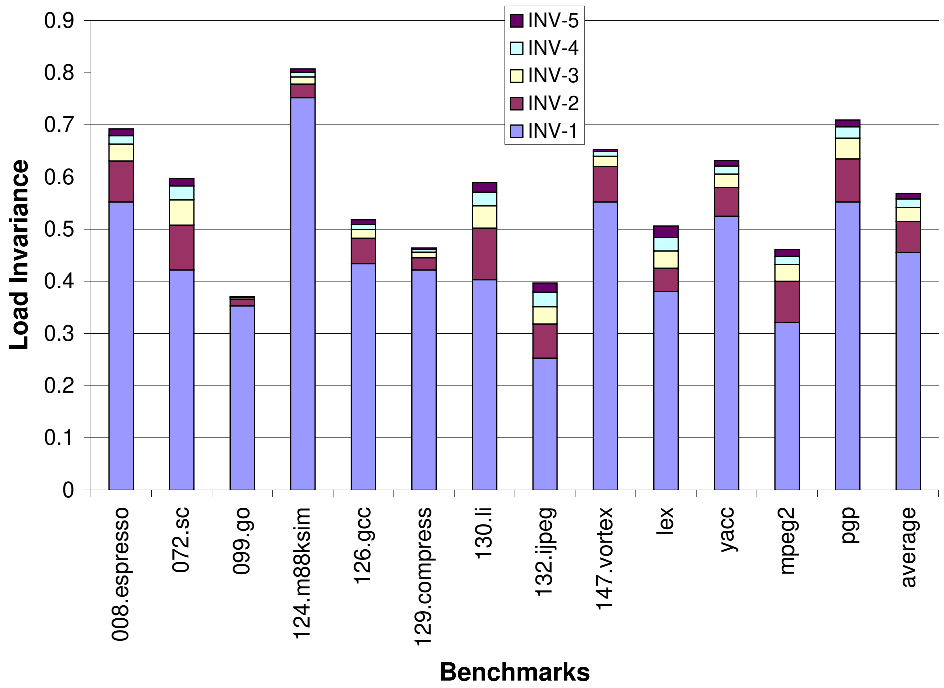


Figure 2.3 Evaluated load invariance of top five values.

Another source of redundancy in programs that can potentially delay the processing of instructions is the incoming parameter variables for a program procedure. Customizing procedures based on parameter specific values is a beneficial form of compilation [15]. This compiler specialization can be successfully used when a small set of values occurs frequently for a procedure. Figure 2.4 shows the percentage of function calls with complete parameter repetition for the five most frequent sets of call contexts. According to the results, specializing procedure calls based on the the most frequently occurring sets could capture an average of 30% of the procedure call invocations. In the case of *pgp*, a significantly higher opportunity to affect program performance is possible since upwards of 50% of the procedure calls are made with the most frequent arguments. Generally the results indicate that a large amount of redundancy exists at the boundary between procedure calls and that any advanced technology that attempts to eliminate computation redundancy will require some element of interprocedural analysis.

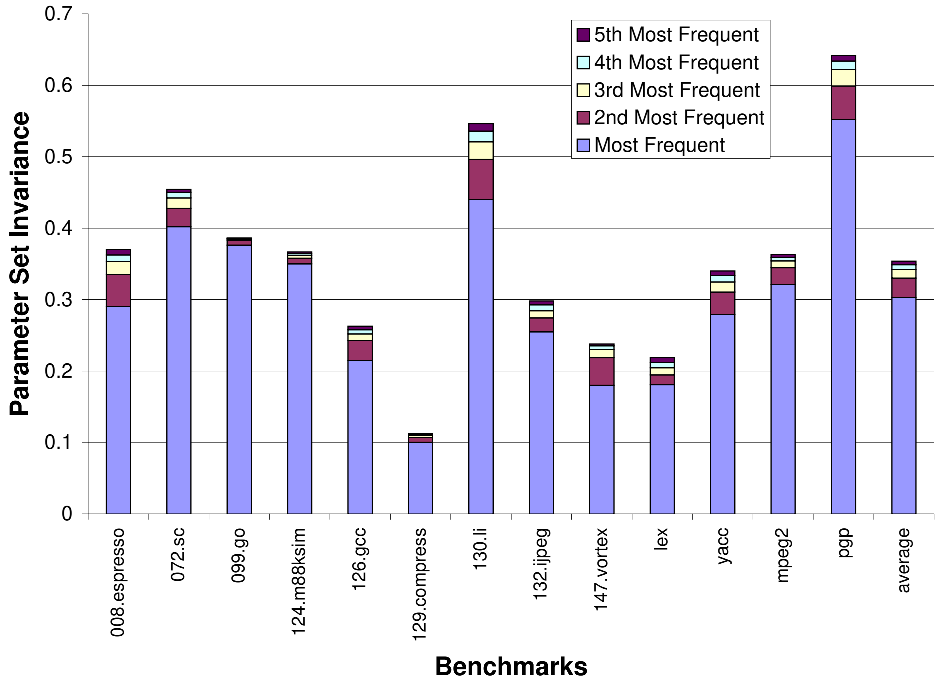


Figure 2.4 Distribution of parameter repetition due to top five most frequently executed sets.

2.3.4 Instruction reuse

The percentage of dynamic instructions reused for three different variations in the number of computation instances maintained for each instruction execution is presented in Figure 2.5. The reuse history sizes are varied between 8, 16, and 32 entries. Even for the small number of entries, the amount of instruction reuse that can be captured is significant, averaging nearly 25% of overall instruction execution. Nearly 40% of instruction execution is reused for computation instance history capacity with greater than 32 entries. These results assume that each instruction will maintain a history of the previous computation executions. Memory instructions in this evaluation are reused if the memory contents accessed during their execution have been previously accessed in their recent history and not referenced by any store instructions.

A valuable evaluation is to determine the observed reuse behavior of a program when a table of instruction information limits the reuse collection process to a set number of instruction

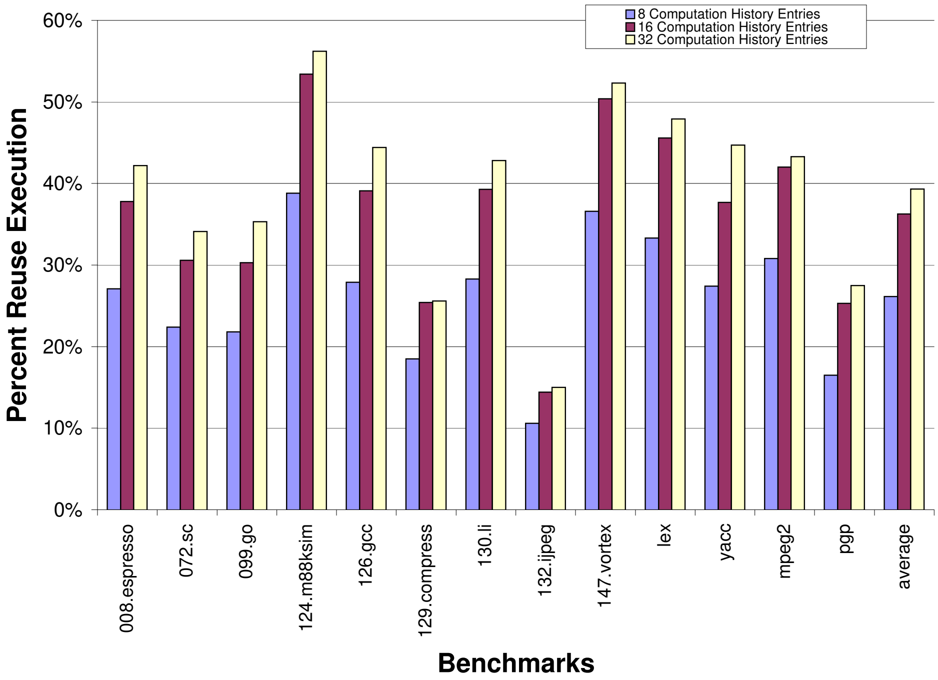


Figure 2.5 Percentage of instruction execution reused for history sizes: 8, 16, and 32 entries.

entries. Figure 2.6 illustrates the effects of altering the instruction reuse collection between 32, 128, and 1024 entries. For all evaluations, a computation instance history size of 16 entries was used. As can be seen in comparing the results, the evaluation using 1024 closely approximates the results of Figure 2.5. Otherwise, when only a limited number of instructions can be observed, the overall percentage of program execution with reuse behavior detected significantly decreases. This indicates that instruction-level reuse exploitation is largely limited by having to spend considerable hardware resources to detect the components of the program with value locality characteristics. This is the primary motivation of the compiler-directed approach proposed within the next few chapters.

In motivating the proposed techniques of this thesis, it is important to determine the percentage of an application that exhibits computation redundancy and recurrence behaviors. Essentially by identifying whether a small portion of program code contributes to the observed

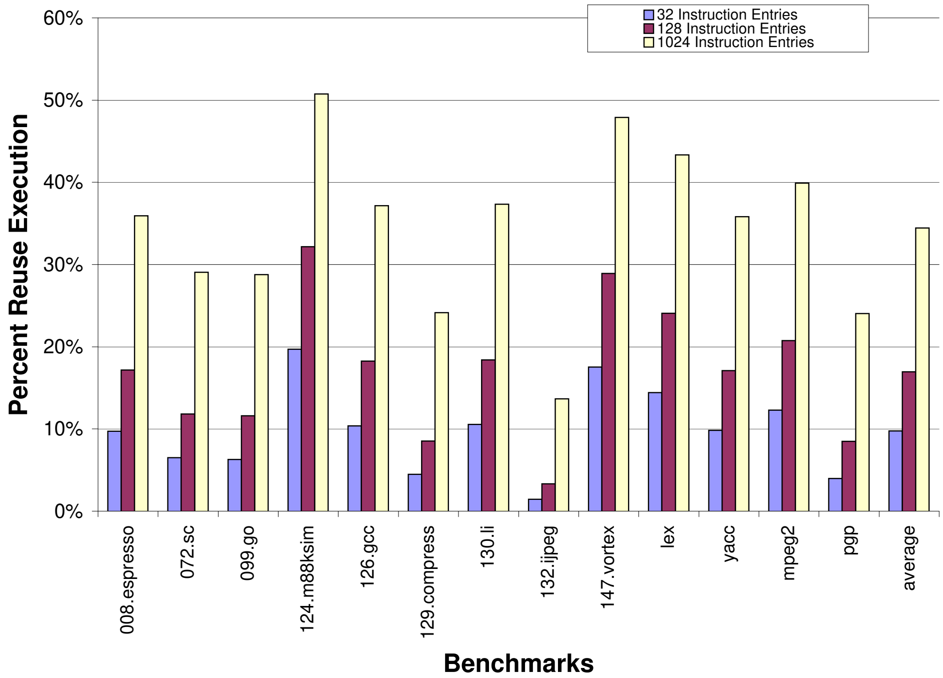


Figure 2.6 Percentage of instruction execution reused using reuse table sizes: 32, 128, and 1024 entries.

dynamic execution characteristics, a more effective mechanisms can be designed. Figure 2.7 illustrates the reuse execution based on the static distribution of the instructions that account for the highest level of reuse in program execution. On average, nearly 40% of the dynamic computations with reuse properties are being generated by 20% of the static program. The evaluation is performed while capturing individual reuse behavior for every instruction without contention in a reuse table. The distribution indicates that certain regions of program computation may have persistent redundant execution. In such regions, new architectural and compiler techniques may serve to exploit the characteristics of value distribution. In conjunction with the results of Figure 2.6, these results indicate that only a small portion a program may have value locality, yet limited instruction-level hardware table techniques employed to find the reuse behavior are likely to fail.

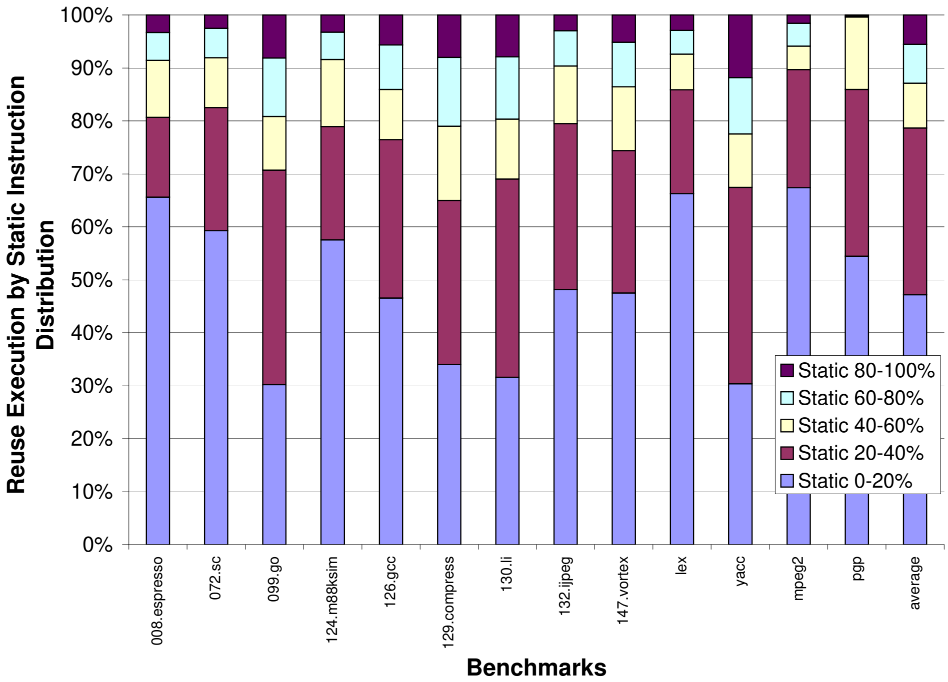


Figure 2.7 Reuse execution based on static distribution of instructions.

The effect of reuse on performance does not only depend on the percentage of reusable instructions, but also on instruction type. Although reuse of any instruction type will reduce the instruction fetch requirements of the processor, reuse of specific instruction types also creates other opportunities to improve processor efficiency. For instance, in the case of reuse of load instructions, data does not have to be retrieved from the memory system. Similarly, reuse of branch instructions reduces the demand for performing accurate branch prediction. Figure 2.8 shows the distribution of operation types (control, memory, arithmetic) within the regions possessing reuse potential. Clearly indicated by the number of control operations is the notion that any reuse mechanism must exploit instruction sequences that extend beyond the basic block.

In order to better understand the reuse behavior of memory values, the reuse property of loads was measured using two different aspects. Memory reuse measures the occurrence when

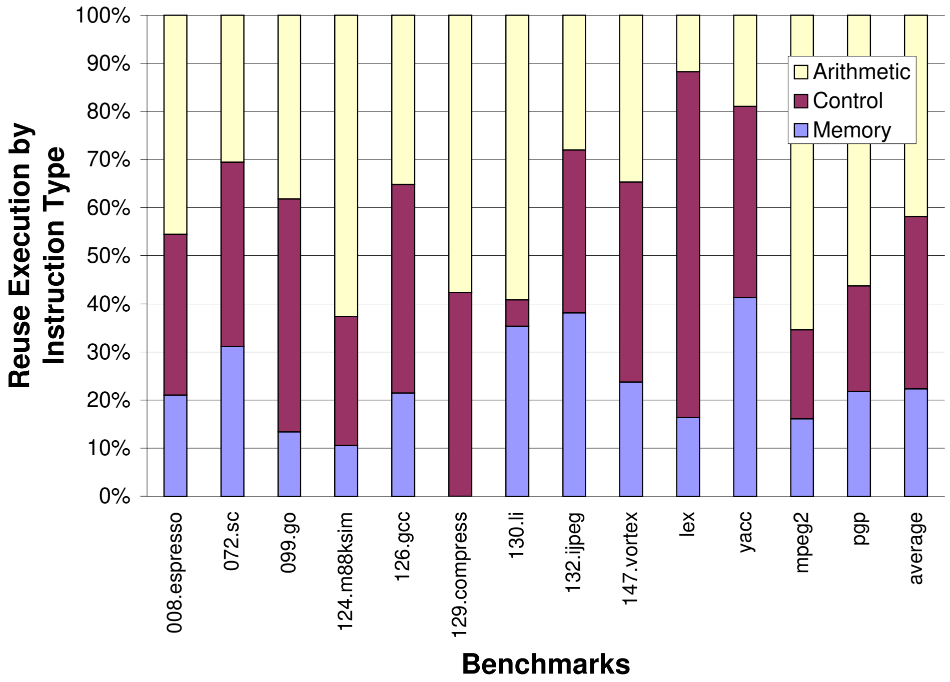


Figure 2.8 Reuse execution breakdown by instruction type: arithmetic, control, and memory.

memory references to an identical memory location by a particular static load are not interfered by intermediate store instructions. There is potential to reuse previous results since the memory location has not been modified. Memory/value reuse measures the same occurrence but allows intermediate store instructions to write to the respective memory location as long as the value of data is unchanged. Figure 2.9 reports the potential of eliminating load operations based on the two aspects, memory reuse and memory/value. Nearly 20% of load instructions do not require execution if the previous results can be recorded. This percentage can be substantially increased if the mechanism can track the updates of values in the memory system and communicate that previous computation results will be unchanged.

One of the most motivating aspects determined by reuse studies for developing new mechanisms for exploiting redundancy in program execution is the existence of redundancy behavior for many input evaluations. This occurrence indicates that the frequently observed repetition

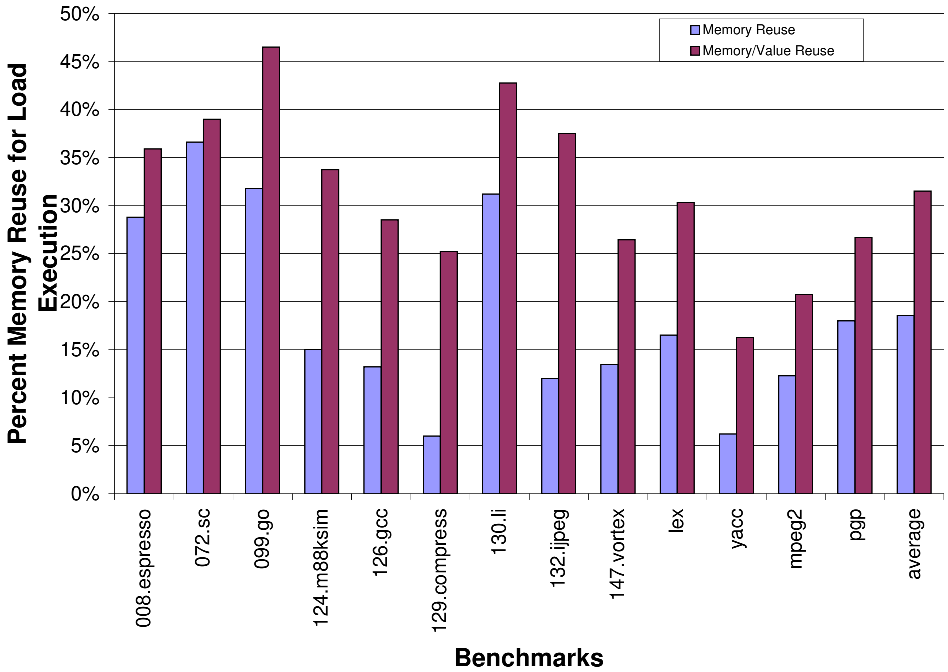


Figure 2.9 Reuse execution for load instructions based on memory reuse and memory/value reuse.

characteristics of programs can serve as focal points for developing new techniques to improve performance. Figure 2.10 shows the reuse execution for each program for two input evaluations, INPUT1 and INPUT2. Each input set evaluation has two variations (REUSE8 and REUSE16) in the number of history entries for each profiled instructions. The results show that a high degree of reuse is present in both input sets and that the locality behavior is not simply a solitary component of the training input. On average, instruction reuse can account for nearly 25% of the program execution. More detailed evaluation of the overlap of computation reuse in the input sets is presented in Chapter 3.

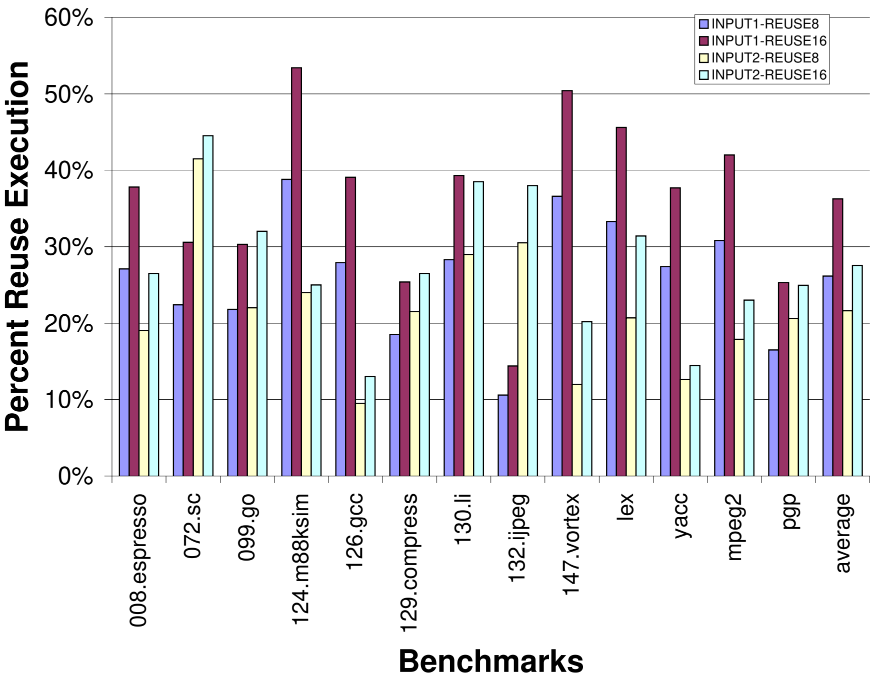


Figure 2.10 Reuse execution observed for multiple input sets (INPUT1,INPUT2) while varying the reuse computation history sizes (8,16).

2.4 Related Work

Previous research in the area of value locality and redundancy exploitation can be classified into four major categories: value prediction, dynamic instruction reuse, memoization, and dynamic compilation. Value prediction and dynamic instruction reuse are two important hardware strategies that attempt to reduce the execution time of programs by alleviating the dataflow constraints at the instruction level. Value prediction [16] speculates the results of instructions based on previous execution results, performs speculative computation using the predicted values, and confirms the speculation. Instruction reuse [17] recognizes that many instructions have the same inputs when executed dynamically, and that by buffering the previous results, future dynamic instances can avoid execution by simply using the saved result. Although alternative schemes include *dependence chains* of multiple instructions or use profiling information to guide the detection mechanism [18], the performance improvement of these proposed approaches is often limited by the exploitation of value locality at the instruction level [19]. In the *block* [20] and *trace-level reuse* [3] techniques, hardware mechanisms are proposed to exploit value locality for large straight-line sequences of instructions. These approaches detect that the inputs and outputs of a chain of instructions are highly correlated, and recognize that the inherent benefits of prediction and reuse only materialize when a large amount of execution is eliminated.

Another category of value locality exploitation research focuses on memoization. Generally, memoization is a technique that stores previous results of computation in memory, and later invocations are preceded by table lookups for already computed results. Functional and logic programs use software concepts of memoization, whereas the *Tree Machine (TM)* [21] and *result cache* [22] are hardware implementations of computation memoization. In these models, computation caching exploits value locality in the way that cache memory systems exploit spatial and temporal locality of memory accesses.

Dynamic compilation and run-time optimization are emerging directions for computer system research which provide improved execution performance by performing some aspect of compilation at run time. These techniques range from completing compiler-generated specialized templates at run time to fully adaptive code generation.

Run-time specialization enables applications to adapt to particular program inputs and actual run-time behavior. Run-time specialization holds the promise of tremendous performance improvements for some applications. Staged compilation is a compilation strategy in which the code-compilation process is completed in multiple stages: for example, at traditional compile time, at link time, at load time, and on demand at run time. By delaying a portion of the compilation process, it becomes possible to take advantage of information available only at the later stages, with the goal of improving performance of the resulting code.

The portion of compilation postponed until run time is called dynamic compilation. The principle challenge and trade-off in dynamic compilation is achieving high-quality dynamically generated code at low run-time cost, since the time to perform run-time compilation and optimization must be recovered before any benefit from dynamic compilation can be obtained. DyC [12], [23] is a dynamic compilation system in which the programmer annotates regions of the programs that should be compiled dynamically and variables for which to specialize the regions. A static optimizing compiler automatically identifies which data will have known values at run time, given the values of the annotated variables, then creates and optimizes machine-code templates. A post-pass uses these templates to automatically produce generating extensions that, when executed, produce optimized executable code at run time.

A key design issue in developing an effective dynamic-compilation system is the method for determining where, when, and on what run-time state to apply dynamic compilation. Current dynamic compilation systems rely on some form of programmer direction to indicate where dynamic compilation would be most profitably applied. Several dynamic compilation systems take a declarative, or transformational, approach, with user annotations guiding the dynamic compilation process [24], [25], [26]. Auslander et al. [24] proposed a dynamic compilation system that uses a form of binding time analysis to generate templates for code sequences that have been identified as semi-invariant. Their approach currently uses user defined annotation to indicate which variables are semi-invariant. Staging analysis has been proposed by Knoblock and Ruf [26] as an effective means for determining when computation can be performed early by the compiler and which optimization should be performed late or postponed by the compiler for dynamic code generation. The approach requires programmers to provide declarative hints to the staging analysis to determine which variables have semi-invariant behavior. In addition, Autrey and Wolfe have investigated a form of staging analysis for automatic identification of semi-invariant variables [27]. The Tempo system by Consel and Noel [25] uses partial evaluation techniques to automatically generate static templates that are selected at run time using dynamic code generation techniques.

Alternatively, Calpa [28] is a system that generates annotation automatically for the DyC dynamic compiler. Calpa utilizes execution frequency and value profile information to drive a program analysis based on a dynamic compilation benefit model. Finally, run-time optimization techniques such as Dynamo [29] and Daisy [30] are software systems that attempt to optimize running codes and program binaries by optimizing and placing the translated codes into a portion of memory for extended execution.

CHAPTER 3

COMPUTATION REGION FRAMEWORK

A key design issue in eliminating dynamic redundancy is enabling a system that can replace large amounts of run-time execution with the results from earlier executions. The fundamental steps of any method of computation redundancy elimination involve identification, exploitation, and adaption of the redundancy. Performing the identification of redundancy involves detecting the recurrence and understanding how that recurrence exists between instructions. As evaluated in the previous chapter, the collection of accurate program reuse characteristics is adversely affected by limited hardware budgets. Similarly, hardware approaches are severely limited in the level of analysis that can be performed at run time. Declarative approaches offer the advantages of an effective interface to identify invariant code regions since the programmer has inferences about the code behavior. However, current trends in programming environments indicate the growing demands on the compiler to perform various degrees of program analysis.

Generally a compiler can identify the critical points of reuse by employing profile information and static program analysis. Furthermore, by finding compile-time relationships among computations, more effective mechanisms of eliminating redundancy can be developed. This chapter investigates a code representation and compiler framework for using compiler-directed hardware techniques to eliminate computation redundancy and for guiding compiler program reformulation techniques for exploiting value locality.

3.1 Motivation

This work is motivated by the potential of computation reuse at granularities greater than the instruction level. From the results of the previous chapter, it can be concluded that instruction-level reuse is abundant and potentially contributes to the redundancy of large dynamic traces of instructions. Recent hardware solutions to ILP limitations have been proposed to reorder code blocks and store them into a special cache called the trace cache [31], [32]; are possibly identifying invariant run-time behavior manifesting as frequently executed paths through code. Reuse of large traces of code is an attractive technique since a single invocation of a reuse mechanism has the potential to eliminate a long sequence of dynamic instructions.

This section presents the intuitive rationale behind using the compiler to identify regions of program computation that exhibit run-time redundancy.

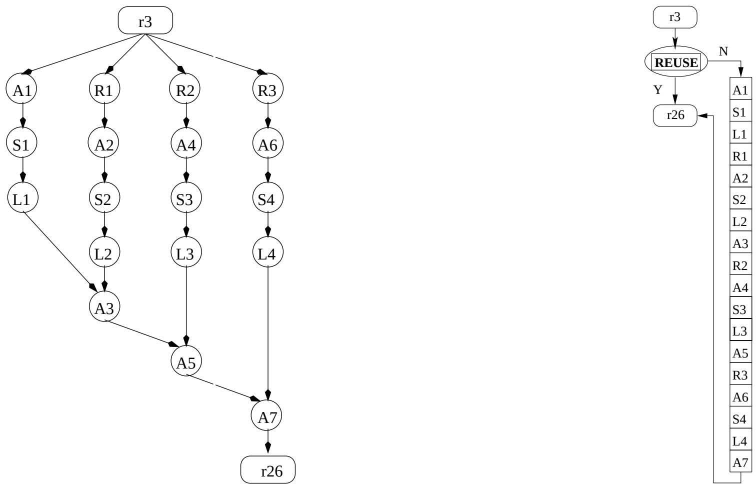
3.1.1 Reuse region opportunities

The goal of any computation reuse scheme is to minimize the execution time of computing results that have been previously produced. The following examples illustrate that an integrated compiler and architecture reuse approach has the potential to eliminate large sequences of dynamic instructions.

Trace-level reuse. Figure 3.1 represents an example from the SPECINT92 benchmark *008.espresso* that demonstrates the complexity of efficiently detecting sequences of reusable instructions. A macro definition for computing the number of bits set to logical 1 in a 32-bit word is shown in Figure 3.1(a). The macro divides the 32-bit word into four bytes and uses each byte as an integer index for the *bit_count* array. The array is defined to map the respective byte index into its corresponding number of logical 1 bits. The four components are then summed together. The dependence graph for this segment is shown in Figure 3.1(b) and uses the following instruction key: **A** for arithmetic/logical, **L** for load, **R** for right shift, **S** for left shift, **M** for move, and **B** for branch. In this case, all the code falls into one basic block because there is no possibility of branching until the end of the instruction sequence. The dependence graph illustrates that the entire sequence of operations is dependent on a single input register $r3$ and defines a single output register $r26$. No other registers defined in the sequence are live-out, that is, used after the computation sequence. Alias analysis can determine that the array *bit_count* is static and does not change during program execution. The instruction sequence clearly designates an opportunity for reusing previously computed results. Any execution that starts with a given value in $r3$ will generate the same computed value in $r26$.

```
#define count_ones(v)\
   (bit_count[v & 255] + bit_count[(v >> 8) & 255]\
   + bit_count[(v >> 16) & 255] + bit_count[(v >> 24) & 255])
```

(a)



(b) (c)

Figure 3.1 Trace-level reuse example source code macro definition (a), dependence graph (b), and potential reuse sequence (c).

Region-level and memory reuse. Several studies [3], [17] have determined the limits of instruction-level reuse by checking whether a dynamic instruction and its current inputs are the same as a previous execution. Such studies hinder the identification of the maximum reuse potential that is possible in improving performance given that each reuse attempt has an associated latency at run time. For instance, reuse at the instruction level may only be effective for long latency instructions such as divide, multiply, and remainder. Reuse of a sequence of instructions is more attractive since a single reuse may eliminate the execution of a potentially long sequence of dynamic instructions.

Reuse Theorem 3.1 Let T be an execution trace of dynamic instructions $< i_1, i_2, ..., i_n >$. Given a reusable instance of the trace T, then instruction i_k is reusable for every $k \in [1, n]$.

Reuse Theorem 3.2 Let T be an execution trace of dynamic instructions $< i_1, i_2, ..., i_n >$. Given a nonreusable instance of the trace T, instruction i_k may be reusable for every $k \in [1, n]$.

Theorem 3.1 implies that amount of trace-level reuse is limited by the amount of individual instructions that are reusable. The theorem is used to state an upper bound on the instruction execution elimination available through trace-level reuse. However, by noting that the overhead of trace-level reuse may be amortized by grouping instructions to the cost of an individual instruction, then there are clear benefits to trace-level reuse. Theorem 3.2 in coordination with Theorem 3.1 states that trace-level reuse may not reach the maximum instruction-level reuse.

The proof of the upper bound for trace-level repetition only applies to acyclic code formations. Sequences such as cyclic loop regions exhibit recurrence without necessarily having repetition at the instruction level. Individual instruction repetition is unnecessary when there exists redundancy over the resulting value and the steps to obtain the value can be verified to execute in an identical way. Theorem 3.3 states that reuse is possible for a given sequence of instructions without instruction-level reuse. In Reuse Theorem 3.3, the *Input* and *Output* terms respectively refer to the complete state, both register and memory, for the input and output elements of a trace of instructions.

Reuse Theorem 3.3 Let T be a cyclic trace of dynamic instructions $< i_1, i_2, ..., i_n >$ with two dynamic instances at t_a and t_b. Given a reusable instance of the trace T execution, then $Input_b(T)$ and $Output_b(T)$ are respectively equivalent to the previous execution record of trace T with $Input_a(T)$ and $Output_a(T)$. In such a case, for every $k \in [1, n]$ instruction $i_k(t_b)$ will execute identically as its respective dynamic instruction instance $i_k(t_a)$.

Consider the loop example in Figure 3.2 which computes the sum of the elements in the array A. To improve program execution speed, it is desirable to remove the loop's computation when the resulting sum is identically computed with a previous invocation. Assume that the loop is first invoked at a time τ and then at a later time $\tau + \delta$. Additionally assume that the loop is not located within a program domain in which the compiler could trivially detect the opportunity to avoid re-computation of the sum. As such, the reuse of the computation is based on determining the equivalence of the array A at time $\tau + \delta$ and time τ, for which there is a previously computed sum. The equivalence holds if array A remains unchanged along all executed program paths between τ and $\tau + \delta$. Once the equivalence is established one can simply use the execution result recorded at τ to eliminate the need to execute the entire loop at $\tau + \delta$. Furthermore, it is possible to exploit reuse on the region of instructions without individual reuse of previous operands occurring during the iterations of the loop.

A real-world example in which control and memory dependences limit the effective exploitation of dynamic redundancy of a cyclic region is shown in Figure 3.3. The example illustrates the function *ckbrkpts* from the SPECINT95 benchmark *124.m88ksim*. Figure 3.3(a) shows

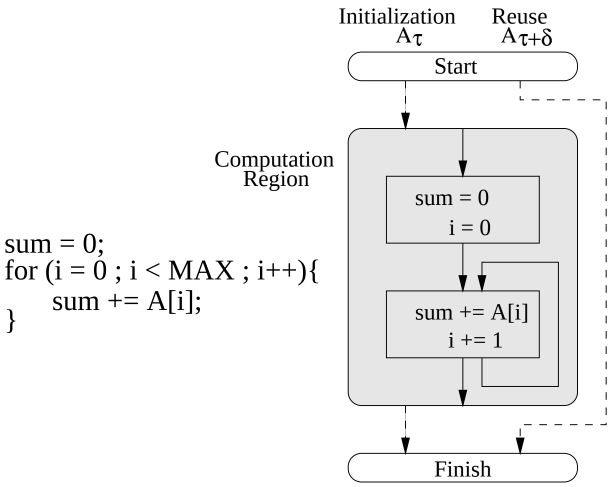


Figure 3.2 Loop example with potential reuse.

source code which scans the contents of the array *brktable* for breakpoint information that is updated from a set of only four functions: *nobr*, *br*, *settmpbrk*, and *rsttmpbrk*. During program execution, the code behaves as a reusable computation region since it is repeatedly executed without subsequent calls to any of the four functions that change the contents of the *brktable* array. As such, the results of one execution can be reused until the *brktable* array is changed.

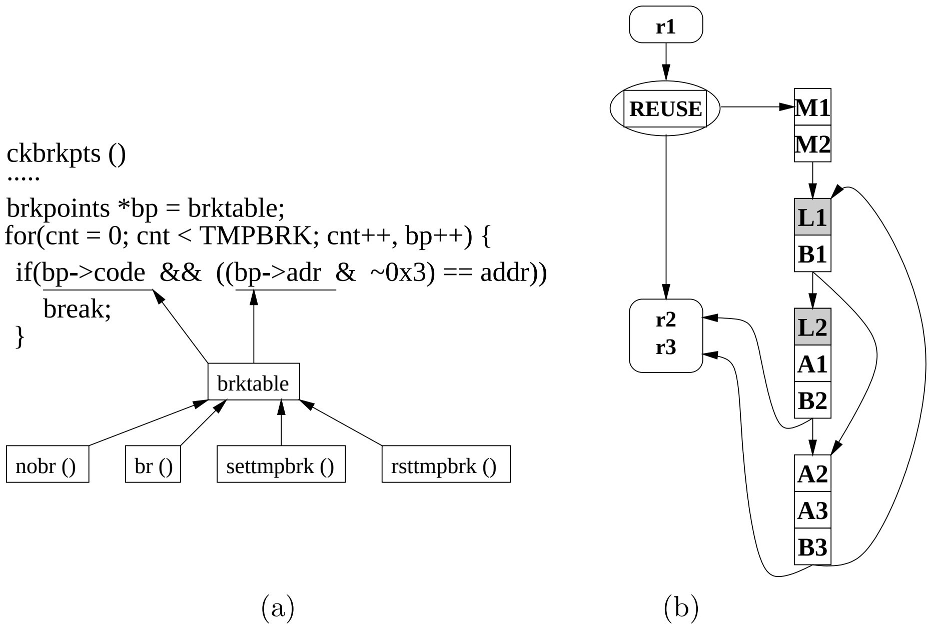


Figure 3.3 Region-level reuse example (a) source code and (b) reuse sequence.

3.1.2 Computation reuse potential

The reuse potential of proposed methods in this thesis is examined by collecting the amount of program execution that is redundant in the form of cyclic and acyclic sequences of instructions. The profiling techniques outlined in the RPS infrastructure record reuse opportunities for basic blocks and regions of code. Regions are defined as paths of basic block segments and include both cyclic and acyclic formations. Reuse for blocks and acyclic regions is detected by considering sequences of values consumed and produced by instructions as a program executes. Load instructions were considered reusable if their source memory location had not been accessed by any store operation between load executions. Reuse for cyclic regions is detected by monitoring additional program state at the invocation of the respective region headers.

Figures 3.4 and 3.5 illustrate the amount of program execution satisfying the evaluation guidelines. Each figure includes a *block* and *region* column. The block reuse shows an upper bound on the portion of a program that can be exploited with previously proposed techniques. On the other hand, the region-level exploitation subsumes the basic block definition and can exploit reuse along several control decisions. For these results, eight records of previous dynamic information for each code segment were maintained to check the potential reuse of program execution. Figure 3.4 represents the static percentage of each program exhibiting reuse behavior and Figure 3.5 represents the dynamic execution percentage of overall program execution exhibiting reuse behavior. These results indicate that region-level reuse mechanisms can potentially exploit almost twice the amount of program execution available to block-level approaches. The substantial difference between the block-level and region-level models is attributed to the ability of the region-level mechanism to reuse code executions that exhibit high-level recurrence not solely caused by instruction-level repetition.

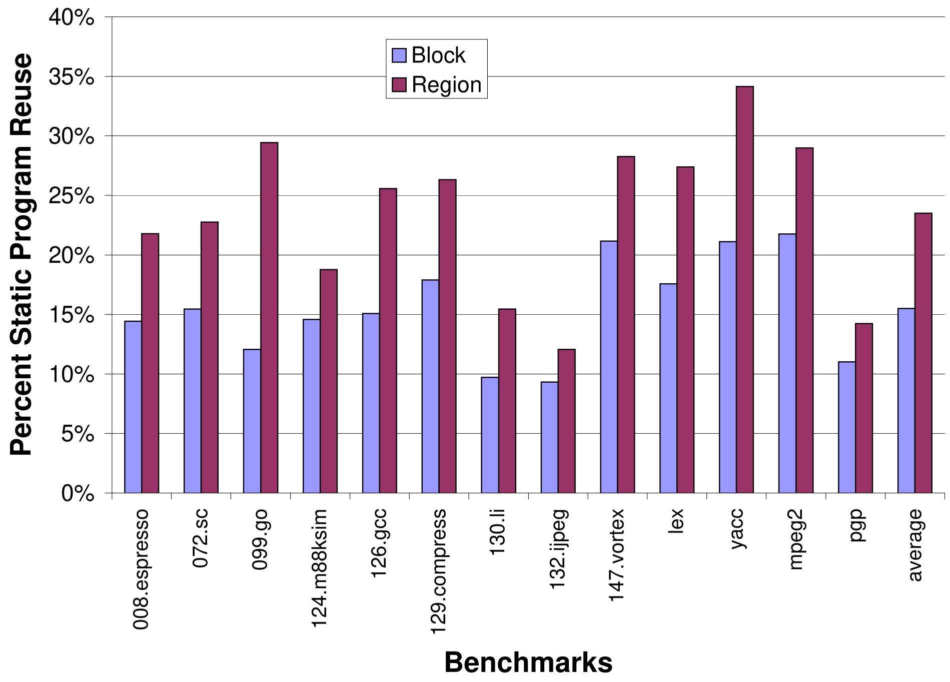


Figure 3.4 Static percentage of program with region reuse potential.

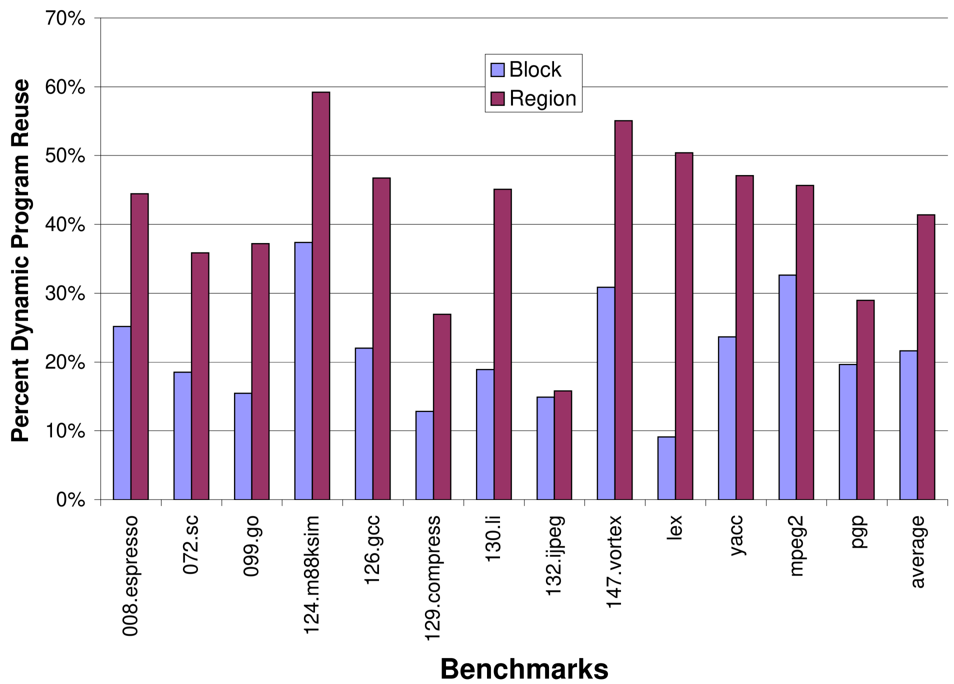


Figure 3.5 Dynamic percentage of program execution with region reuse potential.

3.2 Compiler Support for Reusable Computation Regions

Compiler support of the CCR approach involves several components: deterministic computation identification, reusable computation region construction, reuse formation, and reuse selection. Support for compiler-directed computation reuse is implemented in the IMPACT compiler framework. This section describes the four components related to the determination of reusable computation within general purpose imperative programming languages.

The most practical region of instructions that can be easily conveyed to hardware is defined by a single starting point and a single ending point. This definition allows all control path executions between the two points to be potentially exploited by the underlying hardware reuse mechanism. A compiler-directed approach could transform the code, as illustrated in Figure 3.3(b), by introducing a reuse instruction to inform the hardware that a sequence could potentially be reused. The reuse of memory computations can be significantly aided by the analysis techniques employed by modern optimizing compilers. Using interprocedural analysis, the complete points-to relation [33] for the *brktable* array can be constructed at compile time. As such, the compiler-directed approach can direct the program points that affect the array to invalidate previously recorded computations based on the contents of the array. This provides potential reduction in the cost of recording computations using memory since otherwise the consistency of all 16 entries of the *brktable* array must be maintained. As long as the equivalence is established, future invocations can use the execution results recorded at an earlier time.

3.2.1 Deterministic computation

The compilation techniques for dynamic computation reuse are based on the concept of *deterministic* computation regions. A deterministic computation region is an arbitrary, connected subgraph of the program control flow graph that can be analyzed to determine the location of all input operands that affect the region's computation. In the context of the CCR framework, two classes of deterministic regions exist: *stateless* (SL) and *memory dependent* (MD). Stateless regions are simply paths of code that define computation results which are based only on register operands and not on memory state. Memory dependent regions are paths of code that define computation based on both register operands and memory state, with the requirement that the memory dependence be either completely or conservatively determined at compile time. The compiler first performs program-level alias analysis to identify such load instructions and annotates them as *determinable*, indicating that all potential store instructions can be determined at compile time. The following section describes the interprocedural memory analysis support that enables the elimination of redundant memory accesses without monitoring the processor's memory access activity.

Static analysis derives regions consisting of simple loops, capturing potential redundancy for linked-list traversals and array scans. When statically annotating regions, a defined set of control blocks composes a region. Formally, a function can be represented by a control flow graph $G(N, E)$, where N is the set of basic blocks in the function and E is the set of all control flow edges between blocks in N. The graph can be divided further to represent the individual instructions that define basic blocks by the graph $GI(I, E)$, where I is the set of all instructions in the function and E is the set of all control edges. The difference between graphs G and GI is that GI uses implied edges between the sequential instructions of a basic block N. The graph GI is trivially created from G by designating every instruction as a basic block.

The first compilation phase of exposing computation reuse is based on the concept of deterministic computation regions. A deterministic computation region is an arbitrary, connected subgraph of the program control flow whose input register and memory locations can be identified. If the inputs do not change from invocation to invocation, and the input memory locations remain stable, the execution outcome of a deterministic region is completely reproducible. In the current implementation, a deterministic region cannot change the contents of memory. This restriction is mainly for the purpose of reducing the complexity of the hardware implementation. To further expose deterministic regions, global variable migration (register promotion), redundant store elimination, and store migration are applied.

3.2.2 Interprocedural memory disambiguation

Interprocedural pointer analysis is a critical component of employing the compiler-directed computation reuse scheme. The key issues concerning interprocedural analysis are flow sensitivity and context sensitivity. A flow-sensitive analysis system constructs pointers assignments with respect to the order of statement execution and determines the exact assignment to a memory location when various program paths include assignments that supersede earlier assignments. A flow-insensitive system provides less accurate information by representing all possible assignments to a memory location. A context-sensitive system distinguishes different caller contexts so that alias information for different functions do not conflict with each other.

The IMPACT compiler is supported with Cheng's work [34] which develops an efficient context-sensitive, flow-insensitive interprocedural modular program analysis to guide aggressive ILP optimization and scheduling. The system uses access paths [35] to represent memory location by how they are accessed from an initial variable. Globally and locally named data structures are analyzed to a precise representation in the points-to relation [33]. Coarse artificial names are also assigned for dynamically allocated objects; however, the complete points-to

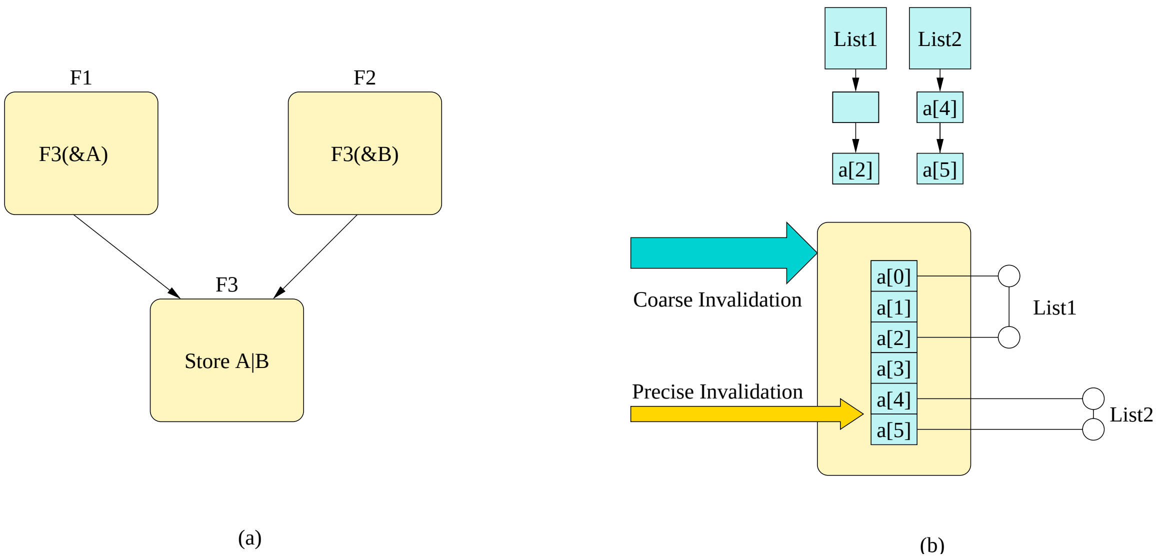


Figure 3.6 Use of compiler analysis in computation redundancy results in two potential false conflicts: context based (a) and dynamic-structure (b).

relation cannot always be derived. The impact on using compiler analysis to determine reuse of memory data results in conservative invalidation of memory reuse and is evaluted in Chapter 4.

Figure 3.6 illustrates the false invalidation conflicts caused by context-based overlap and course-grain dynamic aliases. Figure 3.6(a) indicates the case when a particular memory access in function *F3* can potentially write to two possible data locations. When reuse is being attempted on one of the data locations, it is necessary to invalidate the reuse when the write statement in function *F3* is executed. This invalidation is necessary even if the opposing data location is being written. Similarly, many dynamically allocated data structures can only be coarsely analyzed since their formation is based on run-time behavior. Figure 3.6(b) illustrates the case in which all potentially reusable computations accessing a data structure must be invalidated if elements cannot be precisely invalidated. The example shows that access to the items in one dynamic linked list of items causes the invalidation of the entire data structure of elements even when no conflict between other lists exists.

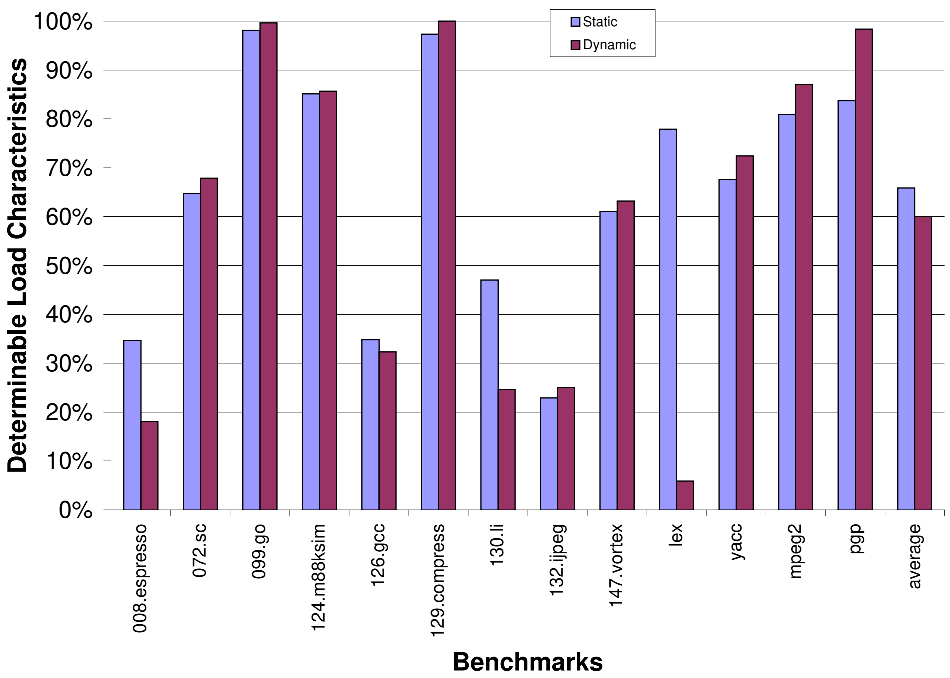


Figure 3.7 Distribution of static and dynamic memory references that are classified as *determinable* for load instructions.

An important aspect to determine is whether the interprocedural analysis system provides sufficient support for developing redundancy elimination techniques. Figure 3.7 illustrates the static and dynamic percentage of memory references that can be classified as *determinable* for load instructions. It appears evident that a wide range of applications can be accurately analyzed to recognize the relation between the dynamic memory references at compile time.

Figures 3.8 and 3.9 report the breakdown of static memory instructions and the respective number of named data locations that can be potentially accessed. By comparing the distributions of load and store instructions that can only access a single named data location, it can be concluded that using context-sensitive alias analysis is sufficient to precisely invalidate reuse in memory references. The results indicate that nearly 90% of the load and store instructions executing in the programs reference a single named data object.

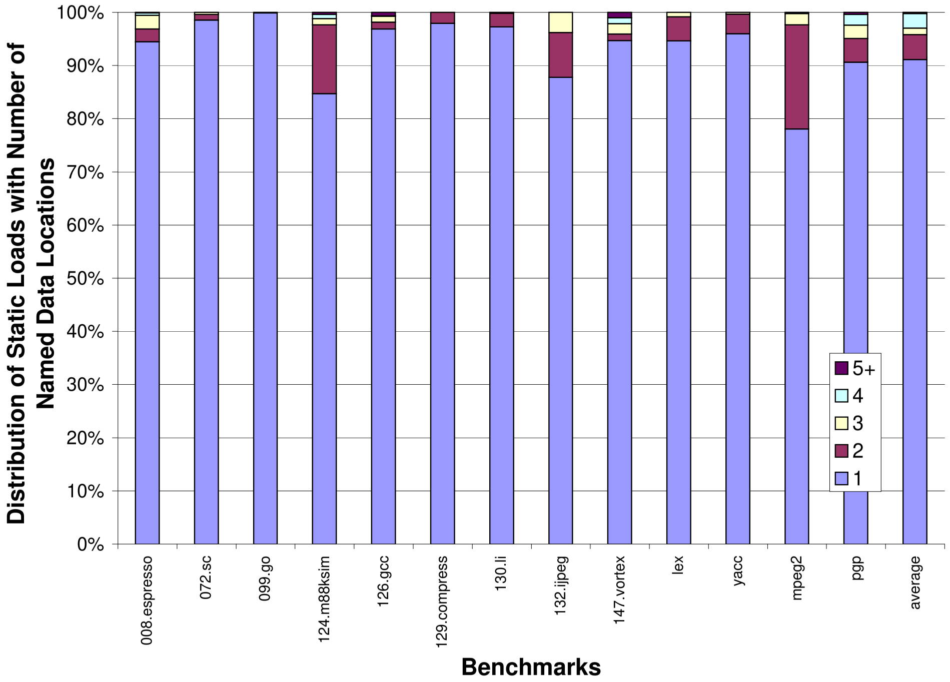


Figure 3.8 Static breakdown of named data locations for load instructions.

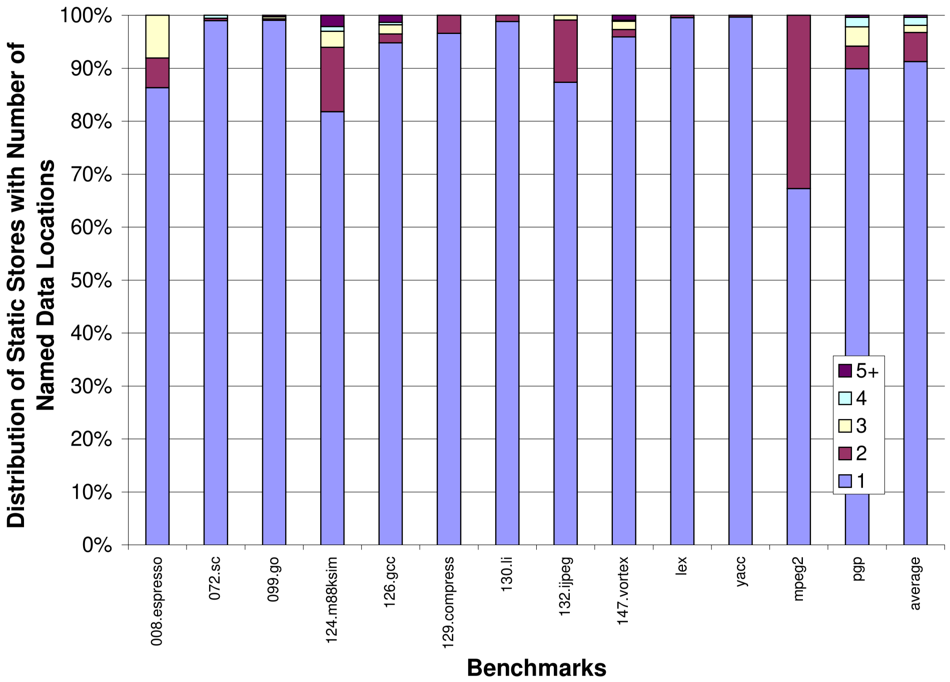


Figure 3.9 Static breakdown of named data locations for store instructions.

3.2.3 Data Connection Framework (DCF)

The overall framework for using the interprocedural analysis is highlighted in Figure 3.10. The figure illustrates the process of gathering the information of individual references in functions and constructing a database containing all access information (reads/writes) to referenced data in the Data Connection Framework (DCF). The database maintains the set of all instructions in a program that can alter a named storage location and is used during placement of invalidation instructions. Two stages of analysis, intraprocedural and interprocedural, are first performed at the high-level (*Pcode*) representation of code a program within the IMPACT compiler. The result of this analysis is then passed down to the *Lcode* representation in the form of *Sync Arcs* [36]. The term *Sync Arc* derives its name because it maintains the analysis information in the form of synchronizations arcs between *Lcode* memory operations. The scope of this thesis does not permit adequate explanation of *Sync Arcs*. One important characteristic of *Sync Arcs* is that *Lcode* modules accurately maintain their information through all compiler transformations.

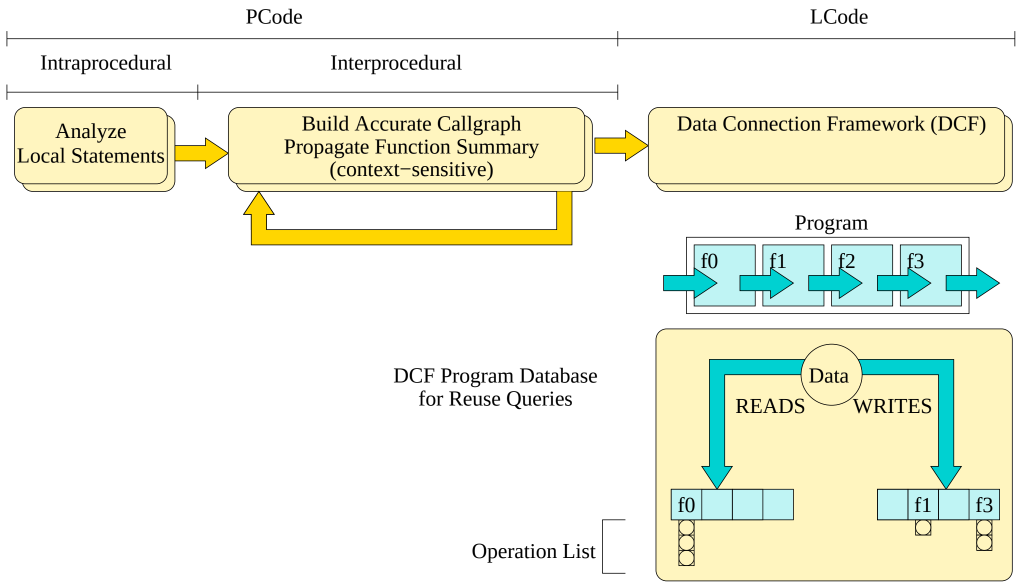


Figure 3.10 High-level interprocedural analysis and Data Connection Framework (DCF).

3.2.4 Reusable computation regions (RCRs)

Several constraints exist in developing effective tasks for the reuse mechanism. The essence of the reuse scheme is that the compiler directs the hardware to regions of code that have reuse potential and can be conveyed through the instruction set architecture modification previously mentioned. Therefore, to use the mechanism effectively, the regions selected are those that have the maximal number of reusable paths (acyclic or cyclic) between a starting and ending point. The reusable computation of such paths is a subgraph of a deterministic computation region, called a *reusable computation region* (RCR).

A reusable computation region delineates the section of program code that will be dynamically reused in the CCR framework and involves four region points:

Inception Point Starting point for memoization mode and location for reuse instruction.

Finish Point Ending point for memoization mode. Any computation within the path between the inception point and finish point can be reused.

Exit Point Any potential side exit from computation region and termination of memoization mode. No reuse along paths from inception to exit point.

Entry Point Side entrance to reusable region that is not involved with reuse or memoization of computation. Entry points do not necessarily exist for RCRs.

Figure 3.11(a) illustrates an abstraction of the reusable computation region concept. Reuse in the computation region occurs is any abstract execution path shown in Figure 3.11(b) while Figure 3.11(c) illustrates paths of execution are not viable for computation reuse in an region-based mechanism.

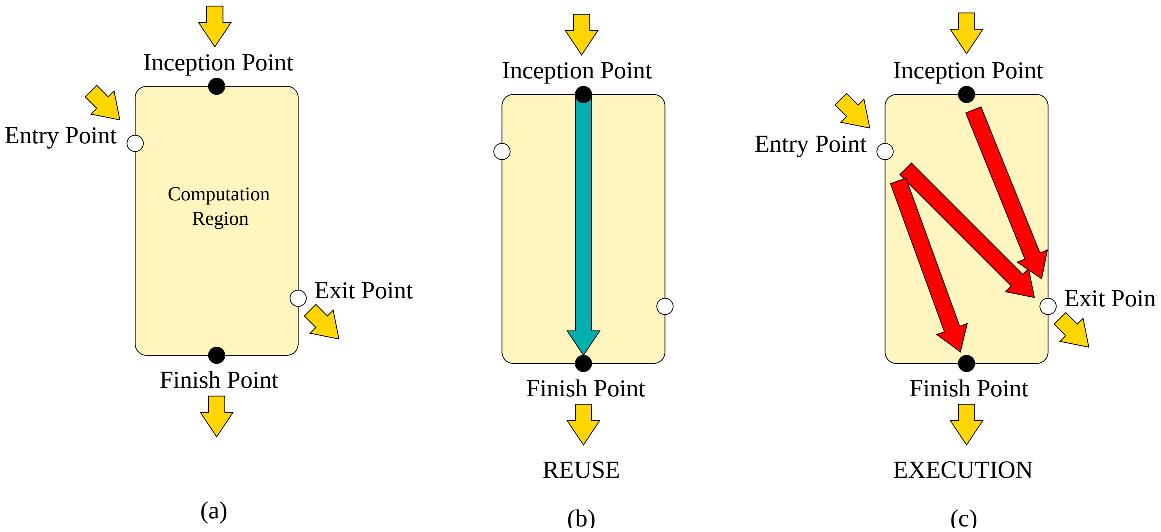


Figure 3.11 Abstract reusable region (a), reuse path in region (b), and execution paths in region (c).

Computation Region Characteristics.

There are a number of computation region characteristics that are useful in effectively exploring the compiler-directed concepts proposed in the next chapter. This chapter has already described how regions can be acyclic or cyclic and memory dependent or stateless. The most basic characteristic is called the region *degree* which is a measure of the number of inputs and outputs for the sequence of instructions. Intermediate register results which are not required after the region are not attributed to the degree of a region. Only dependent registers that are live-in or live-out for the region are necessary for completing successful reuse.

Two characteristics that describe the properties of the region are the *count* and the *height*. The *count* characteristic is the number of instructions connected between the input and output register operands. Similarly, since the instructions of a region do not necessarily belong to the same dependence tree, the characteristic *dimension* indicates how many independent dependence trees exist in designated region of code. The *height* characteristic describes the dependence height (level of dependent instructions) of all of the dependence relations found in the region tree. Similarly, since the regions can include multiple paths of execution, distinctions can be made about the height characteristic. The primary distinction is reporting the *path height*, which is the dependence height for stated path through the computation region. For regions without multiple paths between the inception and finish points of the computation region, the height of the region is constructed by finding the maximum dependence relation of the included instructions. The formation of computation regions is generally heuristic based since natural contiguous regions rarely exist and effective regions must be constructed by trying to eliminate the redundant execution for either a sequence with a high count of instruction or significant dependence height.

There also exist region characteristics that indicate properties about the region execution run-time behavior and functionality. Concerning input variation, an input value may range in behavior from being dominant to transient. A dominant input remains invariant for many executions, while a transient input changes frequently (although possibly remaining in a small number of variations). The behavior of an input to a region can influence the construction of the hardware mechanism that validates the computation results. Related to the method of describing the variance of a single region input, the *polyvariant* and *nonvariant* region types describe classes of region execution. Polyvariant regions may have many computation instances that are valid for a region at any time, whereas nonvariant region can only have one valid computation instance and respective result. Generally, polyvariant region exploitation is based on capturing recurrence, and nonvariant region exploitation is a redundancy elimination.

3.2.5 Computation region formation

With compiler and hardware support in an integrated reuse scheme, the compiler must identify a computation tree containing a sequence of instructions in which results can be recorded in hardware structure.

The compiler identifies the source operands to the tree of instructions so that each time it initiates the tree, the hardware determines if the input operands were used in the tree's previous history of execution. The primary advantages of region-level reuse are that (i) the execution a large number of instructions can be eliminated, (ii) the reuse information may be stored more concisely at region-level than at instruction-level, and (iii) group-level reuse may require fewer ports architectural register files to validate previous computation results. However, there are several critical issues for the design of effective compiler techniques to form the optimal groups of instructions to facilitate group-level reuse.

First, to increase the number and size of computation regions in real programs, several compiler optimization techniques need to be applied. These optimizations include global variable migration (registerization), redundant store elimination, store migration, and profile-guided function inlining, in addition to traditional and advanced optimization. Several programming language barriers to directing the elimination of dynamic computation redundancy are removed by performing these optimizations. When implementing region-level reuse it is important to reorder instructions to optimize the construction of the region. Region-level reuse dissipates to instruction-level reuse if only a single instruction can be gathered, which degrades the motivation of integrating the compiler and hardware in coordination approach.

The process of selecting reusable computation regions is heuristic-based and divides the computation into cyclic and acyclic region formation. The next sections describe profile-guided and static region formation methods that are developed using the following techniques. The process for computation regions consists of five primary steps: seed selection, successor formation, predecessor formation, and subordinate path formation, and reiteration of the previous formation steps. The first step is to select a starting instruction for creating a computation region, known as the reuse seed. The second step is to extend the region from the reuse seed by selecting a path of reusable successor instructions. Selection of each successor is based on three criteria: instruction reusability, region inputs, and region accordance. Determining the instruction reusability is a requirement for the region formation method.

The region input heuristic is used to determine if the instruction inputs overlap with the inputs of other instructions already selected. For example, if the instruction uses source registers that are also used by selected instructions, then the cost of adding the instruction is already subsumed. Similarly, when considering a successor instruction, reaching definition dataflow analysis is performed to detect an occurrence when a source register value is confined to a limited set of values with the currently selected region. This detection reduces the variability of the instruction's result, increasing its potential of being part of a successful reuse. Finally, the total number of live-in and live-out registers within a computation region is limited to eight. The region accordance heuristic is used to prevent the inclusion of memory instructions to the region that increase the potential of invalidating the computation region already selected. As such, the accordance heuristic limits the number of distinguishable named data locations to four in order to reduce the creation of ineffectual memory dependence regions. Early experimental evaluations of the accordance heuristic led to the definition of this setting. A summary of the algorithm applying all the heuristics for the selecting a successor instruction with the highest potential for improving the existing reusable region is shown in Figure 3.12.

```
Reuse_Successor(succ, Region)
potential_succ_list = Dependent_ops(Region);
FOREACH candidate IN potential_succ_list DO
// Include only instructions that can be migrated to region
    IF !Operation_migration(candidate, Region) THEN
        potential_succ_list = potential_succ_list - candidate;
        continue
// Include only instructions matching method's estimation of reuse
    IF !Reuse_Applicability(candidate, Reuse_method) THEN
        potential_succ_list = potential_succ_list - candidate
        continue
    candidate.reuse = Reuse(candidate, Reuse_method);
// Calculate the priority items for each candidate instruction
FOREACH candidate IN potential_succ_list DO
    Candidate_Region = Region ∪ candidate;
    candidate.candidate_region = Candidate_Region;
    IF Input_Overlap(candidate, Region) THEN
        candidate.overlap += 1;
    candidate.overlap += Overlap(candidate, potential_succ_list);
    candidate.height = Dependence_Height (Candidate_Region);
    candidate.count = Region.count + Instr_cost(candidate);
    candidate.order += Dependents(candidate);
    candidate.degree += (Input_Degree(Region) - Input_Degree(Candidate_Region));
    candidate.degree += (Output_Degree(Region) - Output_Degree(Candidate_Region));
    candidate.degree += (Memory_Degree(Region) - Memory_Degree(Candidate_Region));
potential_succ_list = Sort (potential_succ_list , SSP_function);
FOREACH candidate IN potential_succ_list DO
    IF Functional_Region(candidate.candidate_region) THEN
        return (potential_succ_list.top_entry);
return (Empty)
```

Figure 3.12 Function for selecting successor reuse instruction.

The heuristic function to coordinate the selection of the region successor as called in the pseudo code of Figure 3.12 is

$$SSP_i = (Reuse_i \times \frac{order_i + height_i + overhap_i}{1 + degree_i}) \tag{3.1}$$

The successor selection priority (SSP) is calculated for every instruction considered for grouping within the computation region. The reuse estimation is generated by the region formation methodology. The purpose of the selection priority function is to increase the dependence height and instruction order of the computation region while minimizing the input/output degree. By minimizing the degree, the opportunities to compact the results of the regions and reduce the number of required validating register ports are reduced. The region heuristics of Figure 3.12 and the individual formation reuse requirement find successor instructions that have good individual reuse, minimize the unnecessary invalidations, and minimize the number of input register dependences to the computation. The selection process attempts to find successor instructions through the dependence graph relations between instructions and migrates candidate instructions to the already selected region to create larger reuse sequences. This prevents the original program ordering from hiding potential reuse. The process of adding successors to the region continues until the successor path can no longer be extended using the successor heuristics.

The third step of RCR formation is to expand the computation path by adding predecessor instructions that flow to the original reuse seed. The fourth step is to add subordinate paths of reuse defined along the principle path. Such paths are selected by applying similar heuristics to those of the main path selection algorithm in the form of the following function:

$$PSP_{path} = \left(\frac{order\_region_{path}}{order\_region_{main}} \times \frac{degree\_region_{main}}{degree\_region_{path}} \times \frac{height\_region_{path}}{height\_region_{main}} \times \frac{reuse_{path}}{reuse_{main\_path}}\right) \tag{3.2}$$

The path selection priority (PSP) is calculated for each path being considered for inclusion in an existing region. Subordinate paths are included based on the highest priority until the candidate priority values being considered are less then 75%. In addition, if a path cannot be included with a region because the composite region exceeds the degree capacity designated for the region generation, then the path is discarded. The region formation algorithm makes further attempts to formulate such paths as individual regions by performing tails duplication for the composite region and re-initiating the region selection process. Similarly, if the subordinate path passes the selection priority and it does not end at the exit point, selective restructuring of the region and path tails is performed. After the inclusion of any subordinate paths, the computation region is generated. Side entrances, other than the inception instruction point to the selected set of paths, are annotated as entry points. Exit points are defined by all branch instructions that are directed to code outside of the selected set of paths. The final step of the reuse selection policy is to continually repeat the process of growing successors, predecessors, and subordinate paths until the region can no longer be expanded. By analyzing the newly formed region contents after each interval, the value-flow analysis heuristics are able to improve the reuse opportunities. The final transformations remove subsumed regions and partially duplicate beneficial regions when subordinate paths exhibit reuse potential but cannot be included with the current region.

A summary of the algorithm for the formation of reusable regions is shown in Figure 3.13. The algorithm consists of cyclic region formation, acyclic region formation, and region transformations. It operates over the function control flow graph (CFG) and processes the set of instructions (I) in each CFG node, applying the respective heuristic functions of either the profile-guided or static region formation method. At the end of iterative process, the top-most resulting instruction is established as the inception point for the reusable computation. The bottom-most instruction is defined as the finish point.

Figure 3.14 illustrates the acyclic formation process as applied to a region of code from the function *_cactive* of benchmark *008.espresso*. The four fundamental steps of region formation are noted in the figure. The seed selection which is reordered by reuse potential is identical

```
R = Empty;
// Construct cyclic regions
FOREACH region IN Cyclic_Reuse(CFG) DO
   R = R ∪ region;
// Order instruction reuse potential
Prioritize_Seed_Selection(I);
WHILE (si = Reuse_Seed_Selection(I)) DO
   R = si;
   succ = si;
   pred = si;
   change = true;
   // Iterate over successor, predecessor, and path formation
   WHILE (change) DO
      change = false;
      // Select successor (i ∉ Region)
      // using heuristics and reordering
      WHILE (succ = Reuse_Successor(succ, Region))
         Region = Region ∪ succ;
         change = true;
      // Record current region boundary
      Region_information = Boundary(Region);
      // Select predecessor (i ∉ Region)
      // using heuristics and reordering
      WHILE (pred = Reuse_Predecessor(pred))
         Region = Region ∪ pred;
         change = true;
      // Form subordinate paths weighted by reuse potential
      subordinate_paths = Reuse_Paths(Region)
      FOREACH path IN subordinate_paths DO
         IF (Reuse_Path(path)) THEN
            change = true;
            FOREACH instr IN path DO
               Region = Region ∪ instr;
      succ = Bottommost_Instruction(Region);
      pred = Topmost_Instruction(Region);
R = R ∪ Region;
// Remove subsumed regions
FOREACH region in R DO
   IF (Reuse_Region_Subsumed(region, R)) THEN
      Delete(region, R);
// Duplicate and coalesce adjoined regions
FOREACH region in R DO
   IF (Adjoined_Reuse_Region(region, R)) THEN
      Duplicate_and_Coalesce_Region(region, R);
```

Figure 3.13 Algorithm for selecting reusable computation regions.

to the example used to motivate trace-level computation reuse (Figure 3.1). The first instruction of the trace example has significant expected reuse value and a high number of dependent operations, making it the best seed candidate. Following the selection of successor and predecessor instructions (steps 2 and 3), subordinate paths are then selected. The region shown in Figure 3.14 has significant reuse and execution run-time characteristics and plays a major role in improving the overall application performance. The region represents a fundamental code segment of the application natural algorithm which cannot be optimzed using traditional techniques. The region formation techniques presented in this section can be used to isolate similar application specific computation regions from general programs. Although many different techniques can be used to form regions from a number of programming languages, the work of this thesis simply presents the initial motivation for developing computation-aware compilation environments.

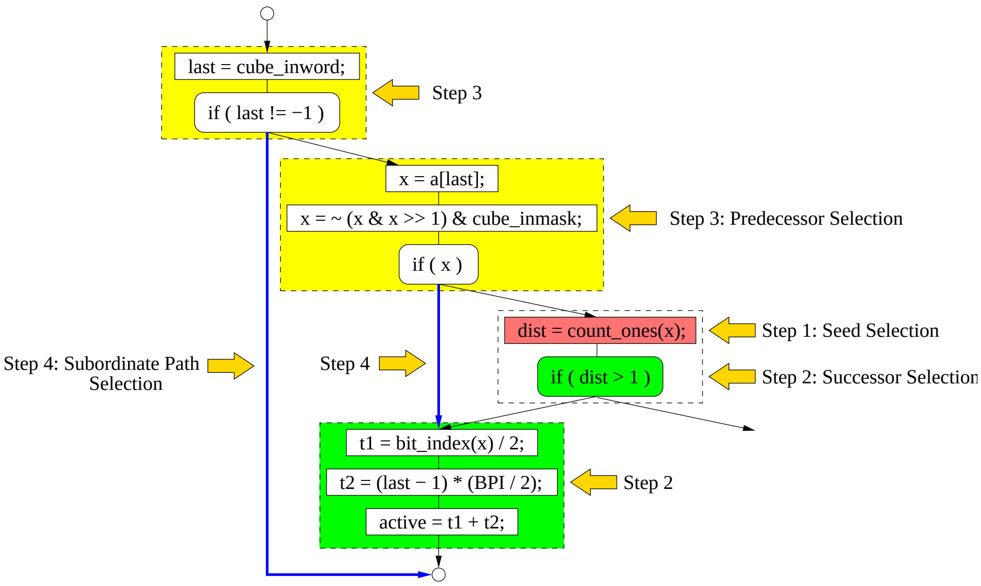


Figure 3.14 Region formation example for the function *_cactive* of benchmark *008.espresso.*

3.3 Profile-Guided Computation Region formation

The nature of detecting code sequences with redundancy execution behavior requires some estimation of the run-time behavior of a program. By profiling an application on a set of sample inputs, representative run-time information can be conveyed to the compiler. This enables an optimizing compiler to increase application performance by transforming its code to achieve better execution efficiency for those sections of the program with the highest execution frequency. Optimization of applications based on run-time value invariance [2], [24] offers great potential in exploiting run-time behavior. Other dynamic techniques have focused on discovering invariant relationships between variables from execution traces [37]. Invariant value profiling was also used to prove the effectiveness of compiler-directed computation reuse [38].

As the initial implementation of the proposed mechanism is completely directed at compile time, an accurate estimation of program reuse potential is essential. The profiling technique of RPS was developed as a result of this work and is designed to report accurate reuse information for three components: instruction-level repetition, reusability for memory operations, and cyclic computation recurrence. Instruction-level reuse information consists of the frequency of individual values and the recurrence of values within a set time interval. Memory reuse information consists of the frequency of updates to the referenced memory locations of each memory instruction. Cyclic computation recurrence is gathered by profiling the input registers at the start of cyclic region invocation, by recording the number of iterations for each loop invocation, and by relating the individual memory reuse information of every cyclic iteration to the reusability of cyclic invocation.

3.3.1 Reuse selection policy

Cyclic region formation. Cyclic reusable regions are identified by detecting inner-nested loops with deterministic computation. This excludes the loops that alter memory state with store and subroutine instructions. Similarly, load instructions within the loop must be classified as determinable. These same regions are identified earlier by the profiling system in such a way that reuse information is gathered for each invocation of the loop. The cyclic profiling information is used to check that a loop has a greater than 40% opportunity to reuse results and that greater than 60% of the loop invocations have multiple loop iterations. The threshold is satisfactory in balancing the cost of unsuccessful reuse attempts with the benefits of successful reuse. Formation of loop regions also use the path selection heuristics and include compiler analysis techniques to prevent loop variable increments from deterring the reusable region selection.

Acyclic region formation. The decision process for acyclic computation regions consists of five primary steps: seed selection, successor formation, predecessor formation, subordinate path formation, and reiteration of the previous formation steps. The first step is to select the seed instruction for creating a computation region from the set of all instructions within a function, ordered by the weight of the instruction execution, reuse potential, and the number of dependent instructions within the instruction's basic block.

The second step is to extend the region from the reuse seed by selecting a path of reusable successor instructions. While applying the region input and region accordance restriction, a successor instruction is then also considered based on its reusability. An instruction can be considered reusable if the weight of the top k recorded executions detected during profiling accounts for a large fraction of the instruction execution. By maintaining the ten most recent instruction executions, an instruction can also be designated as reusable if the history of the computation instances indicates a high tendency for reuse execution. Load instructions must satisfy two additional conditions: (1) the memory location referenced is reusable (defined by the frequency that stores access a location used by the load) and (2) the memory location is annotated as determinable. Control flow transitions between code blocks are considered likely if the weight of the control edge is 60% of the weight of instruction i, $Exec(i)$. Otherwise, the invariance of the conditional branch operands is used to select the successor path. Essentially, instruction-level profiling information is used to find the individual repeating instructions and to construct large regions of potential reuse in a bottom-up fashion. Therefore, an instruction i is reusable if it first satisfies the heuristic functions shown below:

$$Reuse\_Persistence(i) = \left(\frac{Invariance[k](i)}{Exec(i)} \geq R_p \right) \tag{3.3}$$

$$Reuse(i) = \left(\frac{Valid(i)}{Exec(i)} \geq R_i \right) \tag{3.4}$$

$$MemReuse(i) = \left(\frac{Valid(i)}{Exec(i)} \geq R_m \right) \tag{3.5}$$

Empirical evaluation found that setting R_i and R_m to .65 produces good reusable computation. Lower values tend to admit too many instructions in the region that are not successfully reused. The setting of R_p is varied depending on its use in compiler-based code reformulation or compiler-directed computation reuse. Setting R_p to .60 and the number of invariant values to five typically accounted for good persistent candidate region behaviors.

These profile-guided heuristics along with the other region restrictions find successor instructions that have good individual reuse, minimize unnecessary invalidations, and minimize the number of input register dependences to the computation. The selection process attempts to reorder instructions to create larger reuse sequences. This prevents the original program ordering from hiding potential reuse. The process of adding successors to the region continues until the successor path can no longer be extended using the successor heuristics.

The third step of RCR formation is to expand the computation path by adding predecessor instructions that flow to the original reuse seed. The conditions for adding a predecessor instruction are analogous to the conditions of successors. The successor and predecessor points define the principle reuse path in the control flow graph representation. The fourth step is to add subordinate paths of reuse defined along the principle path. Such paths are selected by applying similar heuristics to those of the main path selection algorithm with the additional path compatibility requirements outlined in the previous section.

3.3.2 Enhancing profile-guided region formation

In the base computation reuse approach proposed in the next chapter, it is important to select only statistically beneficial regions because the reuse mechanism could not be selectively used at run time. More importantly, the use of profile information by compiler-directed mechanisms can hide opportunities since profile-guided decisions may not be representative of all program workloads. Generally, it is imperative that all regions with any potential reuse behavior be annotated. Because many programs have different modes of operation for different inputs, collecting profiling information on a wide variety of inputs is an essential part of integrating compiler-directed computation reuse into processor design and software development.

Table 3.1 Region formation statistics based on profile-guided techniques.

Benchmark	Regions (Training)	Regions (Reference)	Overlap	Unique	Max
008.espresso	148	155	0.85	0.20	1.20
072.sc	70	71	0.95	0.06	1.07
099.go	440	484	0.80	0.30	1.30
124.m88ksim	128	128	0.99	0.01	1.01
129.compress	36	36	0.98	0.02	1.04
130.li	57	60	0.95	0.10	1.12
132.ijpeg	60	58	0.92	0.05	1.05
147.vortex	192	199	0.97	0.07	1.07
126.gcc	1764	1905	0.86	0.22	1.22
lex	51	53	0.99	0.05	1.06
yacc	69	73	0.96	0.10	1.10
mpeg2	83	80	0.95	0.01	1.02
pgp	51	54	0.82	0.24	1.24
average	-	-	0.92	0.11	1.12

To enhance the effectiveness of profile-guided region formation, multiple-input value-invariance profiles were collected. Region formation steps were applied to programs annotated with value-invariance information from the separate inputs. Table 3.1 shows the resulting region statistics. Columns 2 and 3 indicate the number of regions formed based on the training and reference inputs, respectively. Using the number of regions formed based solely upon the training input set as a base, three comparison fractions are calculated. The first comparison, *overlap*, indicates the regions identified using both selection methods. The average overlap fraction indicates that upwards of 92% of regions have invariant value behavior detected when region formation is guided by either input set. The second comparison, *unique*, designates the regions found only with the second input set. The final comparison, *max*, describes the maximum number of regions found when using both input sets. The average max result indicates that 12% of regions could be lost if only a single input were used to train the formation of computation regions.

The process of guiding reuse formation by using multiple input sets to find more region opportunities can be further extended by changing the reuse threshold. Code regions may have periods of execution where reuse does dominate execution, although such periods may not account for the majority of region behavior. Since the goal of enhancing profile-guided region generation is to select any possible computation with reuse behavior, the criteria for determining favorable R_i and R_m thresholds changes and new evaluations are required. A *region layering* technique is used to grow regions by incrementally lowering the reuse threshold. First, regions with a higher threshold are exposed and then the formation process gathers more regions by steadily relaxing the instruction inclusion threshold. Figure 3.15 shows the variation in percentage of total dynamic program execution captured in regions by region layering with four reuse threshold levels: 65%, 60%, 55%, and 50%. The results of Figure 3.15 indicate that nearly 15% more program execution can identified by lowering the reuse threshold.

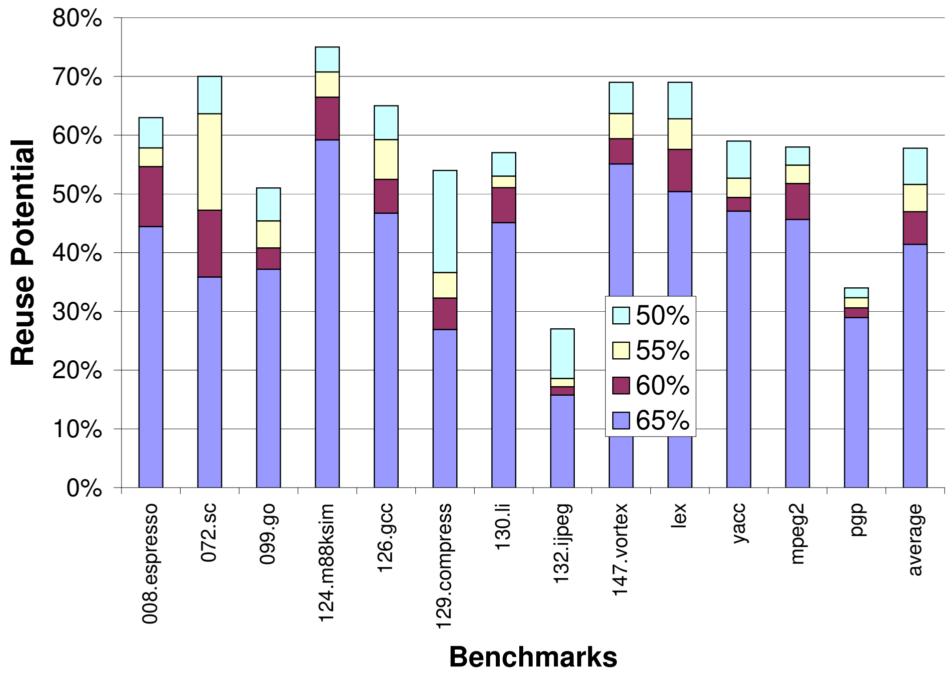


Figure 3.15 Dynamic reuse potential based on reuse layering thresholds.

3.4 Static Computation Region formation

Although the benefits of profile-directed optimization have been widely accepted, there are several fundamental drawbacks to profiling. First, profiling can be time consuming. Second, profiling may not be feasible in some environments, such as real-time or embedded applications. Third, profiling assumes that program behavior remains relatively constant for all possible inputs. If the program's behavior varies, poor performance after compilation may occur for some inputs. Finally, it is generally infeasible to generate all of the inputs to accurately model all program behavior.

An alternative to using value-invariant profile information is to use static program analysis to find traces of code with invariant behavior. A framework, called the Reuse Approximation System (RAS), was developed to estimate reuse behavior from other program information than reuse profiling. The RAS framework includes two methods, *inferred* and *structured*, for computation region identification. Both techniques are based on control profiling (branch execution frequency), a generally accepted technique used in most modern optimizing compilers. Traditionally, to expose sufficient instruction-level parallelism (ILP), basic blocks can be coalesced to form *superblocks* [39], an extension of trace scheduling [40], which reflect the most frequently executed paths through the code. Since superblocks have a single entry point and may have multiple side exits, they are similar to computation region code structures.

Frequently executed paths of code, found statically or dynamically, represent fundamental opportunities for reusing previous execution results. The main path of a superblock is generally a long sequence of frequently revisited instructions. Also, since branches are controlled by program data, the nature of the flow of control through an execution path relates directly to the value locality being exercised by the decision components of code. For these reasons, the proposed static region techniques are based on assigning superblocks as computation regions for the CCR approach.

Figure 3.16 illustrates the percentage of program execution for the training input attributed to superblock and candidate superblock traces of code. These results indicate that a significant percentage of program execution is attributed to candidate regions and can be exposed without the aid of value-invariant profile information. Both static region identification techniques involve finding candidate superblocks, determined by examining two features: instruction characteristics and region input/output requirements. First, superblocks are selected and partitioned based on the characteristics of the instructions in the path of basic blocks. Basic blocks with procedure calls and unresolvable memory accesses are not included within regions. In addition, a candidate region can only include load instructions which have been analyzed as *determinable*, indicating that all potential store instructions to a load can be determined at compile time.

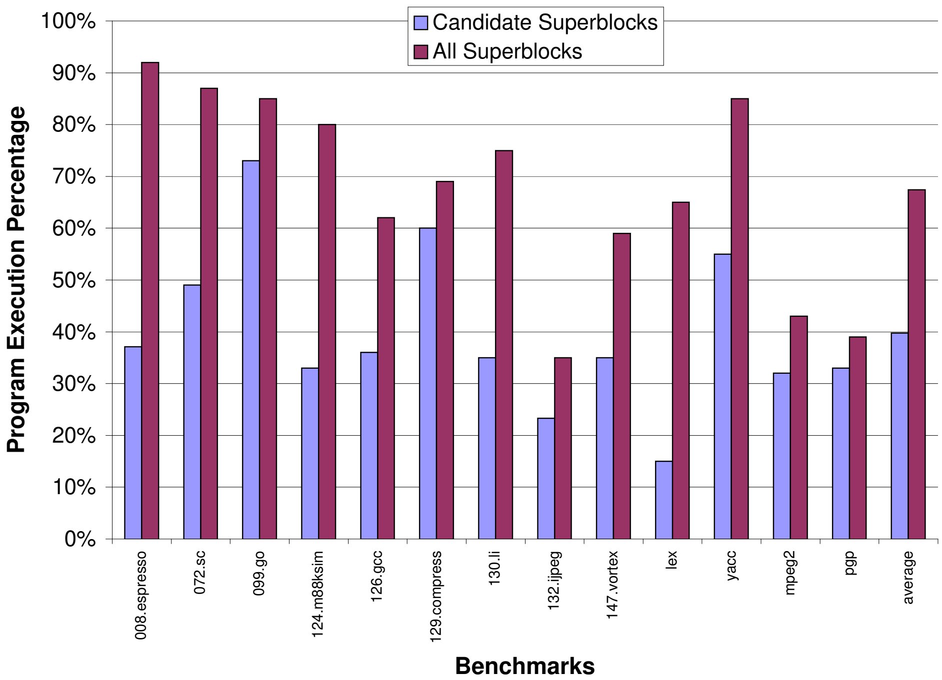


Figure 3.16 Distribution of dynamic execution in candidate superblocks and all superblocks.

The second feature of candidate superblocks relates to the design of the CRB entries and is used to achieve reuse of a computation region at run time. The base model of the CRB supports an input and output eight-entry register array, storing a computation mapping between eight input registers and eight output registers. Experiments revealed that 90% of superblocks matched these input and output requirements. Partitions of superblocks are selected when the capacity of the computation entries is insufficient to generate an execution mapping for a potential region.

3.4.1 Inferred computation regions

The *inferred* region identification approach uses interprocedural propagation of dataflow information to identify regions that possess some invariant behavior. The approach uses a coarse-grain dataflow analysis technique to infer value invariance for candidate regions. Inferring invariance in candidate superblocks is based on detecting *constructive* and *destructive* inferring instructions. Constructive inference indicates that a confined set or range of values may occur for an instruction or an instruction operand during some mode of execution of a program. The precision of an instruction's inference indicates whether the set of values is *closed* or *open*. A closed inference is made when a complete set of relations has been determined that governs all path executions, while an open inference has at least one unknown definition. The *order* of inferred precision indicates the number of inferred relations that have been determined for an instruction. An open precision for a variable means that it has an unknown value that it can assume at run-time.

Table 3.2 shows the inferences that the system makes for memory (loads) and the operands of computation instructions. Memory instruction inferences are made with the support of context-

Table 3.2 Instruction type inferences.

Instruction Type	Inference
Memory	Read only
	Write once
	Write infrequent
	Write value set
Computation	Value set - small number of values
	Value range - ($a \leq X \leq b$)

sensitive, flow-insensitive interprocedural alias analysis [41] and the DCF system. Read-only load instructions are best candidates for computation reuse since the accessed data is guaranteed to be identical to that of previous references. In addition, write-once data is a common inference data related to run-time invariant variables. Write-once data is determined by analyzing the callgraph and recursion relations of procedures. Many programs are designed with initialization phases in which run-time invariants are determined or parameters are set by reading parameter files. Write-infrequent data is detected when a load is defined at a significantly higher stage (level) in the program's loop-nested annotated callgraph than the respective referencing store instructions. Finally, using global analysis techniques, limited sets of variable values can often be determined, thus indicating the amount of run-time variation that can be expected for a variable. Although the initial system does not differentiate the execution frequency of any inferred relation, future work will include Binding Frequency Analysis (BFA) [42] to enhance the estimation of inferences. A related technique for automatically identifying invariant variables, Glacial Variable Analysis (GVA) [42], is extended to conservatively analyze recursive programs. GVA is an interprocedural staging analysis for estimating the variable modification rate by generating a measure of the difference between the execution and modification frequencies.

Inference of register operands is based upon interprocedural propagation of value relations throughout the program. Value relation propagation [43], [44], [45] consists of dataflow systems with solutions that identify expressions that are constant or constrained for every possible execution of a program. This is a simple forward dataflow problem: the value relation information of expressions is propagated interprocedurally on the program callgraph and intraprocedurally on each program function. Conversely, the inference system only attempts to determine which value relations are available to an instruction operand during execution. To do this, the inference system modifies the techniques of Wegman and Zadeck [43] and Callahan et al. [45] to propagate value relations for each expression until a fixed point is reached. Figure 3.17 illustrates the value relation lattice for inferring invariance. It is important to note that value-invariant profiling techniques essentially collect dynamic views of closed sets.

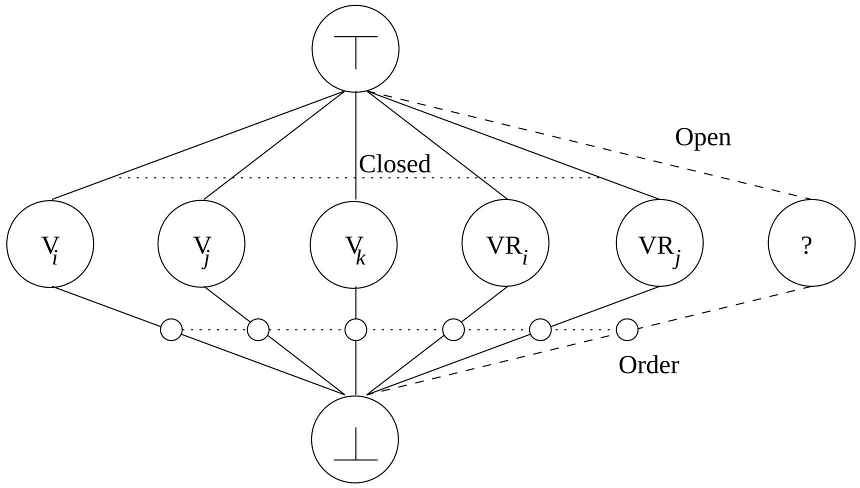


Figure 3.17 The value relation inference lattice.

Generally, the value relations for each variable assignment are described in terms of a lattice of all possible relations as shown in Figure 3.17. The highest level represents undetermined value relations, while the middle level contains a run-time deterministic relation. Loop-carried value relation expressions are detected and handled by conservatively resolving value relation assignments. The primary difference between value range propagation and this value inference method is in the join operator applied to the middle level of the lattice structure. In the standard constant propagation lattice, only one value relation exists, but for the value relation inference lattice, determination of any unique value relation causes the lattice to be extended. Essentially, the breadth of the lattice is used to infer run-time value invariance. Unknown value relations are created from a variable's definition set when no relation is propagated, generally resulting from noninferred load instructions. Eventually, a fixed point is reached and the propagation process terminates.

Destructive inferences are program indications that reduce the probability of observing a frequent, but small, number of input varieties. The primary example of destructive inference is sequenced operands, such as loop increment variables, which are used within their loop level. Increment variables used at a nesting level deeper than their definition are constructive inference since their invariance is based on the iteration space of the inner loops. Table 3.3 illustrates the result of inference analysis on candidate superblocks in the experimental set of benchmarks. The *inferred percentage* is the percentage of instructions with constructive inferences relative to the total number of instructions in the superblock. The amount of superblock execution corresponds the percentage of superblock execution designated in regions with the respective inferred percentage. The results of Table 3.3 indicate that assigning the inferred percentage to 30% enables the majority of candidate superblock execution to be captured.

3.4.2 Structured computation regions

Some regions of code represent fundamental algorithmic computation in which data is manipulated from input sources without any inferred value relations. Superblocks of this nature

Table 3.3 Percentage of superblock execution with various inferred percentages.

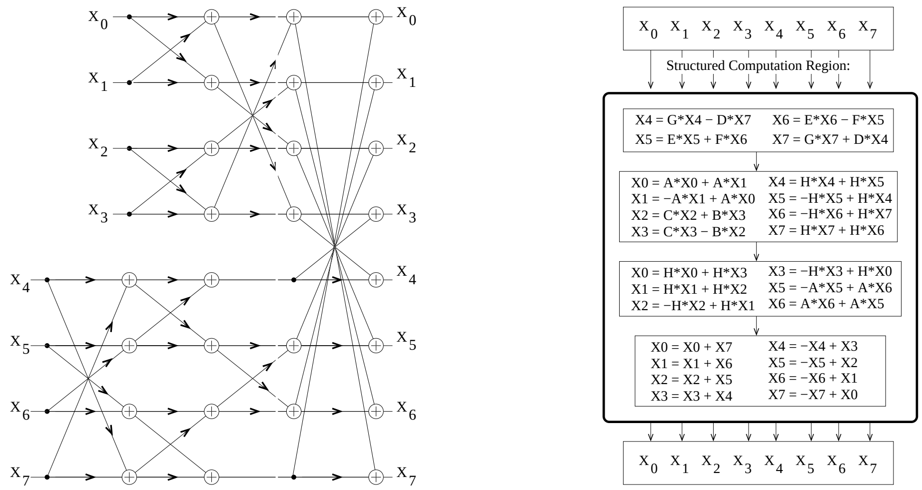
Benchmark	Inferred Percentage					
	0%	1-20%	21-40%	41-60%	61-80%	81-100%
008.espresso	42	4	7	25	11	11
072.sc	40	9	9	21	10	10
099.go	20	36	12	19	7	5
124.m88ksim	13	17	4	50	5	11
126.gcc	21	17	21	20	9	10
129.compress	32	18	18	16	8	8
130.li	30	14	12	19	10	16
132.ijpeg	33	26	4	20	8	8
147.vortex	35	6	14	19	13	13
lex	48	3	0	24	12	12
yacc	48	3	1	24	12	12
mpeg2	37	2	24	18	9	9
pgp	35	13	14	20	9	9
average	33.4%	12.9%	10.8%	22.7%	9.5%	10.3%

have no reuse inferences, yet may be excellent computation regions. To form regions structurally, two features of each superblock are examined: size and dependence height. Large size is the primary superblock selection constraint, since reusing the results of large superblocks could provide significant performance improvement. Likewise, a large reduction in latency may be achieved if superblocks with a considerable dependence height can be reused. In selecting superblocks, these features are referred to as the *structure size* and *structure height* parameters. Structure-based region formation aids the process of statically generating an abundant number of regions for the dynamically managed reuse system. In addition to superblocks, the structured technique also selects control flow cycles (inner loops) as candidate regions. Such loops often represent linked-list traversals and array scans that result in significant amounts of hard-to-detect redundant execution.

Although regions may only describe a fraction of reusable execution, it is important for the compiler infrastructure to identify all potential reusable computation regions. The structure-

$$dest[k] \;=\; \sum_{l=0}^{7} c[l] \cdot block[l] \cdot \cos(\tfrac{\pi}{8} \cdot (k + \tfrac{1}{2}) \cdot l)$$

(a)



(b) (c)

Figure 3.18 Structural identification of computation region, MPEG2 IDCT code (a), dependence graph (b), and computation (c).

based technique can be applied to regions with zero weight profiles, further reducing the reliance on profile-guided techniques. Figure 3.18 illustrates an example of a structured region formated from the row transform of an MPEG2 inverse discrete cosine transform (IDCT). Figure 3.18(a) depicts the dependence graph from the Chen IDCT algorithm [46], which requires 36 multiplications and 26 additions and has a dependence height of seven. Figure 3.18(b) shows the instructions of the dependence graph. These instructions constitute a superblock that is a candidate region and can be selected structurally. The selection affords the CCR a significant opportunity to eliminate redundant execution due to the inherent nature of the program code. Typically, DCT blocks of MPEG-compressed video sequences have only five to six nonzero coefficients, mainly located in the low spatial frequency positions [47]. This property has been used to streamline the design of custom low-power IDCT systems, which indicates that execution behavior of this region is quite predictable over many inputs. For each of the eight rows of an 8x8 coefficient block, the IDCT is called, and there is significant reuse locality (80%) for a history of 16 instances of the row computation.

3.5 Computation Region Summary

Tables 3.4 and 3.5 present some characteristics of regions identified by the various compiler methods. The region count and execution numbers compare the characteristics of inferred regions and structured regions to profile-guided regions (formed using a reuse threshold of 65%). Overlap occurs when more than 90% of the instructions of the static regions are found within a profile-guided region. The results of Table 3.4 show that while the inferred method (using an inferred threshold of 25%) identifies only 49% of the profile-guided regions, using a region height of four and region count of six identifies 68% in the structured approach. Other experiments indicate that together, both static approaches can identify an average of 75% of the profile-guided regions. In addition, the static methods appear to find a significant number of regions which were neglected when value invariance profile information guided region formation. The benchmark *126.gcc*, the largest program used, has the greatest fraction of regions found using static methods. A compression program, *129.compress*, is an interesting case because there does not appear to be a significant number of profile-guided regions that are not identified by the structured method, although the inferred approach is not able to detect many (less than 21%) of the regions.

Table 3.4 Region count fraction comparing static region formation techniques.

	Region Count Fraction					
	Inferred			Structured		
Benchmark	Overlap	Unique	Max	Overlap	Unique	Max
008.espresso	0.70	0.99	1.99	0.72	1.46	2.46
072.sc	0.66	1.05	2.05	0.96	0.17	1.17
099.go	0.53	2.92	3.92	0.88	0.78	1.78
124.m88ksim	0.79	0.35	1.35	0.86	0.26	1.26
126.gcc	0.85	3.24	3.34	0.54	12.5	12.6
129.compress	0.21	0.37	1.37	0.71	0.16	1.16
130.li	0.32	2.04	3.04	0.42	2.33	3.33
132.ijpeg	0.37	0.68	1.68	0.46	0.74	1.74
147.vortex	0.25	0.60	1.60	0.28	0.64	1.64
lex	0.53	2.75	3.75	0.67	2.45	3.45
yacc	0.48	2.03	3.03	0.52	2.24	3.24
mpeg2	0.28	1.37	2.37	0.96	0.08	1.08
pgp	0.45	3.06	4.06	0.95	0.29	1.29
average	0.49	1.65	2.58	0.68	1.85	2.78

Table 3.5 Execution fraction comparing static region formation techniques.

	Overall Region Execution Fraction					
	Inferred			Structured		
Benchmark	Overlap	Unique	Max	Overlap	Unique	Max
008.espresso	0.69	0.17	1.17	0.72	0.35	1.35
072.sc	0.62	0.27	1.27	0.88	0.47	1.47
099.go	0.76	0.23	1.23	0.89	0.26	1.26
124.m88ksim	0.92	0.44	1.44	0.93	0.62	1.62
126.gcc	0.76	0.23	1.23	0.42	0.34	1.34
129.compress	0.10	0.55	1.55	0.56	0.66	1.66
130.li	0.34	0.22	1.22	0.44	0.35	1.35
132.ijpeg	0.32	0.32	1.32	0.47	0.54	1.54
147.vortex	0.29	0.36	1.36	0.31	0.44	1.44
lex	0.54	0.70	1.70	0.55	0.82	1.82
yacc	0.34	0.14	1.14	0.32	0.23	1.23
mpeg2	0.19	0.30	1.30	0.84	0.50	1.50
pgp	0.55	0.12	1.12	0.82	0.40	1.40
average	0.49	0.31	1.31	0.55	0.46	1.46

CHAPTER 4

COMPILER-DIRECTED COMPUTATION REUSE APPROACH

4.1 Introduction

A computation reuse mechanism can provide performance improvement if the system can significantly reduce execution time for computing previous results and can adapt to run-time variations. Different trade-offs exist between hardware-based run-time methods and software approaches, and many design alternatives have been considered [17], [19], [22]. Our approach strives to achieve the best of both concepts in an integrated architecture and compilation framework. Toward this end, instruction set architecture extensions and micro-architectural mechanisms, referred to collectively as the Compiler-directed Computation Reuse (CCR) approach, are developed. The approach allows the compiler to identify code regions whose computation can be reused during dynamic execution. The instruction set architecture provides a simple interface for the compiler to communicate the scope of each reuse region and its live-out register information to the hardware. During run time, the micro-architectural components of the approach record the execution results of the regions for potential reuse. Explicit designation of computation reuse allows a large number of dynamic control, memory, and arithmetic instructions to be removed from the processor execution. Inclusion of control instructions such as conditional branches within the computation further increases the exploitation of value locality over block-level reuse methods [3], [20].

The remainder of this chapter is organized as follows. Section 4.2 provides a brief overview of works related to the concept of reuse and the intuitive rationale behind the proposed scheme. Next, Section 4.3 presents an overview of the architecture that facilitates the exploitation of reusable computation designated at compile time. Section 4.4 details the compilation issues and challenges associated with expressing and assigning computation region using the CCR approach. The effectiveness of the proposed approach in improving performance and exploiting instruction repetition is presented in Section 4.5. Finally, the chapter is summarized in Section 4.6.

4.2 Motivation

To exploit dynamic redundancy, two hardware-only strategies, speculative value prediction [16] and dynamic instruction reuse [17], have been proposed. Due to hardware complexity limitations, these techniques detect reuse opportunities at the instruction level rather than at a larger granularity. A more aggressive alternative is to allow the compiler to partition the program into potentially reusable regions of computation whose results are then dynamically recorded in hardware for future reuse.

The goal of any computation reuse scheme is to minimize the execution time and resource consumption of computing results that have been previously determined. The following examples illustrate that an integrated compiler and architecture reuse approach has the potential to eliminate large sequences of dynamic instructions at a much lower hardware cost than hardware-only schemes.

The example in Figure 3.1 demonstrates several fundamental barriers to effective exploitation of dynamic redundancy for both hardware-based and software-based methods. First, in exploiting the full redundancy of the sequence, a run-time hardware scheme must detect the dependences between instructions as displayed by the dependence graph of Figure 3.1(b). The dependence representation allows the mechanism to determine the set of instructions that are reusable from a particular set of starting instructions. Similarly, the hardware approach is limited in scope and may be unable to determine that only a single register is live-out of the computation. The alternative of storing the results of all registers defined in the region can be very costly. Third, the example could also benefit from code specialization, a software scheme that duplicates the code to efficiently handle certain run-time values. Although value profiling can be used to determine whether certain variables have the same value across multiple input sets, code specializations cannot easily adapt to variations in the value set.

A compiler-directed hardware approach has the advantages of both hardware and software methods. At compile time, the mapping relation between the single input register and single output register may be determined. With this information, the compiler can construct an alternative control flow graph, as illustrated in Figure 3.1(c), based on a reuse instruction that communicates with hardware buffers to detect reuse scenarios. If the hardware determines that a previous computation can be reused, the reuse instruction will update the corresponding output registers and proceed to the next sequential instruction. Otherwise, no recorded computation can be reused and the reuse instruction will branch to the original sequence, which executes and returns. In contrast to hardware schemes, the compiler-directed approach can accurately inform the hardware of the input and output registers which, in Figure 3.1, are $r3$ and $r26$. Compared to software code specialization techniques, the reuse instruction can be implemented to use multiple recorded instances, allowing a large number of instructions to be skipped for several input sets.

Similar to the block-level example, the region example of Figure 3.3 presents several fundamental barriers to exploiting reuse opportunities. First, since control instructions designate program direction changes, any potential region-level reuse detection mechanism must have the ability to understand the implication of start and completion of the computation intended for reuse. Essentially a run-time scheme must construct the implied control flow graph of Figure 3.3(b) and the dependence relations among its instructions. Only by constructing such information can attempts be made to reuse execution results along separate control paths and loop regions. The detection mechanism must also identify that certain instructions within loops, such as the increment of the loop index variable and the loop-back branch in Figure 3.3, do not exhibit repetition of all their operands. Such instructions are integral to the operation of the loop, yet their run-time behavior may inhibit the detection mechanism in determining that the entire loop is indeed reusable. Similarly, the memory instructions of the loop are an obstacle to effective reuse. For example, in order to reuse the results of previous loop invocations, the reuse approach must determine if array *brktable* in Figure 3.3 remains unchanged along all executed program paths between an initialization time and a reuse time.

Since the hardware cost to perform complex control analysis is high, most schemes are limited to only exploiting reuse along sequential sets of instructions. However, the ability to eliminate redundancy across basic blocks is fundamental to exploiting the full potential of computation reuse. Likewise, determining the equivalence of memory structures at different times requires substantial communication between the memory system and the reuse mechanism. A compiler-directed reuse approach has the advantage of being able to exploit reuse for large regions of instructions by (1) communicating the scope of each region to the hardware responsible for storing dynamic instances and (2) communicating the equivalence of memory structures at different times of the program execution.

The communication of a region boundary designates the section of code that can be dynamically reused. The most practical region of instructions that can be easily conveyed to hardware is defined by a single starting point and a single ending point. This definition allows all control path executions between the two points to be potentially exploited by the underlying hardware reuse mechanism. A compiler-directed approach could transform the code, as illustrated in Figure 3.3(b), by introducing a reuse instruction to inform the hardware that a sequence could potentially be reused. The reuse of memory computations can be significantly aided by the analysis techniques employed by modern optimizing compilers. Using interprocedural analysis, the complete points-to relation [33] for the *brktable* array can be constructed at compile time. As such, the compiler-directed approach can direct the program points that affect the array to invalidate previously recorded computations based on the contents of the array. This provides potential reduction in the cost of recording computations using memory since otherwise the consistency of all 16 entries of the *brktable* array must be maintained. Otherwise, as long as the equivalence is established, future invocations can simply use the execution results recorded at an earlier time.

4.3 Architecture Support

The Compiler-directed Computation Reuse (CCR) scheme introduces a set of hardware features and instructions to eliminate the need to dynamically detect reusable computation. In contrast to run-time instruction reuse schemes, the reusable computation is designated at compile time. The proposed architecture mechanism consists of the following components:

Reuse Architecture Hardware that records the dynamic computation information. The Reuse Architecture consists of a Computation Reuse Buffer (CRB) to store the reuse information.

Reuse Instruction Set Extensions Instruction extensions and execution semantics for conveying program information to the Reuse Architecture.

Computation Reuse Microarchitecture Hardware components that validate the recorded computations stored in the CRB and perform the update of architectural state for successful reuse of the computation instances.

4.3.1 Computation Reuse Buffer design

To achieve the reuse goals in the compiler-directed hardware approach, a caching structure is designed. Figure 4.1 depicts the basic model of the structure, called the Computation Reuse Buffer (CRB). The CRB is a set-associative structure indexed by an identifier number which is specified by the proposed ISA extensions in the CCR framework. The structure is similar in design to a cache that consists of an array of entries, referred to as *computation entries*. An entry supports the reuse for a particular compiler specified region by detecting the situation in which all of the input information to the region is recurrent. To do this, each active entry is responsible for recording computation information for future region executions. As such, each entry contains four fields: (1) the computation tag, (2) a *valid* bit indicating whether the entry currently contains a valid computation, (3) an array of computation instances, and (4) a least recently used (LRU) information array for managing the replacement of the computation instances. The computation tag field contains the computation identifier and is used for verifying the exact computation. A *computation instance* is defined as the set of input register operands and their

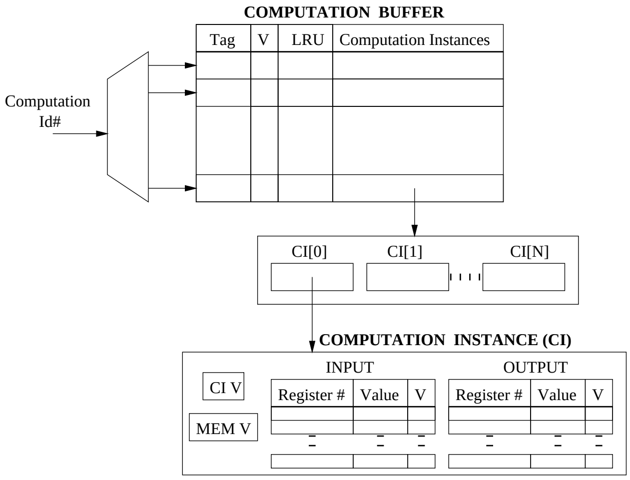


Figure 4.1 Computation Reuse Buffer (CRB) design.

respective values, the set of output register operands and their respective result values, and the validation of memory state used by the computation. A computation instance is reusable when its input register values match a previous execution of the computation and the input memory state has not been invalidated.

Multiple computation instances are used to record computations with different input values available for reuse. Each computation instance has two banks that contain an array of register entries, a *computation instance valid* bit indicating whether the instance defines a valid reuse and a *memory valid* field indicating whether the computation accesses memory and whether the memory dependent region is valid. Each register entry consists of three fields: the register index, the register value, and a *valid* field. The two banks respectively designate the necessary input and output information for the computation being reused, and the number of register entries is also specified by the particular implementation. For the input bank, the register index and register value fields record the necessary values that the respective registers must hold for the computation instance to be reusable. For the output bank, the information fields record the complete results of the computation that update the architectural register file during reuse. The valid fields indicate whether the register entry is active for the computation instance.

Several design enhancements can be made to the base CRB architecture. These enhancements focus on creating specific rather than uniform implementations of computation entries and instances. For instance, one enhancement might be to partition the design space of the computation entries to include variations on the type of computation instances that could be recorded. Such design modifications would reduce the complexity for reusable computations that consist of a small number of input and output operands. A second enhancement allows variations in the number of computation instances available for different computation entries. This enables reusable computation entries with large variations to be assigned to entries with sufficient computation instances; whereas computations with limited variations would be more reasonably assigned to entries with fewer computation instances. Any partition of the computation buffer entries is said to conist of *Computation Segments*, where a particular segment might include different properties. In Section 4.5 the characteristics of reusable computations designated are examined and different variations in the design of the base CRB configuration are investigated. However, the overall focus of this chapter is on evaluating the rationale of the approach, rather than investigating these specific implementation details.

4.3.2 Instruction set extensions

The CCR approach involves the introduction of new instruction extensions and two new instructions. The new extensions designate certain aspects of reusable computation to the reuse framework. The two new instructions are (1) *computation reuse*, which directs the hardware to determine if a computation has already been performed, and (2) *computation invalidate*, which directs the hardware to invalidate computations based on memory state changes. If the hardware does not find an opportunity to reuse previous computation results, the reuse instruction will branch to the computation code, which executes the sequence of instructions and updates the computation buffer. The reuse instruction provides a low overhead method of communicating with the hardware about the state of the machine, and is similar to the proposed mechanisms in data speculation [48] and software-controlled value prediction [49].

The introduction of the reuse opcode and instruction extensions allows the compiler to designate regions of computation that can be executed and then subsequently reused. The approach accomplishes this by a having a *region memoization* mode of execution that begins when the reuse instruction fails to find a valid computation instance. Upon starting the mode, the LRU computation instance is selected and construction of a new instance begins. Any register used before being defined while in this mode will record its information in the input bank of the instance. Additionally, instructions executed in the memoization mode have specific requirements for updating the records of the reuse instance and terminating the memoization mode. As such, the proposed ISA extensions enable the following execution semantics during the memoization mode:

Live-Out Register One new instruction extension is used to designate instructions generating live-out values. Destination registers defined for instructions marked with live-out extensions record the respective information in the output bank of the instance.

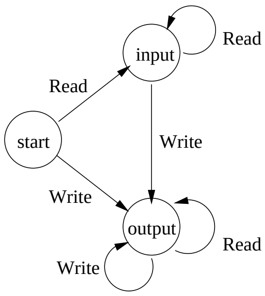
Load Instruction Load instructions executed during the memoization mode set the memory valid flag of the computation instance to indicate that a memory-dependent region is being prepared for reuse execution.

Control Instruction The compiler designates the end of the memoization mode by marking certain control instructions with new extensions to indicate computation region reuse endpoints and region exits. The recording of a computation instance occurs when a reuse endpoint instruction is used to leave the region.

Note that these extensions require the instruction format to contain more information and may increase the code size of programs. The execution results gathered during a successful memoization mode define a particular path in the region of instructions selected by the compiler. The compiler has the responsibility of insuring that the number of registers in the statically assigned reusable region can fit within the capacity of the computation instances. Similarly, another compiler responsibility is the program-level placement of invalidation instructions for regions accessing memory. The invalidations instruct the region computations that changes have potentially been made to the region's input data, and the computation instances may no longer be valid.

4.3.2.1 Memoization mode

The introduction of the reuse opcode and instruction extensions allows the compiler to designate regions of computation that can be executed and then subsequently reused. The approach accomplishes this by a having a *region memoization* mode of execution that begins when the reuse instruction fails to find a valid computation instance. Upon starting the mode, the LRU computation instance is selected and construction on a new instance begins. While executing in the memoization mode, all instructions that use or commit register results update the corresponding register information in the computation instance according the semantics illustrated in Figure 4.2. The semantics define the tasks necessary to identify and reuse the



State	Operation	Action	Transition
start	Read	record input	input
start	Write	record output if live-out	output
input	Read	-	-
input	Write	record output if live-out	output
output	Read	-	-
output	Write	record output if live-out	-

Figure 4.2 State transition diagram for recording register contents during memoization mode.

computation results in future executions. The new instruction extensions are used to designate instructions generating live-out values. Essentially, any register used before defined will record its information in the input bank of the instance. Otherwise registers that are defined by instructions marked with live-out extensions record the respective information in the output bank of the instance.

The execution of load and control instructions during the memoization mode also requires additional execution semantics in relation to the computation instance. The compiler designates the memoization mode to end by marking certain control instructions with extensions to indicate computation region endpoints and region exits. These annotations are used to distinguish between intraregion and interregion control traversals of a region. The recording of a computation is completed and the memoization mode ended when a control instruction with the new CCR extension returns to the location following the reuse instruction. The mode can also end when a control instruction with the new CCR extension exits (taken branch) the computation. Essentially, the execution results gathered during a successful memoization mode define a particular path in the region of instructions selected by the compiler's transformation of the region. The compiler has the responsibility of insuring that the number of registers in the statically assigned reusable region can fit within the capacity of the computation instances. Similarly, another compiler must place invalidation instructions for guarding the computation region executions that access memory. The invalidations instruct the region computations that changes have potentially been made to the region's input data, and the computation instances may no longer be valid.

4.3.3 CCR microarchitecture

The interaction of the microarchitecture pipeline and the computation reuse buffer is illustrated in Figure 4.3. For the initial study of the proposed approach, an in-order issue microarchitecture model is assumed, although the discussion contains material applicable to a generic dynamically scheduled superscalar processor. Four unique tasks define the reuse execution for a particular computation region: accessing the CRB computation entry, reading the architectural state for the computation instances, validating reusable computation results, and issuing the results. If a reusable result is found, the recorded results are committed. The traditional pipeline execution of the reuse instruction begins with instruction fetch and instruction decode.

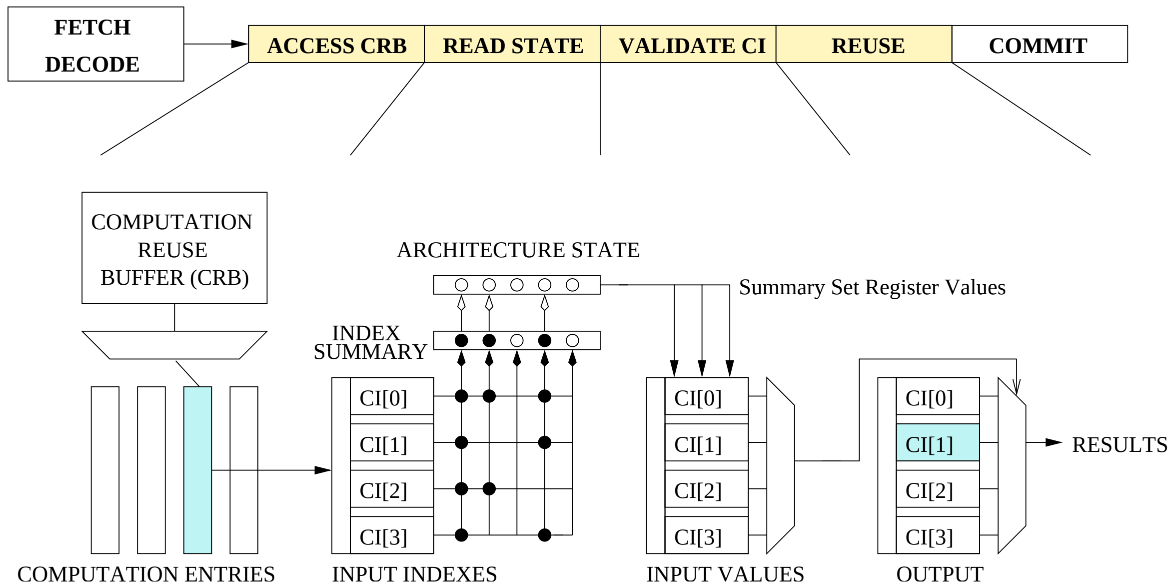


Figure 4.3 CCR microarchitecture pipeline.

During the CRB access step, the CCR architecture either detects a valid computation entry for the reusable region, or flushes partially executed instructions and directs the instruction fetch stage to begin executing new instructions at the location of the reusable code segment. For valid computation entries, it is necessary to read the architectural state corresponding to the input registers of the corresponding computation instances. To do this, a summary set of the active input registers for all of the computation instances is maintained during the creation of the instances. The summary set is the set of registers that require operand values from either the committed architectural state of the register file or uncommitted instruction results. In-flight instruction results and in-flight invalidate instructions force the reuse instruction to wait in order to validate the recorded computation instances. An interlocking mechanism exists between the computation reuse instruction, the processor retirement stage, and the register bypass circuitry to determine the necessary wait scenarios. Additionally, early feedback is available in the case when the hardware determines that the stalled register values can no longer affirm any of the computation instances. This is the case when the resolved register information either detects an independent computation instance or all computation instances are invalidated by the already confirmed register information. Since the performance of the reuse scheme is determined by the reuse latency and the percentage of reused instructions, these timing design considerations have been simulated within the evaluation environment.

Figure 4.4 illustrates two methods of constructing the summary set for computation instance. The index summary method of Figure 4.4(a) shows a basic method of reducing the number of register operands that the CRB entry queries from the architecture state when validating one of the computation instances. Generally, register queries are shared when there is overlap of register indices between the computation instances. Figure 4.4(a) shows a sample scenario in which eight total register accesses are required by the four individual computation instances. However, only three unique register indices are used over the four instances. By summarizing the register set prior to the access of the computation entry, an efficient query can be made of the necessary registers. Overall, experimental results indicate that with the summarized register index set, the necessary register queries only exceed the register porting fewer than 5% of all computation reuse attempts.

Although the basic summary technique reduces the number of register state queries, the method does not reduce the number of computation input register comparisons that must be made to validate a computation instance. Frequently there is significant overlap in register indices and values between neighboring computations instances. This overlap relates to the value locality behaviors that characterize the computation region execution histories represented by the computation instances. Figure 4.4(b) illustrates a summary model that constructs common input register comparisons for overlapping computation instances. The summarized comparisons reduce the number of register queries and the number of parallel comparisons necessary to validate a computation entry with a large number of computation instances.

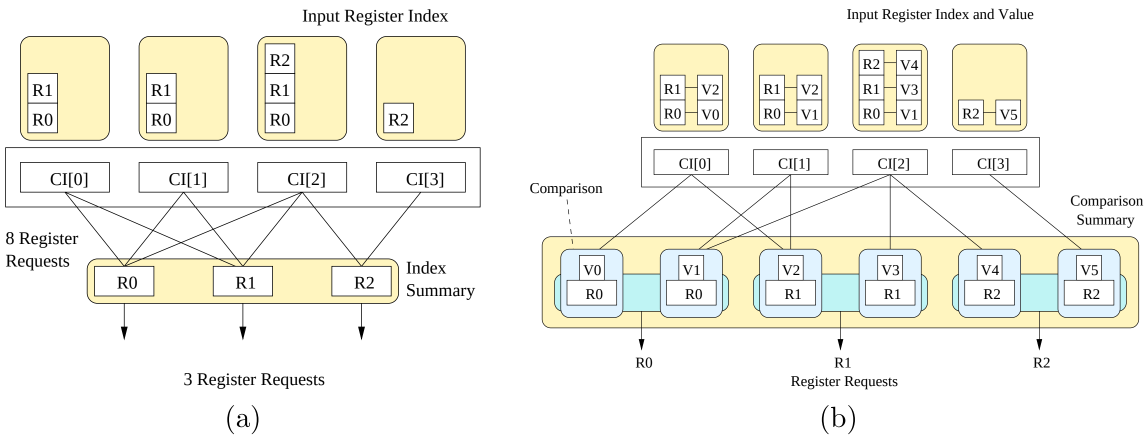


Figure 4.4 Methods of summarizing computation instances for verification: index summary (a), index and comparison summary (b).

In the example of Figure 4.4(b), the sharing of computation comparisons is present between $CI[0]$ and $CI[1]$ (for register $R1$ with value $V2$) and between $CI[1]$ and $CI[2]$ (for register $R0$ with value $V1$). Instead of performing eight comparisons, by exploiting overlap of possible computation instances, only six comparisons are necessary. The summarized model has an additional benefit; it is capable of providing a faster response in computation instance validation for some scenarios. By coordinating the multiple computation instances to share comparisons, the register indices affecting the largest set of instances can be trivially determined. Likewise, the register with the most value variation can be determined by noting the number of expected values being compared. The critical path of deciding the correct computation instance can be improved by initiating compares for the register with the highest variation or the largest number of computation instances. As such the CRB can begin polling the architectural state for those registers before the computation entry is completely resolved.

The step of summarizing the shared components of computation instances can be extended to a different design structure known as the *Shared-Instance CRB (SI-CRB)*. Figure 4.5 illustrates the CRB design that partitions all of the input and output register indices and values into summary sets. The summarized sets include a *register position* that indicates for which register index information is being maintained, and for the set of *value elements*, which respectively indicate the values being attributed to the register index. Two banks of summary sets, the *input value bank* and the *output value bank* hold the summary sets for the input and output relations for every computation instance in the computation entry. The value elements of the input bank of a summary set correspond to the live-values of the region and are used in comparison with the architecture state. Correspondingly, the elements of the output bank contain the respective result values for the register designated by the register position.

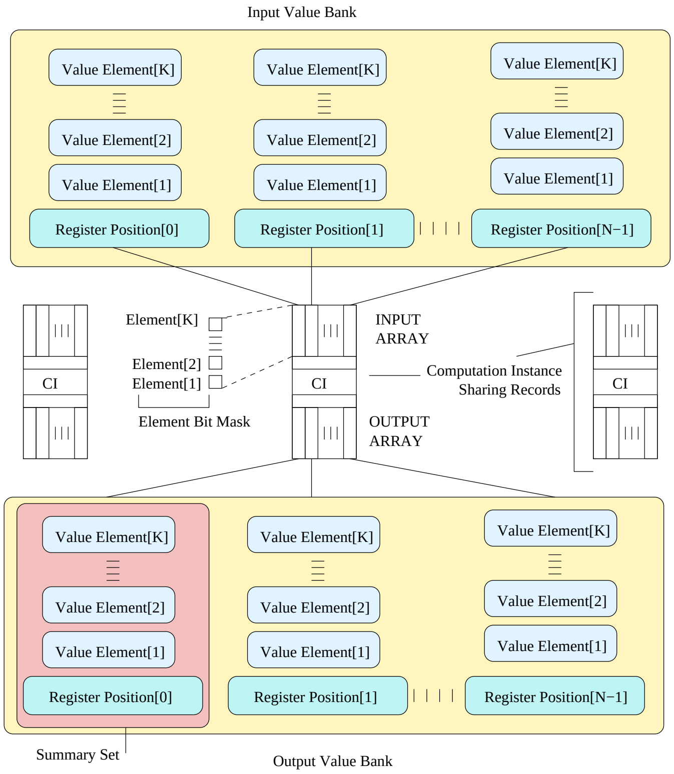


Figure 4.5 Shared-Instance CRB design.

A computation entry supports reuse of multiple computation instances by connecting the information necessary for each instance at a designated computation instance *sharing record*. The purpose of sharing records is analogous to the representation of the base CRB model's computation instances. Each record contains two arrays, the *input element vector array* and the *output element vector array*. The arrays hold N K-sized bit masks, where N corresponds to the number of register positions for the computation entry and K corresponds the number of value elements for a register position. The bit mask is set to indicate which value element is used for the summary set. To share the contents of summary sets between the computation records, each active record maintains the N bit masks so that the original computation instance can be represented.

There are many trade-offs in using either the base CRB or the shared-instance CRB. While the base CRB is generally uniform in its construction, it performs a significant number of parallel comparisons to validate a computation entry. Similarly the base CRB does not efficiently store redundant value information of the computation regions. The shared-instance CRB solves these limitations by exploiting the computation instance redundancies and eliminating redundant information representation and validation comparisons. The utilization of the the shared-instance is significant for regions with significant overlap in computation instances. However, the shared-instance CRB has a fundamental limitation. The flexibility provided by the uniform base CRB design was that each computation instance held. In the shared-instance design, each active computation record contends for representing its computation instance in the register positions and value elements shared by every computation instance of the computation region.

In the shared-instance CRB the primary design parameters are to decide on the number of register positions to create and the number of values to maintain for each register. If a region was simple and always depended on a set number of registers, the design decision would fix the register positions. However, this would limit the number of inputs and outputs of the region generated by the compiler. The shared-instance design solves this problem by recording only the registers that need to be compared to validate all of the computation instances. For contention of register positions, the register position with the fewest dependent records is selected. This is easily determined from valid fields maintained in each summarized sets. Contention for register positions cannot exist if the shared-instance CRB design is constructed with $R * X$ register positions, where R is the max number of records that can be represented and X is the restricted number of input/output registers of a region.

The variation of a particular register value is important since the variation can cause more than one computation instance record to be inactive. Contention for value elements is solved with the solution of determining the element entry with the fewest dependent records. Value elements with the fewest dependent records are replaced. Replacement of value elements can remove up to $R-E-1$ records, where R is the max number of records that can be represented and E is the max number of value elements in a summary set. Therefore, if there is an equal number of value elements and computation records, replacement of a value element can at most remove one computation record. Experimental results concerning the trade-offs between shared-instance and base CRB are examined in Section 4.5.

Although the SI-CRB design has major advantages in maintaining the computation instances and efficiently validating results, there are also techniques for improving performance in the base CCR design space. Figure 4.6 illustrates design techniques to make the process of verifying computation results more efficient. Figure 4.6(a) shows the base case of having the computation instance information of a computation entry stream down to the architecture state. In turn the architecture state is used to validate the computation instance results. Figure 4.6(c) divides the computation entry into *computation frames* of a history-based buffer. The history buffer validates the computation instances with the most recent execution. The advantage is that the number of parallel comparisons is reduced and many computation instances have temporal locality of their results, indicating that checking the recently executed instances most quickly will provide benefit. The number of frames determines how many additional cycles the validation process will take for computation instances not accessed in the recent history.

A final performance enhancing technique, called Computation Anticipation (CA), uses the design of Figure 4.6(c). The Computation Anticipation mechanism reduces the dispatch latency

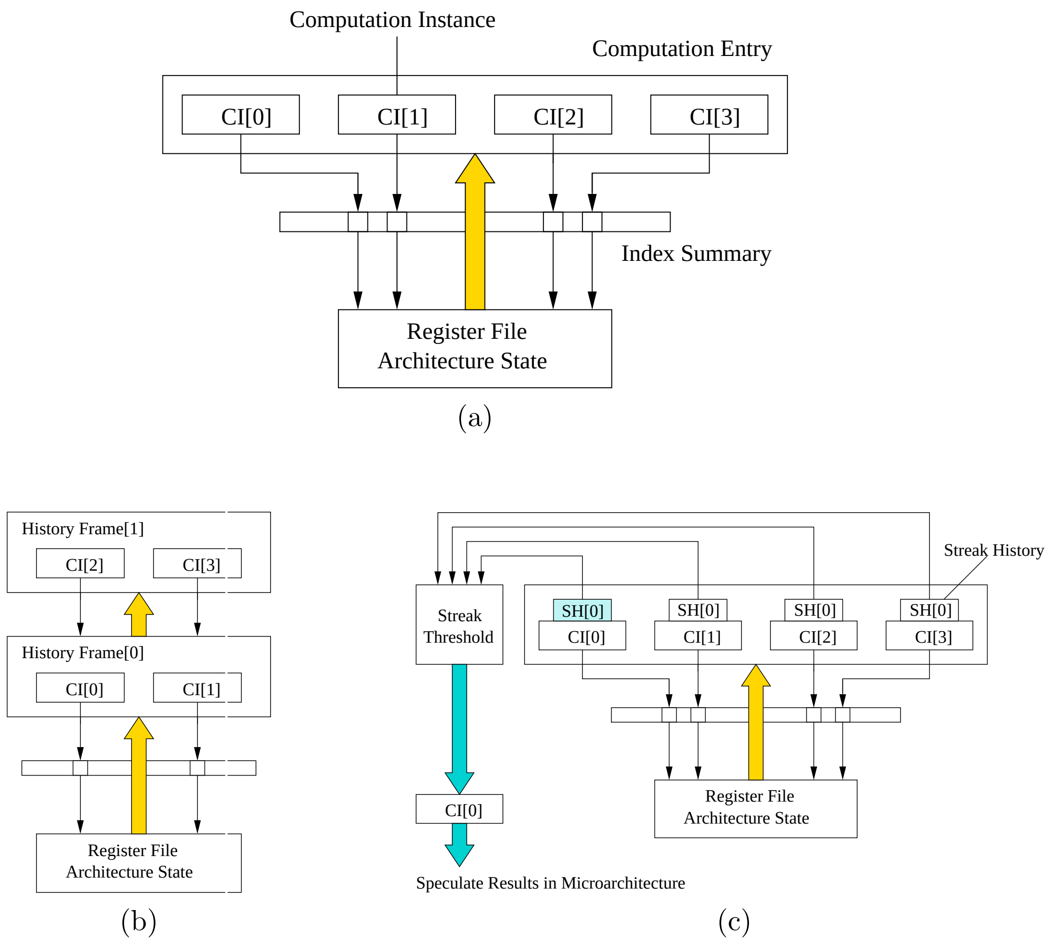


Figure 4.6 Verification of computation instance, base model (a), history-based model (b), and computation anticipation (c).

of the approach by using the computation history to speculatively dispatch cached computation results. A streak history is maintained for each computation instance that indicates the number of times in the recent region execution that the particular instance was selected. By assigning a confidence threshold to the number of repeated executions of the same computation instance, an opportunity exists to speculatively anticipate that the results held in the computation instance will be used. The microarchitecture of the processors must include support to recovery in the case of mispredictions. The primary use of this technique is to extend the concept of value prediction into use with the compiler-directed computation reuse approach. These enhancements focus on investigating general rather than specific CRB implementations.

After the process of validating the computation instances has been performed by any of the techniques discussed, there are two cases that result: the nonexistence of reusable computation and the presence of reusable computation results. For successful reuse determination, the live-out registers are updated by issuing multiple results to the retirement buffer. Otherwise the processor flow of control is directed to the computation code after the pipeline is cleared of partially executed instructions.

Finally, a final design mechanism explored in this thesis is the extension of the dual path execution concept [50], [51]. Dual path execution has been investigated to reduce the branch mispredictions penalty by selectively executing instruction from both paths following a conditional branch. In the CCR paradigm, a failed reuse attempt behaves in an identical manner as a mispredicted branch. As such, there are opportunities to extend dual path execution to the CCR approach. Since studies have indicated that dual path execution can potentially reduce branch misprediction penalties by 30%, there is potential for exploring the concept with the CCR approach. In Section 4.5 several of the architecture optimization methods proposed in this thesis focus on improving performance in the same area as the dual path execution methods.

First, computation anticipation works by speculating path execution prior to target resolution. The concept is related because instructions are placed in the pipeline to increase utilization. Also, since the dynamic management system explored in Chapter 5 applies a confidence mechanism to a path decision problem, there is significant overlap in the proposed dual-path techniques which operate by confidently asserting path predictability. Finally, the compiler transformation techniques of Chapter 6 evaluate how sequences of reformulated code and original program code can be concurrently executed to improve performance.

4.4 CCR Compiler Support

In the CCR approach the compiler identifies the code regions whose computation can be reused using the CRB hardware resources. Due to limitations in the available silicon resources, the CRB design employed for a processor may have restrictions in the number of computation entries and the number of computation instances per entry. At run time, the CCR design is incapable of designating the assignment of computation buffer resources to the most profitable computation regions. However, in practice there are compiler methods of prioritizing the computation regions to be designated for a processor with a CRB with specific capacities.

After the region formation process has gathered all potential candidate regions, the region prioritizing algorithm of Figure 4.7 can be applied to limit the designation of computation regions in the program and increase the likelihood of successful run-time reuse using the CRB resources. Two estimates are performed using the instruction-level reuse information, region reuse percentage and region variance. Region reuse is estimated by averaging the individual reuse behavior of all instructions. Region variance is estimated by computing a weighted average of the variance of all the region instructions. These estimates are made and used in the region selection heuristic function shown below:

$$RSP_{region} = Exec_{region} \times Reuse_{region} \times [Height\_Factor] \times \frac{Count_{region}}{k_{count}} \times \frac{CRB_{instances}}{Variance_{region}} \quad (4.1)$$

```
Select_Pprioritized_Regions (Region_Candidates)
FOREACH region IN Region_Candidates(Program) DO
// Calculate estimated reuse behavior
  region.reuse = Reuse_Estimation(region);
// Calculate estimated variance behavior of instances
  region.variance = Variance_Estimation(region);
// Apply region selection priority function
  region.priority= Region_Selection_Priority(region);
Region_Candidates = Sort (Region_Candidates , RSP);
region_count = 0;
Selected_Regions = NULL;
WHILE (region_count <= C_entries * CRB_entries) DO
    selected = Top_Region(Region_Candidates)
    Selected_Regions = Selected_Regions ∪ selected;
    region_count = region_count + 1;
ENDWHILE
return (Selected_Regions)
```

```
Reuse_Estimation(region)
reuse = 0;
count = 0;
// Calculate an average of the estimated instruction reuse behavior
FOREACH instruction IN region DO
   reuse = reuse + instruction.reuse;
   count = count + 1;
return (reuse / count);
```

```
Variance_Estimation(region)
region_variance = 0;
count = 0;
weight = 0;
// Calculate an average of the estimated instruction reuse behavior
FOREACH instruction IN region DO
// Calculate the weighted average of the number of computation
// instances used during profiling
   instr_variance = 0;
   FOR i = 1 TO computation_profile_capacity DO
      instr_variance = instr_variance + i * instruction.variance[i];
      weight = weight + instruction.variance[i];
   count = count + 1;
   region_variance = region_variance + instr_variance;
return (region_variance / weight);
```

Figure 4.7 Algorithm for selecting prioritized reusable computation regions.

The Region Selection Priority (RSP) is calculated for each region being considered for activation. The execution weight and estimated reuse are main terms used to prioritize the region selection. In addition, both the region height and instruction count are used to encourage regions with the greatest effect on improving execution performance. The variables k_{height} and k_{count} are constant factors used to adjust the priority calculation. The estimated variance of a region is weighed against the number of computation instances for each computation entry of the CRB. The variance term deters candidate regions that require a higher degree of computation instance support than the CRB design provides. To account for any limitation in the number of computation entries, prioritized regions are selected until a constant factor of the number of entries is designated. Experimental studies determined that the performance of the compiler allocated regions could be effectively managed by assigning only two times the number of regions compared to the number of entries in the target CRB.

4.5 Experimental Evaluation

4.5.1 Methodology

The IMPACT compiler and emulation-driven simulator were enhanced to support a model of the proposed architecture framework and the region formation techniques respectively introduced in Section 4.3 and Chapter 3. The benchmarks used in all experiments consist of SPECINT92, SPECINT95, UNIX, and MediaBench programs. The base level of code consists of the the best code generated by the IMPACT compiler, employing function inlining, superblock formation, and loop unrolling.

The base processor modeled can issue in-order six operations up to the limit of the available functional units: four integer ALUs, two memory ports, two floating point ALUs, and one branch unit. The instruction latencies used match the HP PA-7100 microprocessor (integer operations have one-cycle latency, and load operations have two-cycle latency.) The execution time for each benchmark was obtained using detailed cycle-level simulation. The parameters for the processor include separate 32K direct-mapped instruction and data caches with 32-byte cache lines, and a miss penalty of 12 cycles; 4K entry BTB with 2-bit saturating counters, and a branch misprediction penalty of 8 cycles. Pipeline stalling for in-flight dependent operations is also integrated to delay the reuse resolution time. Failure to correctly reuse computations causes the processor to experience a delay similar to the branch misprediction penalty. The simulations evaluate computation buffers with 32, 64, or 128 direct-mapped entries with 4, 8, or 16 computation instances (CIs) per entry. Each CI supports an input and output 8-entry register array.

4.5.2 Experimental results

Three categories of results are presented. The overall performance of the CCR approach is first examined. Second, some of the characteristics of the reusable computations are presented. Third, some general results relevant to the compiler-directed scheme are evaluated.

Performance. The overall cycle-time speedups for the compiler-directed approach are presented in this section. Two variations in the CRB design are evaluated: variation in the number of computation instances per computation entry and variation in the number of computation entries. Performance is reported as speedup, which is derived by dividing the number of execution cycles for the base architecture by that of the architecture with the CCR framework.

The first CRB variation considered is the number of computation instances per computation entry. Figure 4.8 presents the effect on performance for varying the number computation

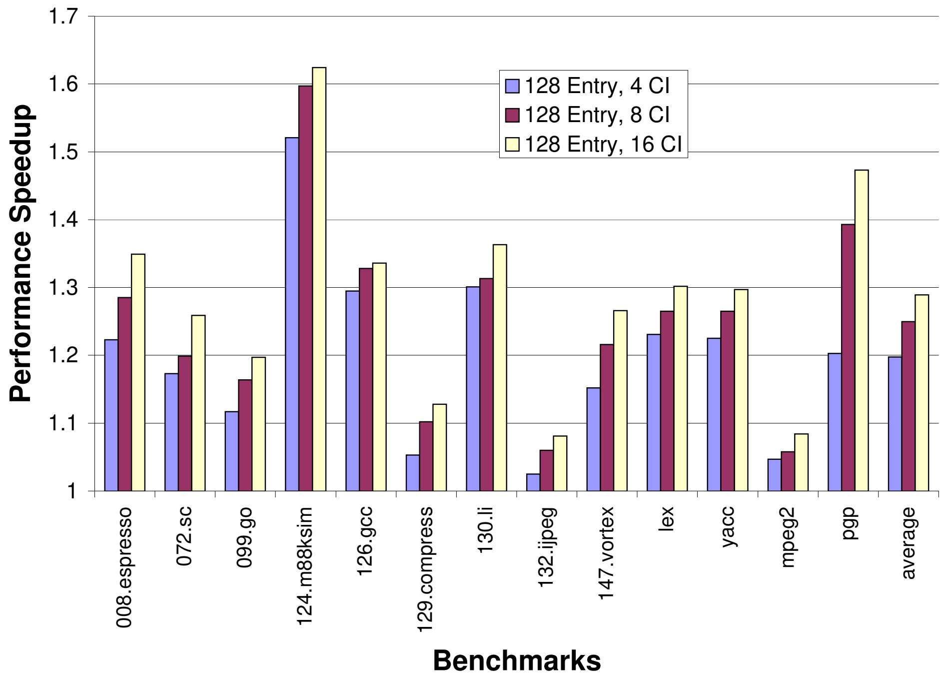


Figure 4.8 Speedup for processor with CCR support varying the number of instances.

instances. On average, a processor with 128 computation entries has speedups of 20% for computation entries with 4 CIs, 25% for 8 CIs, and 30% for 16 CIs. The reuse of computation using the CCR approach is most effective for *124.m88ksim*, where there are a number of substantial computations that are frequently reused. Variation in the number of computation instances substantially increased the performance speedup of *pgp*. This was mainly due to the type of computations being reused. In this benchmark, a number of stateless computation regions were formed using the heuristics based on average reuse occurrence, but the computations have considerable dynamic variation. A large number of computation instances is able to effectively handle this variation. Overall, the average performance improvements indicate that the 128-entry computation buffer with 8 CIs per entry is potentially the most cost effective. In improving the processor performance, the CCR approach eliminates on average 40% of the dynamic instruction repetitions that occur on the base processor configuration.

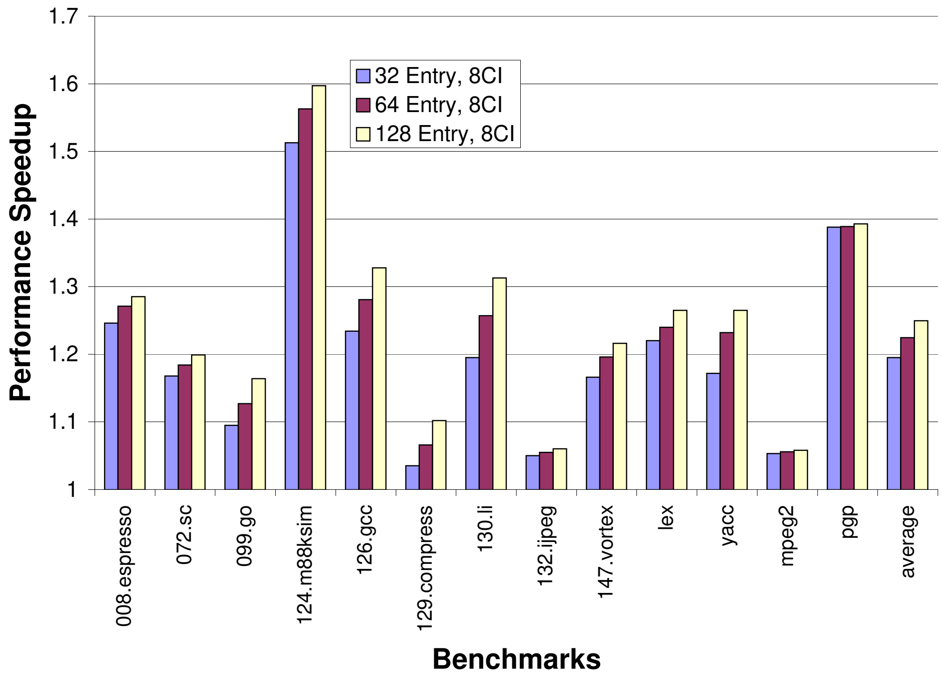


Figure 4.9 Speedup for processor with CCR support varying the number of entries.

The next CRB variation considered is the number of computation entries. Figure 4.9 illustrates the effect on performance of varying the number of computation entries. For a model with 8 computation instances, the average speedups are 20% for 32 computation entries, 23% for 64 computation entries, and 25% for 128 computation entries. The benefits of reuse are sustained for even a small number of computation entries. On average, the majority of benchmarks are characterized by a small number of reusable computations that account for a large portion of the overall execution time. This indicates that the amount of region-level reuse can potentially be exploited with a moderate number of computation entries.

Reusable Computation. Several important aspects of the computations being reused in the CCR approach were investigated. The first is the class of computation and the second is the type of computation. The classification of computation is either stateless (SL) or memory dependent (MD). The classes can be further subdivided into groups indicating the input type of the computation. The group naming convention of SL_{*num_input*} is used to indicate the group of stateless computations dependent on *num_input* registers. Similarly, MD_{*num_input*}_{*num_mem*} is used to indicate the group of memory dependent computations dependent on *num_input* registers and *num_mem* distinguishable memory structures. The number of distinguishable memory structures for each MD group is determined by the compiler's region formation heuristics. Overlapping group results are included within some of the group entries. For instance, group SL_8 includes the computations of SL_7 but not SL_6 since the group SL_6 is also presented. Similarly, MD_6_1 includes all computations dependent on a single memory structure and up to six register inputs.

Figures 4.10 and 4.11 compare the static and dynamic distribution of seven groups of computations that account for the most reuse of program execution. Figure 4.10 presents the static distribution of the computation groups. On average, nearly 90% of the computations are included within the seven selected groups. The distribution indicates that stateless computations account for an average of 65% of the static computations created by applying the current RCR heuristics to optimized programs. Evaluation determined that the acyclic formations of the the seven computation groups replace the execution of an average of 10 instructions. Figure 4.11 illustrates the dynamic distribution of the seven computation groups. On average, the dynamic execution of stateless computation regions accounts for 60% of the reuse execution.

Several of the benchmarks are able to effectively reuse computation results stored in memory by employing the computation groups MD_3_1 and MD_6_1. Groups not accounted for by the groups shown in Figure 4.10 include memory dependent computations dependent on higher numbers of input registers that account for less than 15% of the overall computation type. The distributions are shown to indicate the potential for constructing reuse buffers with resource

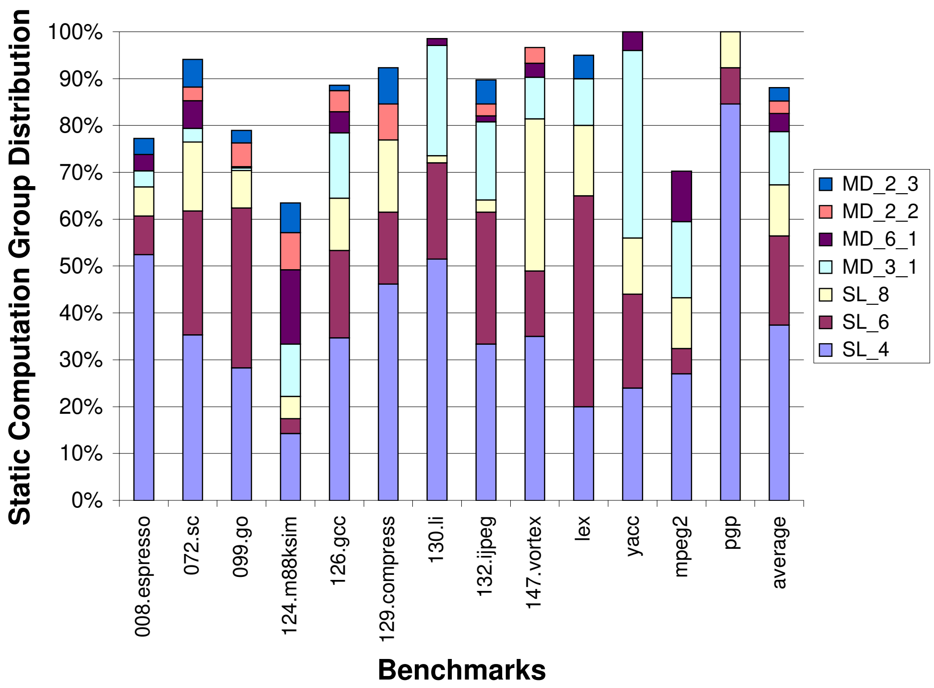


Figure 4.10 Computation group static distribution.

capacities designed to cost effectively attend to the nature of the reusable computation rather than large orthogonal computation entries and instances.

Figure 4.12 displays the average number of instructions reused by each computation group. Only acyclic computation regions are included. The numbers indicate the how reusing computation results rather than re-execution provides improved performance since the many instructions are not fetched from the memory system and are not executed in processor resources.

The observed variation behavior of the computation groups is presented in Figure 4.13. The variation behavior is calculated by observing the region execution over many intervals and collecting the number of computation instances used within that interval. Each column includes the distribution of the computation group's execution spent in intervals requiring 1, 2, 4, 8, or 16 computation instances. The results indicate that a high percentage of all of the computation groups have little variation. The most computation variation is observed in group

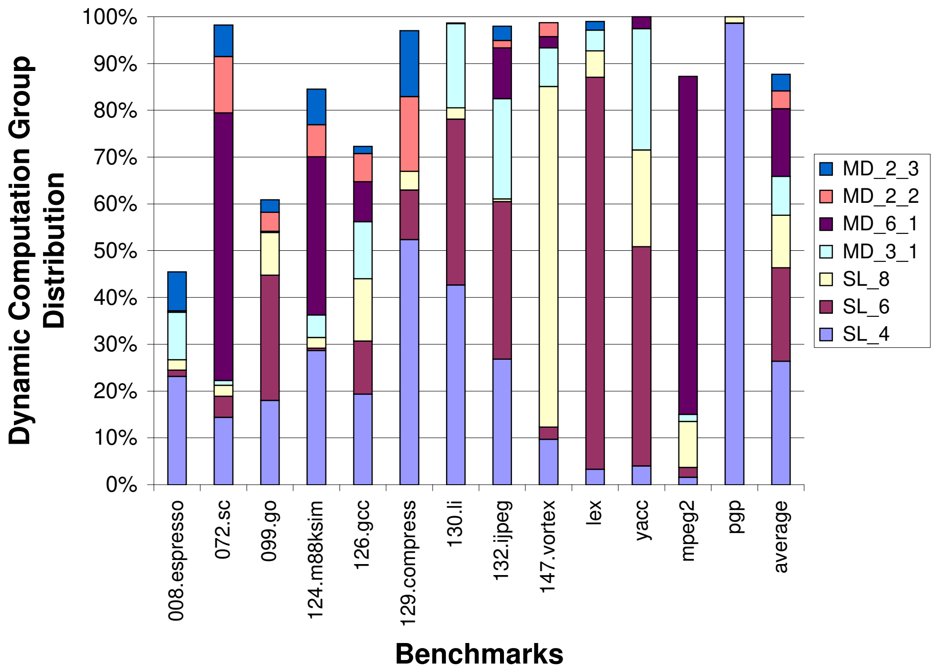


Figure 4.11 Computation group dynamic distribution.

SL_8, in which nearly 70% of executions require more than 16 instances. The variation of the computation groups can be used to effectively design reuse hardware.

Figure 4.14 presents the amount of reuse execution distributed by the percentage of active computations. Four sets are used to indicate the amount of dynamic reuse generated by 40% of static computations. Each set includes 10% of the computations which are sorted by their contribution to the total reuse execution. For instance, TOP 10% indicates the reuse attributed to the top 10% of contributing computations. The cumulative results indicate that 40% of static computations account for nearly 90% of total reuse. *129.compress* is one of the few benchmarks for which different reuse distributions exist. In this case, each of the program computations are closely weighted in the amount of reuse execution that they contribute. Nevertheless, the average characteristics of Figure 4.14 are good indicators of the possibility of exploiting redundancy with limited hardware.

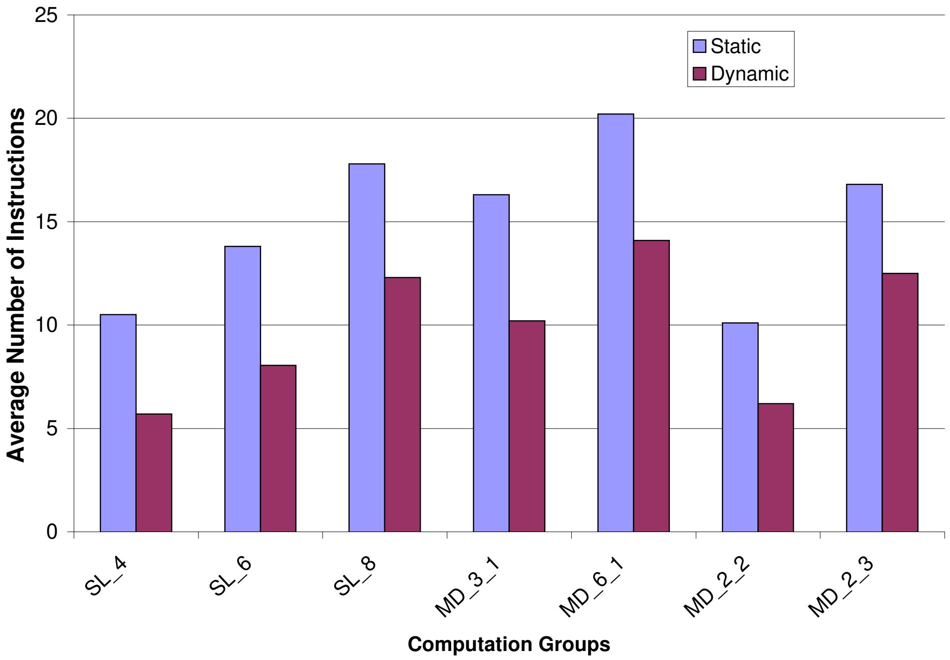


Figure 4.12 Average number of instructions reused per each computation group.

The results of Figure 4.14 indicate that a CRB design with lower capacity in the number of entries and the number of computation instances per entry may be the most cost effective design. Using the compiler selection technique outlined in Section 4.4, reductions can be made in the CRB size while achieving approximately the same performance.

Figure 4.15 shows the result of assigning only computation regions with the highest priority to two smaller CRB models, a 16-entry CRB with 16 computation instances per entry and a 32-entry CRB with 8 computation instances per entry. The results of Figure 4.15 also include the compilation model for the same CRB models with all candidate regions. The compiler prioritized regions are able to achieve better performance since there are fewer unsuccessful computation reuse attempts. The observed results indicate that the CRB designs with compiler-prioritized regions achieve about 85% of the performance of their larger counterparts examined in earlier figures. Execution with entry-prioritized regions achieve slightly better performance

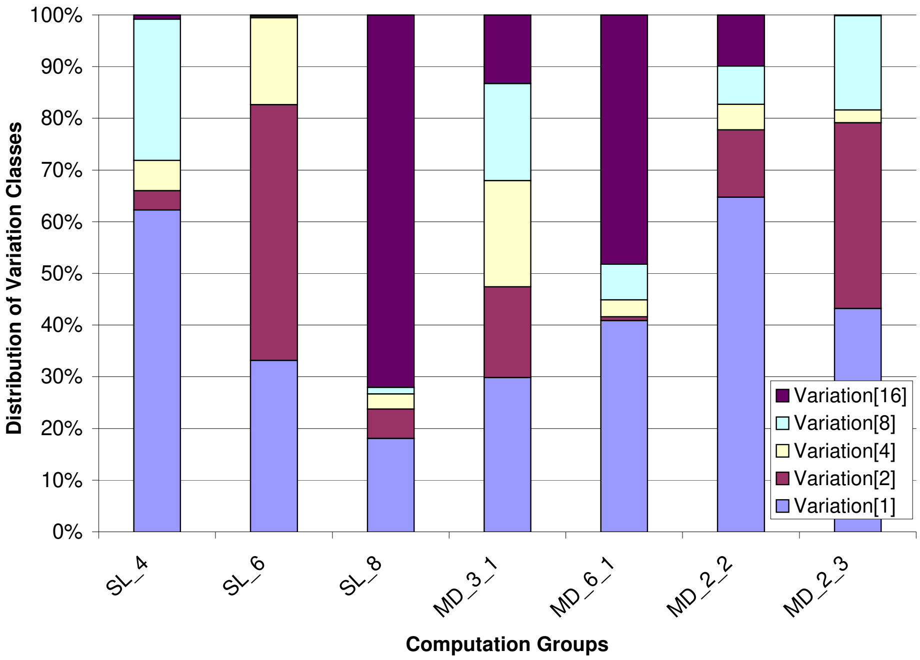


Figure 4.13 Computation group variation behavior.

as temporal locality of the region execution naturally reduces the CRB contention. Selecting regions which can operate effectively using a limited number of computation instances amounts to less performance since a region with high variation will not be selected at compile time. These regions may experience periods with little variation and may in fact be able to eliminate redundant execution with a limited set of computation instances per entry. This motivates the work of the next chapter, where the region selection is performed at run time.

Evaluation of CRB Microarchitecture Alternatives. The alternatives for validating the computation entries within the CCR microarchitecture are evaluated in this section. To begin, Figure 4.16 indicates the distribution of added latency of the reuse instruction when the input register results are in-flight and cannot be used to validate the computation results. Nearly 85% of the computation executions are successful in validating results within the base four stage process. Less than 7% of the executions are spent in delay of more than three cycles.

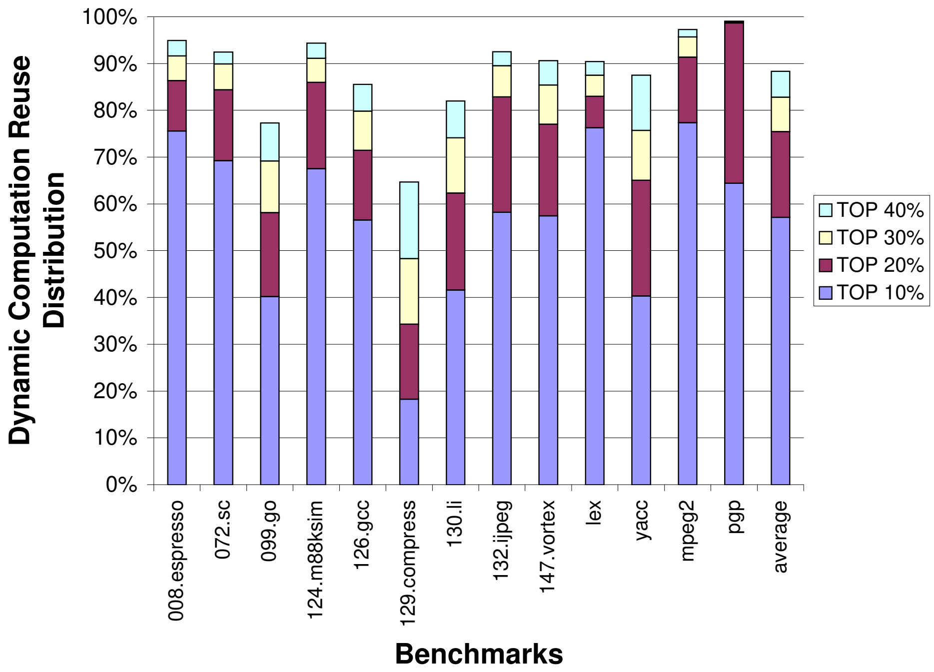


Figure 4.14 Dynamic reuse distribution.

Although the results appear promising, as noted in the microarchitecture discussion, there is still sufficient design space to improve the overall base access latency of the CRB.

The primary design alternative of the CRB is the Shared-Instance CRB. Figure 4.17 evaluates a 32-entry base CRB with 16 computation instances per entry against a 32-entry SI-CRB with 16 register positions and 8 value elements per register position. The hardware costs of these designs are further discussed at the end of this section. The overall results of Figure 4.17 indicate that the SI-CRB can achieve nearly the same performance as the base CRB approach. In the cases of *129.compress* and *pgp*, performance for the SI-CRB suffers because there is not a high concentration of shared values among computation instances. Increasing the number of value elements per entry would eliminate this penalty. However, by organizing the SI-CRB with the evaluated parameters, the SI-CRB design requires about half the hardware cost and achieves the same relative performance. It is also important to note that since both methods

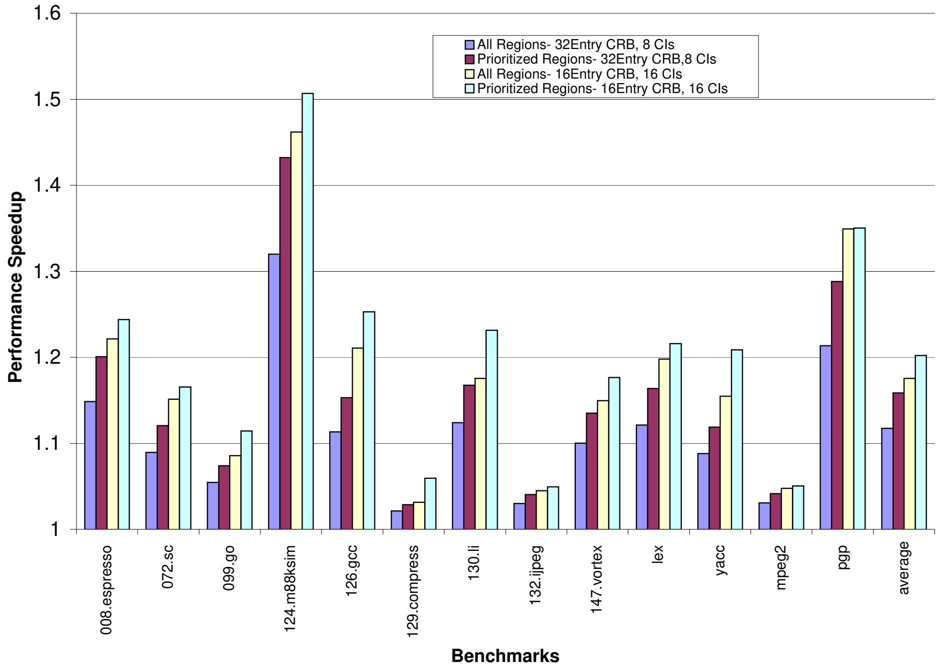


Figure 4.15 Compiler allocated computation regions based on priority.

treat memory-dependent regions identically, the performances for memory dependent regions are equivalent in either model.

Figure 4.18 illustrates the overlap in values of the computation instances held for computation regions. The percentage is evaluated by determining the numbers of register inputs of a region that could share value elements (for the same respective register index) with a region already in the computation reuse buffer. The percentage is weighted by the number of input and output values matched with an existing computation instance. The regions have at most 8 register input or output values, and can have as few as 0 register inputs (strict memory-dependent regions). Overall, the distribution of input and output value overlap illustrates that there is a potential for achieving nearly a 30% reduction of the CRB resources. The high percentage indicates how the SI-CRB is able to have limited hardware support and still maintain the base CRB's representation of the region's computation instances.

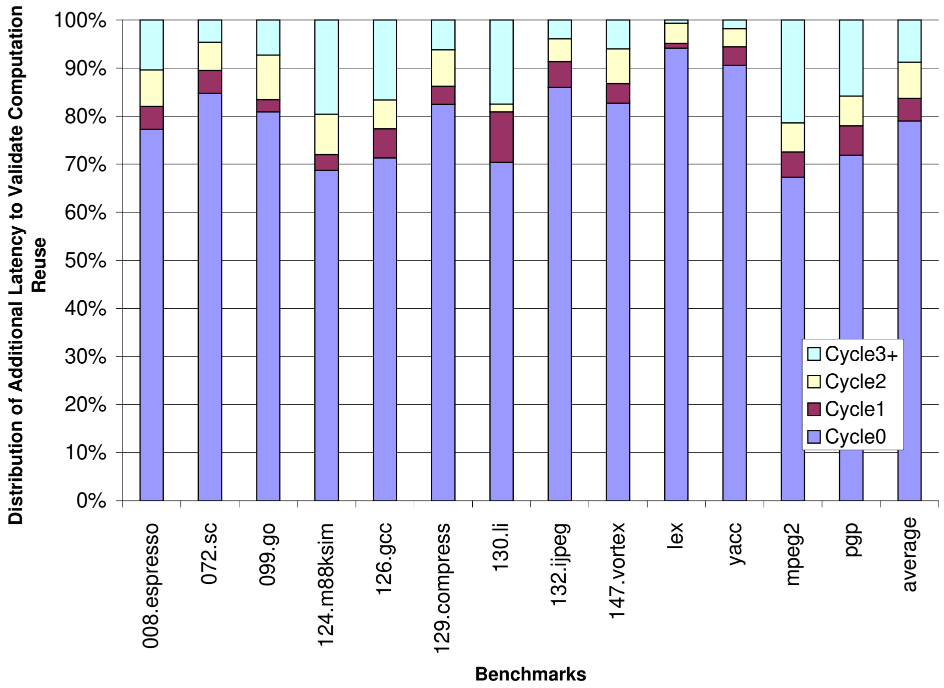


Figure 4.16 Performance degradation for increased computation validation latency.

The results of Figure 4.18 indicate that the computation instances of many of the region executions of *129.compress* and *pgp* could not be efficiently represented using the SI-CRB configuration. Furthermore, in separate experiments, the overlap of register indexes was very high, indicating that since the primary deficiency of the SI-CRB design is handling control-intensive regions with greatly differing input registers, the design will be able to achieve cost-effective performance improvements over the base CRB.

The general microarchitecture alternative that can reduce the CRB microarchitecture design complexity is the history-based computation instance entry. As mentioned, the primary trade-off is a dramatic reduction in the amount of parallel comparisons at the cost of extending the overall latency for validating reuse. Figure 4.19 illustrates the performance of the CCR scheme when additional latency is attributed to the computation result validation associated with validating stages of the computation instances. Three schemes with various numbers of

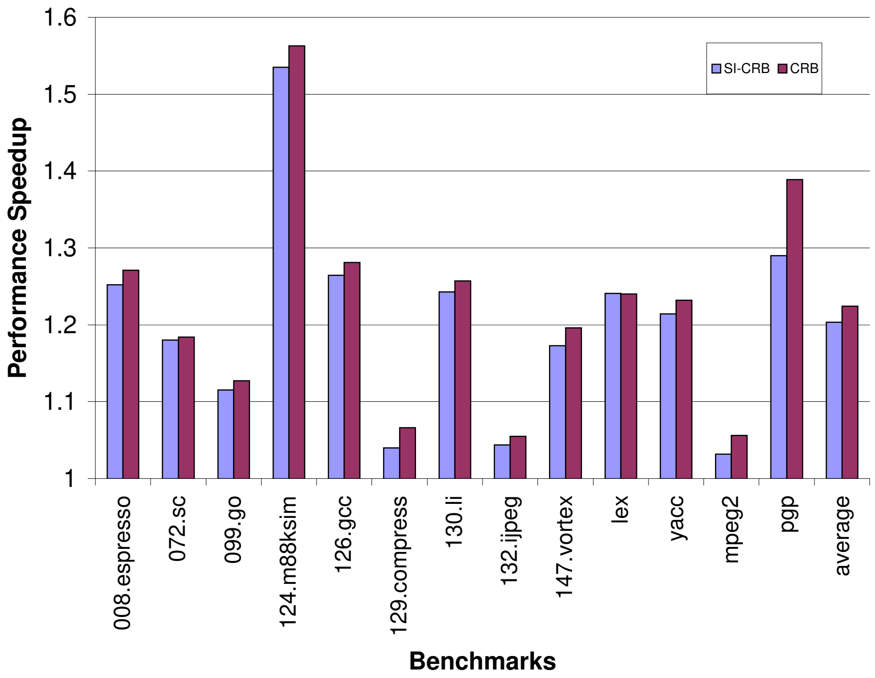


Figure 4.17 Speedup for the base CRB model and SI-CRB model over base architecture.

history frames are evaluated. The first scheme evaluates all of the computation instances in a single cycle, using a single history frame. This history model is the same proposed model of the general CRB. The second history scheme has a history queue with two computation instance frames. The two frames allow access to the most recent instances in a single cycle and the other instances require additional access time. The final scheme has a history queue that is partitioned into four frames, each section being ordered by history of the recent instances. The most recent sections are accessed in a single cycle, while the other stages require additional cycles based on their relative order.

The results of Figure 4.19 indicate that any delays in issuing the computation can be very costly to computation reuse performance. A few benchmarks evaluated using the CRB with four history frames no longer achieved performance speedup over the processor model without any

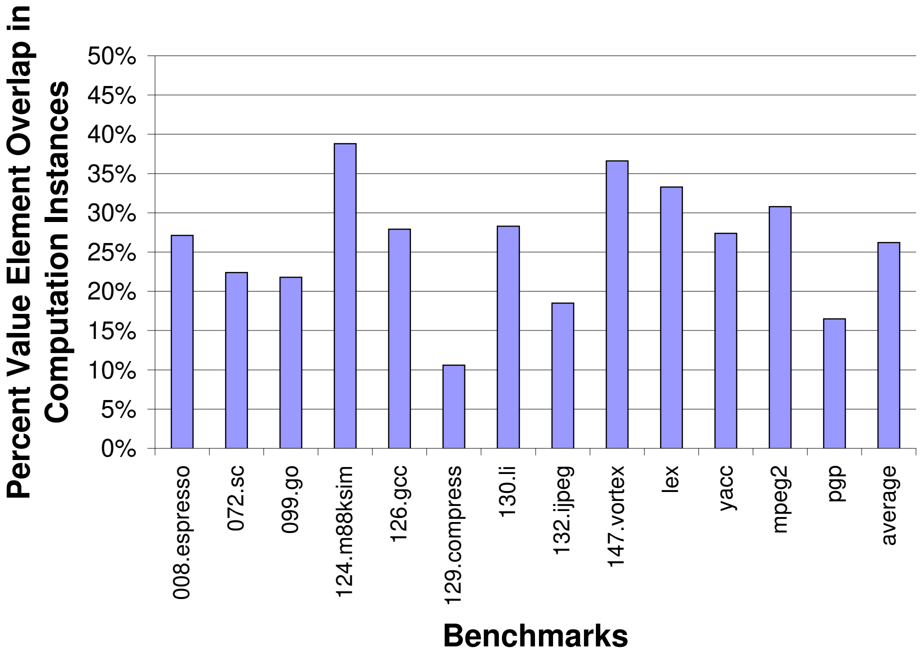


Figure 4.18 Percent overlap of computation region register values with regions already occupying CRB.

computation reuse support. On average, history-based access using two frames of computation instances reduces 20% of the performance improvement of the proposed CRB model.

The result of using the computation anticipation technique to reduce the dispatch latency of computation instances shows the most promise of the alternative methods of validating computation results. In this model, the selected computation entry is able to speculatively issue results of computation instances based on the previously observed access patterns. The access patterns that indicate one instance being repeatedly accessed are used to confidently predict that the pattern will persist. Figure 4.20 shows the accuracy of using the confidence mechanism to predict the outcome of the computation reuse attempt. The mechanism can correctly predict an average of 50% of computation instances, allowing the computation results to be successfully speculated within the microarchitecture pipeline. Nearly 30% of the time,

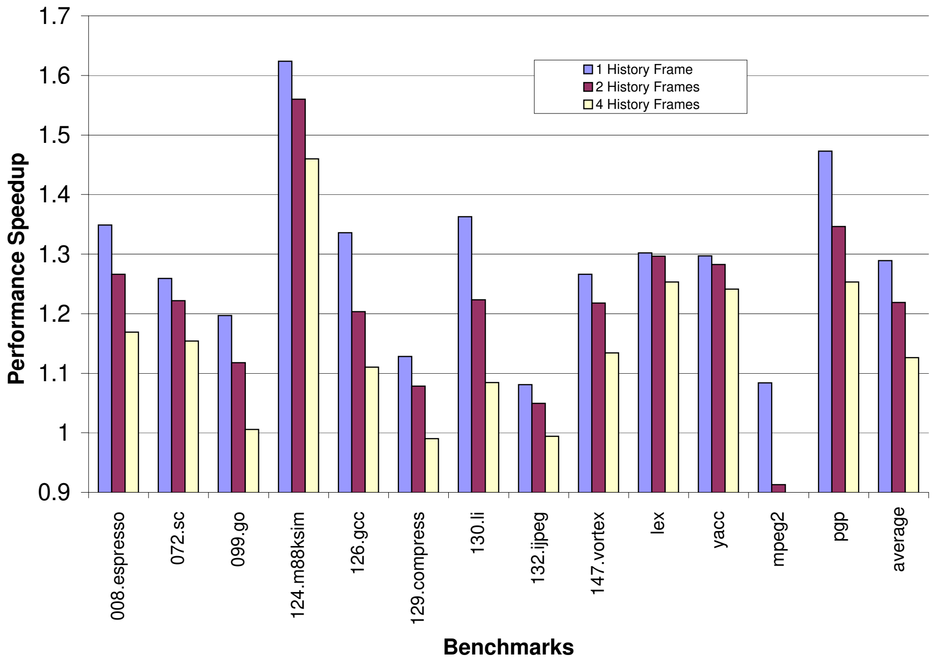


Figure 4.19 Performance of history-based computation instance array.

the computation variation cannot be confidently predicted by the anticipation mechanism. Performance results indicate that the computation anticipation results in 3-5% performance improvement.

General Approach Evaluation. To evaluate the overall CCR approach, it is important to understand which components in the scheme cause the most disruption to achieving the best performance. Three conflict types govern efficient reuse in the CCR approach: computation entry miss, computation instance miss, and false (unnecessary) compiler-direction memory invalidation. Figure 4.21(a) shows the breakdown of these conflicts for the 128-entry computation buffer with eight computation instances. For all benchmarks, the computation instance miss conflict dominates the percentage of time that a designated computation region can not find a previous result. For only two benchmarks, *099.go* and *126.gcc*, do the conflicts caused by computation entries being replaced account for greater than 15% of the overall computation

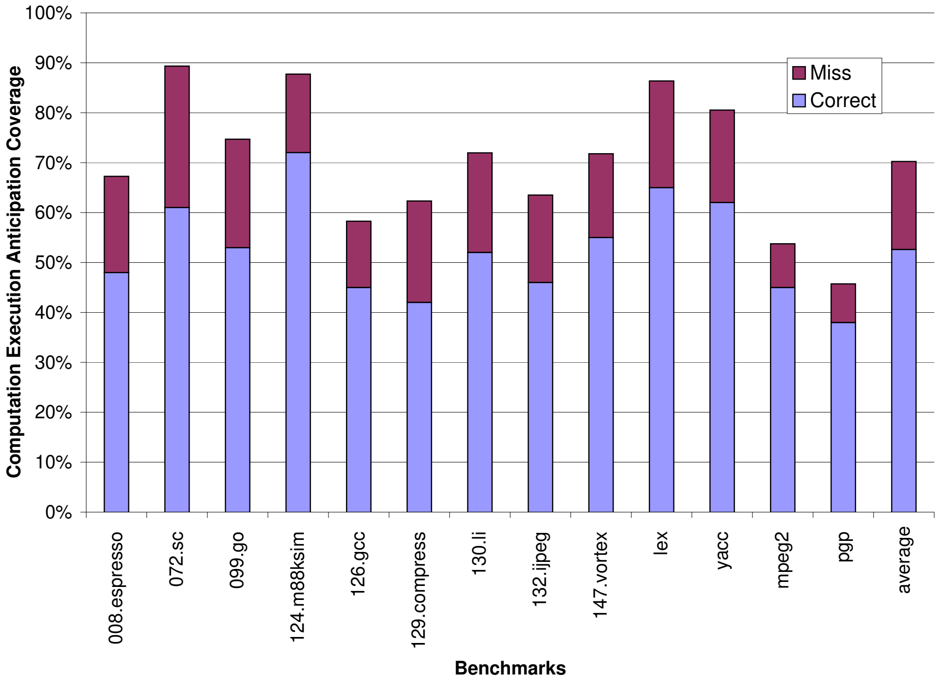
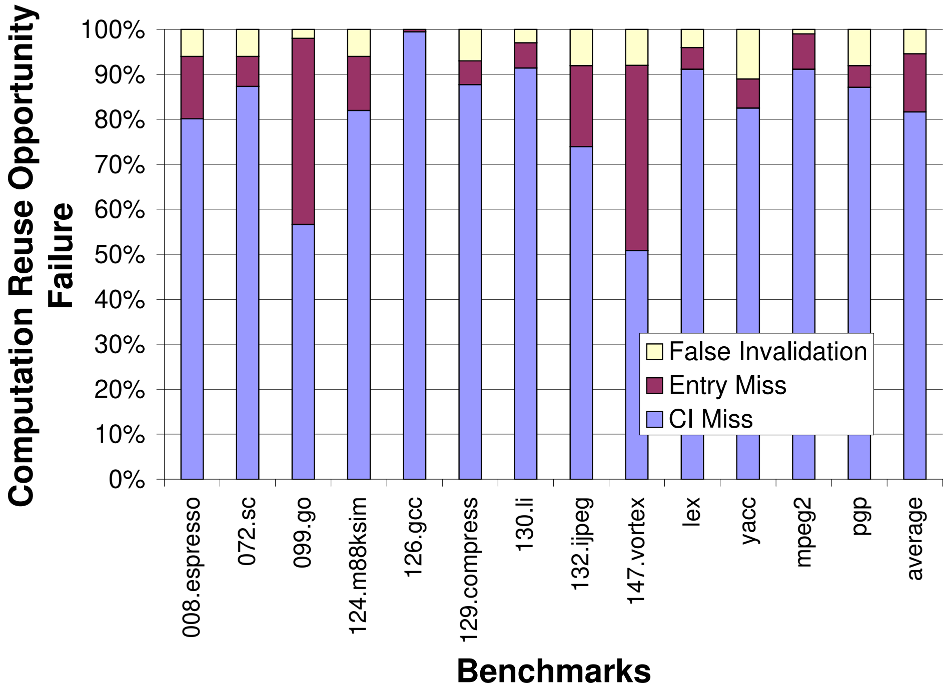


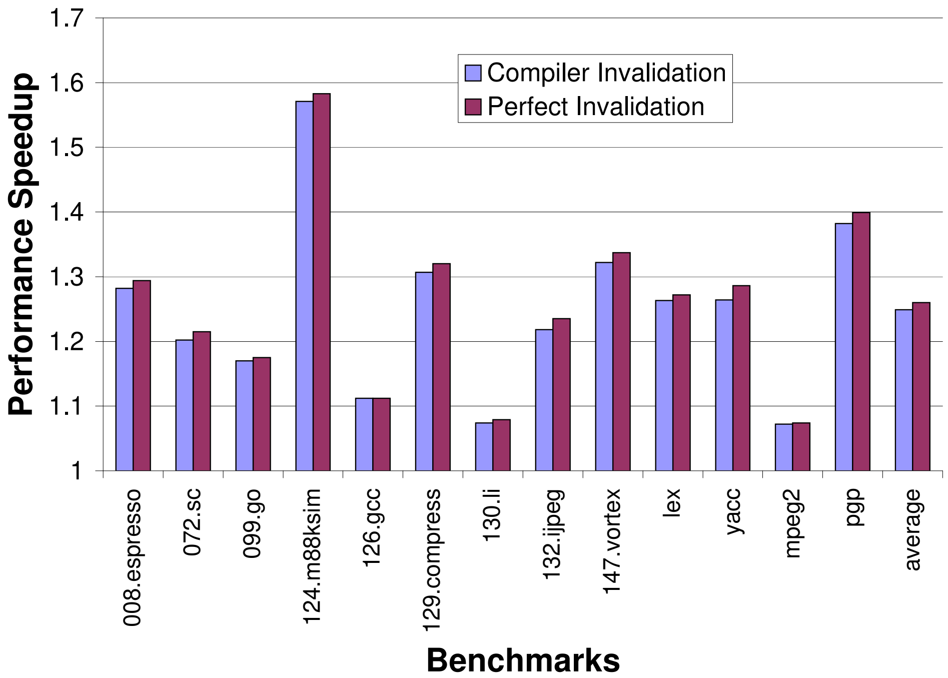
Figure 4.20 Accuracy of anticipating computation instance execution behavior.

conflicts. This result is not surprising since these benchmarks are two of the largest programs and have a significant number of compiler-directed computations contending for the computation buffer resource. A false invalidation is caused by the compiler's inability to determine the relationship between loads of a computation region and certain program stores at compile time. Figure 4.21(b) shows the performance improvement based on eliminating all of the false memory invalidations. On average, the performance of pointer-intensive programs of *130.li* and *147.vortex* appear most affected by the false invalidations. Overall, context-sensitive analysis shows little precision inefficiencies in the benchmarks studied. The performance results indicate the importance of supporting interprocedural alias analysis within the compiler and its direct influence on the effectiveness of the proposed approach.

Due to the nature of determining reusable computation at compile time, the performance potential of the CCR framework depends on utilizing effective heuristics and having accurate



(a)



(b)

Figure 4.21 Distribution of computation reuse conflicts (a) and speedup assuming perfect compiler-directed memory invalidation (b).

run-time estimation of program behavior. To determine the significance of using value profiling information outlined for the CCR approach to make these decisions, the performance of an input data set different from the one used to determine reusable computation was measured. The training input set is used to select reusable computation regions; and the training and reference input sets were then evaluated, and the results were compared to each input's respective base performance. Figure 4.22 examines these results for a 128-entry CRB model with 8 CIs per entry. The average performance speedup for the training input set is 26%, and the average for the reference input set is 23%. Although on average there is a reduced relative speedup for the reference input set, several of the benchmarks achieved higher relative speedups for the reference input set. In addition, the average amount of instruction-level redundancy eliminated using the reuse architecture averaged 33%.

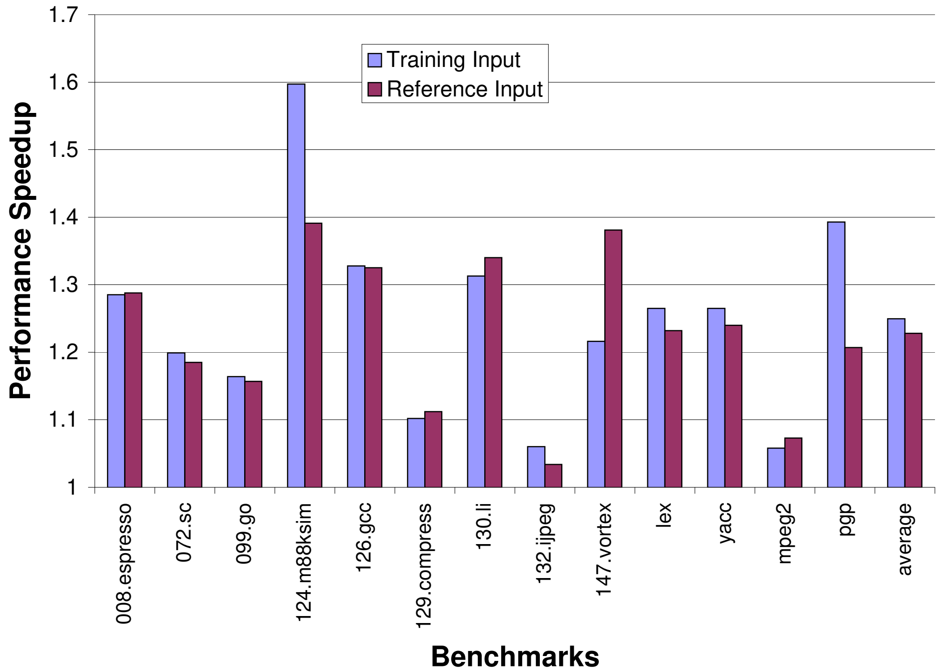


Figure 4.22 Performance for training and reference input data sets.

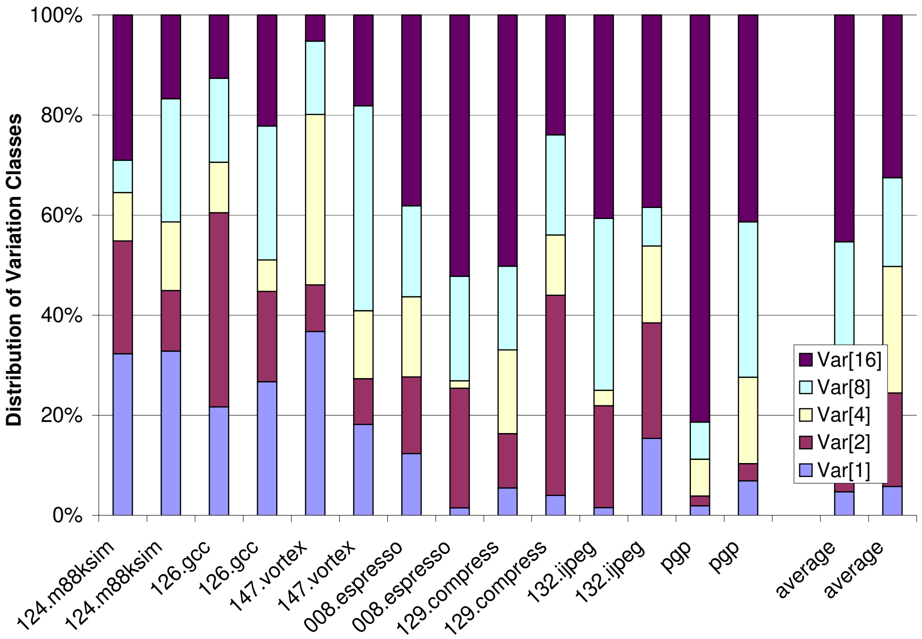


Figure 4.23 Variation of computation instance behavior for multiple input sets.

Overall, the moderate performance speedups of Figure 4.22 reinforce the strategy of using profile information to aid the compiler in detecting opportunities for computation reuse. And secondly, for the benchmarks achieving lower relative speedups for the reference input, the percentage difference is moderate, indicating agreeable execution behavior across input sets. Figure 4.23 illustrates the variation in region requirements observed for the two input sets evaluated for a subset of benchmarks. The bars indicate the type and amount of invariance occurring. Generally each application sustains relatively the same variation-based reuse behavior between the input sets, indicating that it is possible to characterize a program's behavior by the redundancy found within its execution.

Hardware Implementation Costs. Table 4.1 represents a hardware cost estimate of the CRB design. This table accounts for bytes of hardware memory, but does not include wiring or logic gate costs. The largest cost comes from maintaining the computation instance register index and value. Experimental results in Chapter 5 will illustrate the performance of the CRB approach evaluated for different levels of hardware resources.

Table 4.1 Hardware cost expressions and cost for models evaluated.

Scheme	Cost Expression	Cost (in bytes)
CRB	$N(Entry + CI(CI\_COST + Num\_Input(Input\_Cost) + Num\_Output(Output\_Cost)))$	42624
SI-CRB	$N(SI - Entry + CR(RECORD\_COST + 2 * Register\_Positions * Num\_Elements)+$ $2 * Register\_Positions(Num\_Elements * Element\_Cost + Element\_Overhead)$	18092

Var	Definition
N	Number of computation entries
$Entry$	Size of computation entry information
CI	Number of computation instances per entry
CI_COST	Size of computation instance information
Num_Input	Number of input registers in computation instance
$Input\_Cost$	Size of input register information (value, reg index, valid)
Num_Output	Number of output registers in computation instance
$Output\_Cost$	Size of output register information (value, reg index, valid)
CR	Number of computation records
$SI - Entry$	Size of computation entry information for SI-CRB
$RECORD\_COST$	Size of computation record information
$Register\_Positions$	Number of register positions
$Num\_Elements$	Number of value elements
$Element\_Cost$	Size of element information (valid)
$Element\_Overhead$	Size of maintaining element information by register position

Finally, in motivating the proposed compiler-directed approach, Figure 4.24 illustrates the ratio of the hardware facilities required to extract a single computation instance for each computation region of Figure 4.24 using hardware-only relative to compiler-directed techniques. The estimations are generated by calculating the amount of information recorded for reuse and not for detecting potential reuse. The compiler-directed approach uses static analysis to understand the live-out of a region and only records those identifiers and values, whereas the hardware-only approach must retain every register updated within the region. Thus, the information recorded for compiler-directed methods is significantly lower for the same reusable

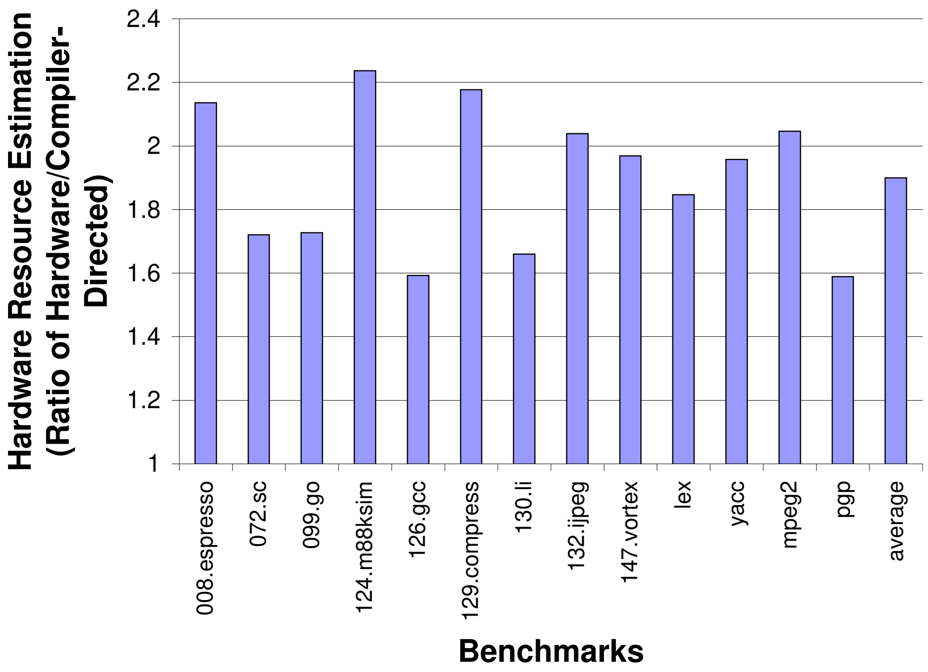


Figure 4.24 Resource estimation for hardware exploitation of regions.

sequences. Figure 4.24 clearly shows an advantage for the compiler-directed approach to reduce the hardware overhead necessary to reuse large amounts of computation.

4.6 CCR Summary

In this chapter, the base compiler-directed approach for exploiting dynamic redundancy was presented. The approach uses the compiler to decide and annotate code segments with the potential for reuse. This enables the underlying hardware architecture to capture the reuse potential in regions of code rather than basic blocks and individual instructions, thereby exploiting more substantial opportunities. Additionally, the approach captures the redundancy of high-level computations that do not possess instruction-level repetition. The effectiveness of the proposed approach was evaluated using several different resource models, varying the size of the computation reuse buffer and the number of computation instances recorded for each computation. The resulting speedup of the approach measured with a moderate architecture model (128-entry CRB with 16 computation instances per entry) achieved an average 30% speedup.

CHAPTER 5

DYNAMIC COMPUTATION MANAGEMENT SYSTEM

5.1 Introduction

Compiler-directed computation reuse in the CCR approach enhances program execution speed and efficiency by eliminating dynamic computation redundancy. A major limitation of the work thus far is that the compiler relies on value profiling to identify reusable regions, making it difficult, if not impossible, to deploy the scheme in many software production environments. This chapter presents a new model where a novel hardware mechanism alleviates the need for value profiling at compile time. The compiler is allowed to designate reusable regions that may prove to be inappropriate. The hardware mechanism monitors the dynamic behavior of compiler-designated regions and selectively activates the profitable ones at run time. The proposed design makes more effective utilization of hardware buffer resources, achieves rapid employment of computation regions, and improves reuse accuracy, all of which promote more flexible compiler methods of identifying reusable computation regions. Experimental results show that even a modest hardware monitoring mechanism can greatly increase the benefit of the CCR approach.

This chapter addresses the limitations of the original CCR approach by presenting new hardware-based enhancements for accurately and effectively determining reusable computations within program execution at run time. To support the targeted enhancements to the

CCR approach, the proposed hardware uses three integrated components. The first hardware structure, called the *Reuse Sentry*, detects regions with significant execution frequency and controls the deployment of active computation regions. The *Evaluation Buffer* evaluates candidate computation regions for potential reuse. This structure monitors selected regions to determine if the regions should be placed in the Computation Reuse Buffer (CRB). Finally, the third component, called the *Reuse Monitoring System*, examines the behavior of computation regions in the computation buffer and makes assessments on the allocation of CRB resources. Overall, the system introduces architectural support for eliminating dynamic redundancy due to inherent aspects of programming languages and application workloads.

The remainder of this chapter is organized as follows. The following section will motivate the proposed hardware components. Section 5.3 describes the system for performing run-time management of computation reuse regions. The effectiveness of the proposed system is evaluated in Section 5.4. Finally, the chapter is summarized in Section 5.5.

5.2 Motivation

The Compiler-directed Computation Reuse (CCR) approach [38] uses the compiler to identify code regions whose computation can be reused during dynamic execution to eliminate dynamic redundancy. The instruction set architecture provides the interface for the compiler to communicate the scope of each region to the hardware. During run time, the microarchitectural components of the approach record the execution results of the potential reuse regions. Experimental results show that the approach can eliminate a large number of dynamic instructions in general applications, resulting in much higher execution speed and efficiency.

In the base CCR approach, code segments called Reusable Computation Regions are selected at compile time using estimates of the expected reuse that will occur during run-time execution. Because the hardware structure of the base CCR approach always attempts to reuse previous computation results for all annotated regions, these regions contend for the same computation reuse resources regardless of each region's benefit. Thus, the compiler must select only those regions whose reuse at run time will result in the most benefit.

Figure 5.1 shows the performance achieved by a six-issue processor with a 32-entry CRB hardware structure with 16 computation instances per entry and with a CRB with infinite resources, both over a base processor without reuse support. The CRB with perfect resources has the capability of maintaining all of the previous execution results for every computation region selected at compile time. On average, the 32-entry CRB design captures about 40% of the potential speedup available using infinite resources. The gap is due to the fact that the compiler is faced with the undesirable trade-off between missing reuse opportunities and exhausting buffer resources. The substantial potential performance benefit provides the initial rationale for enhancing the base CCR scheme to allow the compiler to be much more aggressive.

Although the CCR approach can eliminate significant amounts of redundant execution, there are additional opportunities that require capturing elements of program behavior. For instance, virtually all programs go through a series of stages during execution. A stage is characterized by changes in the execution properties for code, the data, or both. Similarly, programs such as compilers, interpreters, and graphics engines exhibit phase behavior, having different modes of operation for different inputs [52]. The use of profile information by compiler-directed mechanisms can hide opportunities since profile-guided decisions may not be representative of all program workloads. More importantly, the use of profile information in many systems is

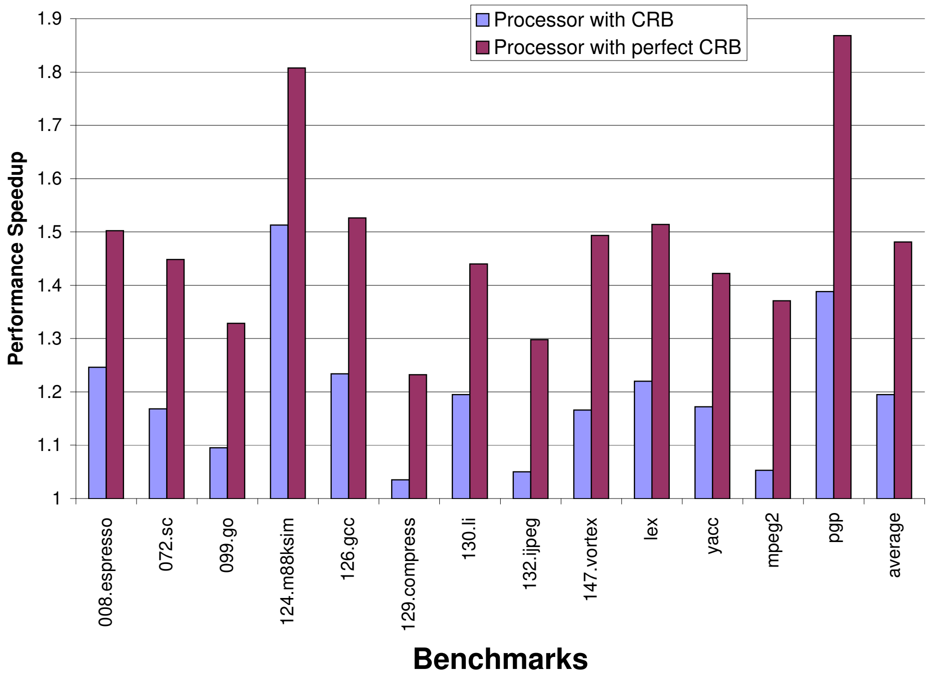


Figure 5.1 Performance for processor with base 32-entry CRB and perfect CRB.

not feasible due to constraints in software development. For these reasons, it is imperative for a system to adapt to variations in program behavior.

In analyzing run-time computation reuse, experimental observations indicate that reuse behavior often occurs in distinct phases. A reuse of any of the cached computation instances contained in the CRB over an interval of time represents an opportunity to eliminate computation redundancy. Periods of region executions with successful reuse are called *reuse intervals*. An enhanced, dynamically managed CCR scheme can take advantage of reuse intervals by favoring the utilization of the computation buffer resources for those regions. Ideally, only computation regions demonstrating reuse success would be deployed at run time.

An example of reuse interval behavior can be seen in *008.espresso* running the bca training input from the SPEC92 benchmark suite. The reuse execution collected over a period of 2 million instruction executions for three computation regions is shown in Figure 5.2. Each point indicates whether the region had a computation miss (no cached computation results), a computation hit (a matched computation result from a previous execution), or no attempt (the region was not executed at that time). All three regions exhibit initial periods during which the computation instances are being stored for future execution. These periods are indicated by the excessive percentage of reuse misses.

The behavior of the first region indicates two medium-sized reuse intervals with intervening periods of reuse misses. The second and third regions have longer reuse intervals, which occur after the initial cold start period. However the third region experiences a long period of time without any execution. The identification of reuse intervals can be made over different lengths of program execution time. Overall, the behavior exhibited in Figure 5.2 motivates dynamic management of computation reuse since regions execute with periods of reuse and multiple regions compete for the same computation reuse resources.

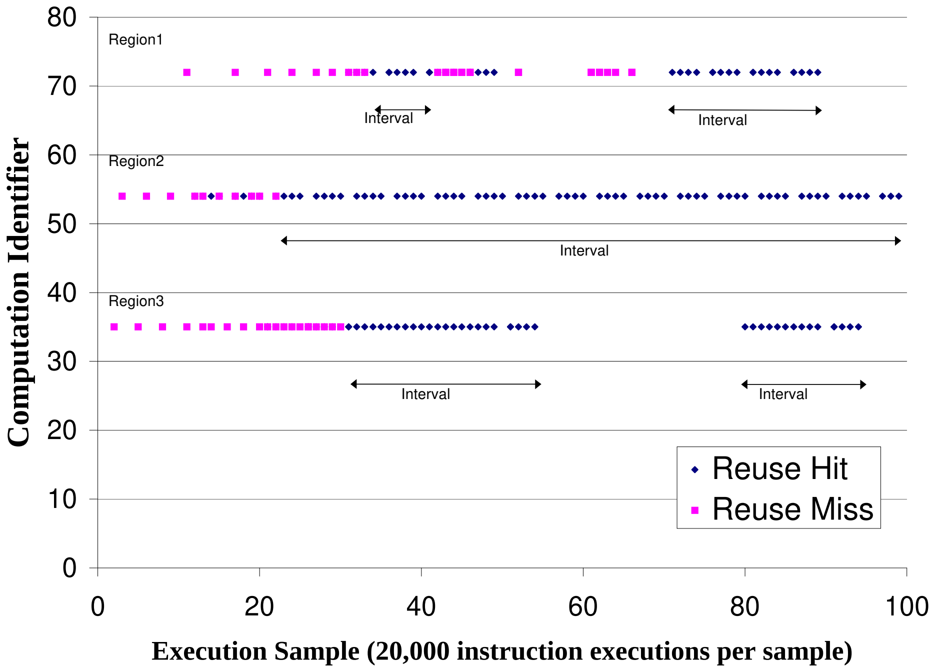


Figure 5.2 Computation regions executed for *008.espresso*. Each data point indicates either a computation reuse miss, computation reuse hit, or no computation reuse attempt.

The third motivation for extending the design of the compiler-directed reuse hardware is the variation in the computation instance capacity required by each region. Generally, the number of computation instances that a region requires over an interval of execution varies with the region type. Regions exhibiting *regular variation* require a set number of computation instances that remains constant over the lifetime of the program. Such regions can be assigned to computation entries with a specific number of computation instances, thereby reducing the hardware costs of providing a large number of instances for every type of region. Conversely, through experimentation an average of 70% of regions have *irregular variation* behaviors that warrant different computation instance capacities at different times. Thus, to effectively utilize computation resources, the CRB needs to adapt its allocation to run-time requirements.

Several proposed architectural techniques can realize significant performance benefits by adapting to run-time variation. Branch prediction and cache management have been the primary areas where run-time information has been applied. However, trends in dynamic optimization [24] and run-time hotspot detection [53] indicate other exciting avenues to exploit run-time behavior. Run-time information allows for a more effective use of processor resources. Likewise, allowing the compiler-directed computation reuse approach to use run-time information would allow it to adapt to program trends. There are several reasons for adapting run-time techniques to the compiler-directed reuse approach. First, the reuse of a computation region can be activated at run time rather than at compile time. This includes using run-time information to direct the deployment of a reuse region. Second, the run-time information can be consulted to effectively utilize a limited number of computation entries by assigning resources to those regions generating substantial performance improvements over a period of time. Third, the run-time variation statistics would allow effective allocation of reuse buffer resources in the presence of hardware with varying recording capacities. Finally, dynamic management techniques allow the compiler to introduce computation regions that are not strictly guided without degrading performance improvements.

The percentage of computation regions with irregular variation requirements is shown in Figure 5.3. For this graph, regions were selected by profile-guided heuristics [38] and then examined for a reuse buffer with one entry per region and 16 computation instances per entry. This model allows the variation behavior of all regions to be studied. Each region was observed over periods of 1024 instruction executions. During each of these time periods, the unique number of computation instances was divided into four bins indicating the number of required instances (2,4,8,16+). If a region executed more than 50% of the time periods in one bin, then the region was noted as having regular variation (stable number of computation instances required). Figure 5.3 indicates the percentage of dynamic region executions in the irregular variation behavior class. On average, 73% of regions require different computation instance capacities. Additionally, only 33% of these irregular regions had greater than 75% of their executions covered by two bins. These results indicate that a method of relocating computation regions to computation buffer entries through program execution may achieve increased utilization under strict hardware constraints.

The final motivation for improvement is that the base CCR approach relies on profile-guided heuristics to identify computations that are potentially reusable during the execution of the program. Specifically, value profiling techniques [2] determine regions called *Reusable Computation Regions* that instruct the hardware how to effectively reuse the computation defined within these regions. Such reliance on value profiling can hamper the use of the base CCR approach in many software production environments.

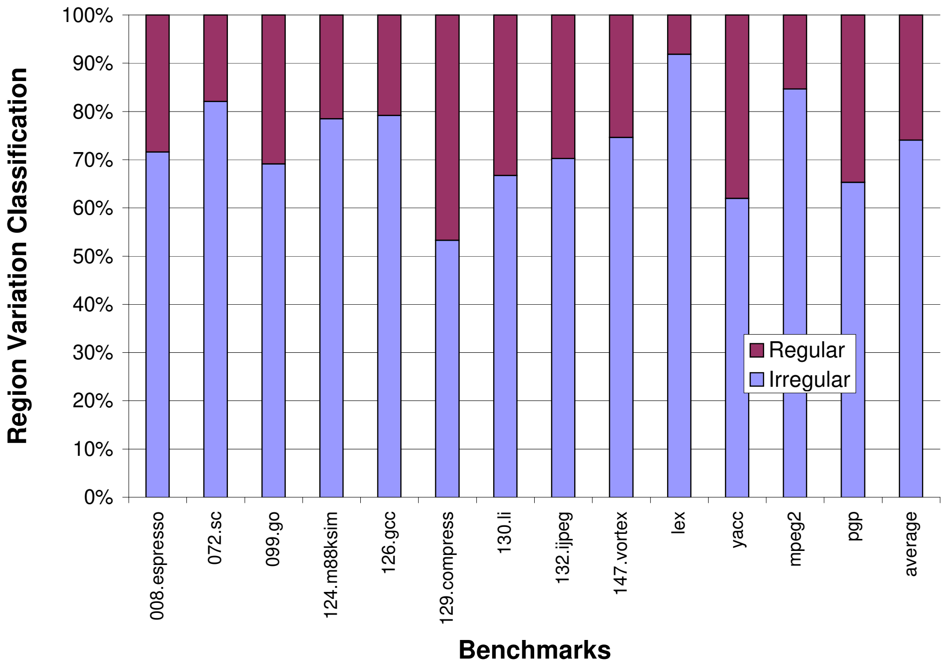


Figure 5.3 Computation region variation behavior.

5.3 Dynamic Computation Management System

The proposed hardware support for dynamic computation reuse management uses three stages to activate a computation region. The stages perform detection, evaluation, and examination of beneficial computation reuse. First, the region must meet a minimal execution requirement to warrant consideration for CRB resources. Second, the region must be evaluated to determine if it is likely to benefit from reuse and placement in the CRB. Finally, the variation of the computation region is examined to determine the most effective way to assign it in the CRB. These stages act as a run-time confidence mechanism to accurately select regions for the computation buffer resources, and result in improved CRB utilization and reuse accuracy. The stages are collectively constructed in the Dynamic Computation Management System (DCMS) which consists of the following three hardware components:

Reuse Sentry (RS) collects execution counts for computation regions and identifies candidate computation regions. The Reuse Sentry contains a Reuse Deployment Buffer to deploy the computation and a Candidate Execution Counter to determine the activity of candidate computation regions.

Evaluation Buffer (EB) evaluates candidate computation regions for potential reuse. The structure monitors regions to determine if they should be placed in the computation reuse buffer to eliminate computation redundancy.

Reuse Monitoring System (RMS) examines the behavior of regions in the computation buffer and assesses allocation of CRB resources. The RMS removes computation regions from the CRB and directs alternate computation regions to utilize the CRB resources.

Figure 5.4(a) shows the base CRB model and Figure 5.4(b) illustrates the proposed dynamic computation management hardware. In Figure 5.4(a), computation regions always attempt to reuse execution by contending for entries in the CRB. In the enhanced design of Figure 5.4(b), the resources are selectively assigned to regions with persistent reuse.

The Reuse Sentry and Evaluation Buffer can be located off the critical path of the processor pipeline because their resources do not directly affect the process of using previously cached computation results. However, the Reuse Monitoring System and the operation of the CRB require close interaction with the processor datapath and cannot tolerate a large access or update latency. This is because the access latency of the computation entries and their respective computation instances is inversely proportional to the performance benefit of the compiler-directed approach [38]. The following sections describe each of the main components of the dynamic management system.

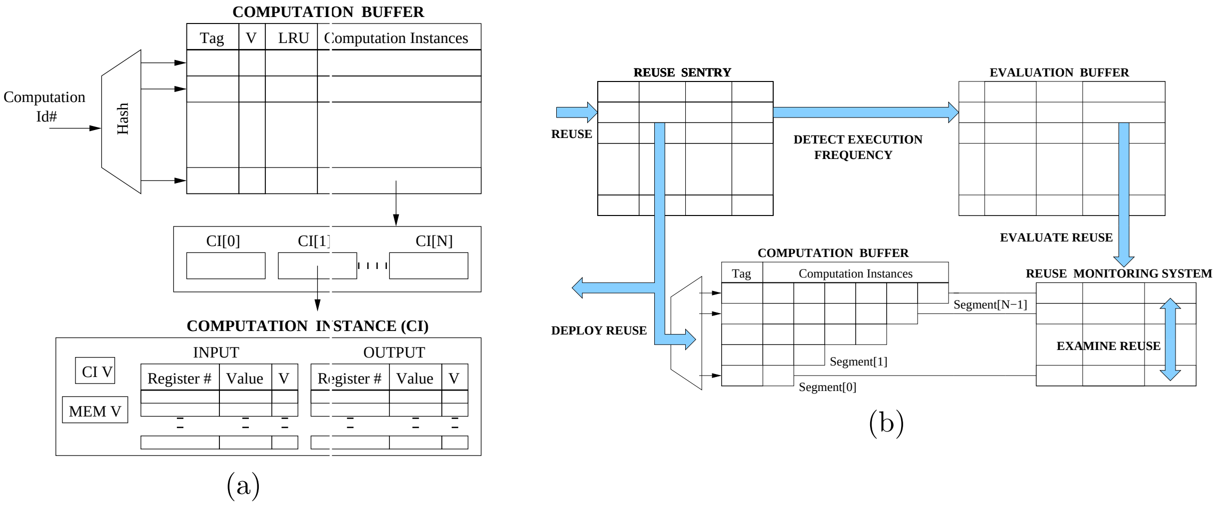


Figure 5.4 The Computation Reuse Buffer (a) and the Dynamic Computation Management System support (b).

5.3.1 Reuse Sentry

The first step in effective management of the computation reuse resources is detection of frequently executing computation regions. Such regions can be easily identified in hardware by detecting a high execution frequency over a particular time interval. By examining all region executions over the same interval, computation resources can be accurately allocated to regions with different requirements.

A structure called the *Reuse Sentry* (RS) collects the execution history and efficiently deploys reuse. This hardware is named for its function as a *sentry*, or guard, that prevents the passage of unauthorized regions. As depicted in Figure 5.5, the RS structure is indexed by the reuse instruction address and contains several fields: address tag, execution count, state information, predicted address, and computation entry index. The RS structure also includes a global *Observation Reset Counter* used to evaluate all region executions. The activities of the RS

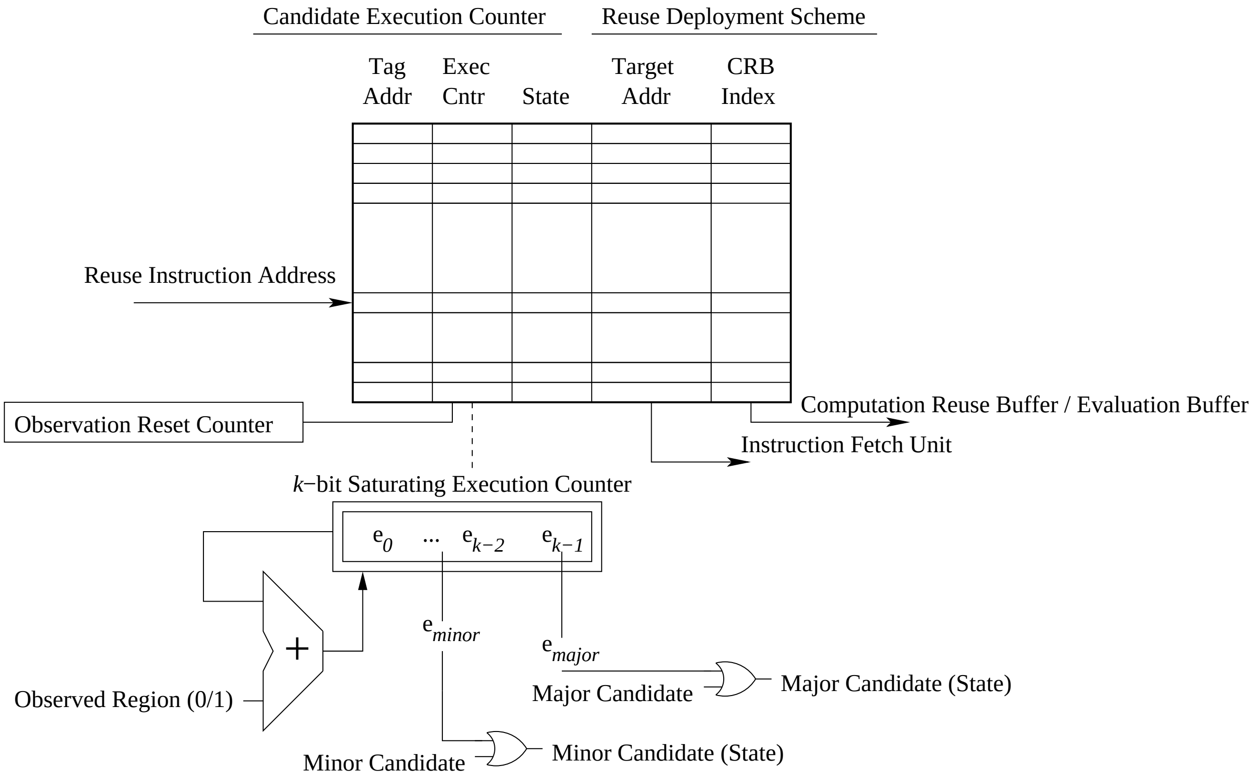


Figure 5.5 Reuse Sentry hardware.

structure are divided between the Candidate Execution Counter and the Reuse Deployment Buffer.

5.3.1.1 Candidate execution counter

The Candidate Execution Counter (CEC) primarily detects regions with frequent execution. Regions that execute often are given higher priority for resources than regions with low frequencies. A secondary function of the CEC is to record the position of each region in the DCMS system. The position follows the transitions of Figure 5.6, which uses four states: *observed*, *candidate*, *active*, and *inactive*.

When the processor executes a reuse instruction, an entry is created in the RS and the instruction address is designated as the index. The initial creation of an entry classifies the

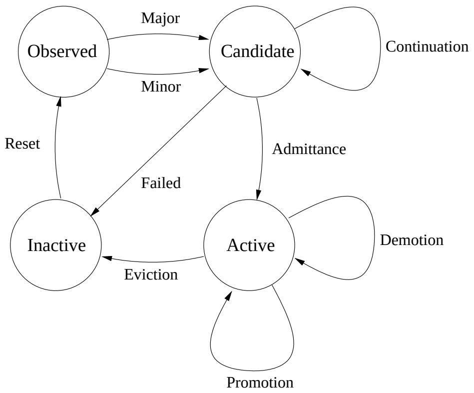


Figure 5.6 Region states and transitions.

respective region as an *observed region.* In this state, the region is being monitored to determine if it frequently executes. Regions that execute frequently over a short time interval are excellent candidates for reuse resources. If executed frequently during the observation state, a region may be passed to the Evaluation Buffer, and is then referred to as a *candidate region.* In order to detect candidate regions, the execution counter of a region's RS entry is incremented on each execution of the region. The execution counter can exceed two predefined levels, the *major* and *minor candidate thresholds*, each of which is associated with a bit in the execution counter. When the counter bit corresponding to a threshold is set for the first time, a candidate flag field is set in the state field for the duration of the observation time. The major threshold indicates regions with a dominant number of executions, while the minor threshold indicates a lower execution frequency. The distinction between major and minor thresholds aids determination of the best candidate regions from the observed regions in a particular interval.

To ensure that only frequently executing regions are marked as candidates, the RS is periodically flushed. The Observation Reset Counter is used to establish a time interval, called the *observation interval*, for periodically refreshing the entries of the RS that have not surpassed any candidate threshold or have been classified as *inactive.* The Observation Reset Counter is incremented each time an inactive or observed region is executed. The inactive classification is assigned by the evaluation and examination stages of the DCMS and is used to designate regions that previously occupied the CRB, but had poor reuse behavior.

When the major candidate threshold is surpassed, hardware logic attempts to add an entry for the region to the Evaluation Buffer (EB) and change the region state in the RS from observed to candidate. If the EB does not have sufficient entries to handle the request, another attempt is made at the end of the observation interval. In this case, the RS is scanned for all entries meeting the major candidate threshold. This means that relative to the observation interval, entries in the evaluation buffer are made both asynchronously and synchronously. If the EB has capacity remaining after obtaining major candidate regions at the end of the interval, then the RS is searched for entries meeting the minor threshold.

5.3.1.2 Reuse deployment buffer

The second function of the Reuse Sentry is to provide a means of efficient deployment of compiler-directed computation reuse. The basis of the CCR approach is placement of a reuse instruction at the entrance to a large region of code that exhibits computation redundancy. The reuse instruction is formulated as a branch instruction with two potential locations for the next instruction to be executed: the fall thru or the taken location. The control resolution is based on whether previous computation results have been stored for the region.

In the dynamically managed approach, if a region has been classified as either *inactive*, *observed*, or *candidate*, then it has not been assigned computation reuse resources and it must execute each of its operations. Otherwise, if a region is *active* then resources have been assigned to it and there is a good chance that region execution can be bypassed by simply reusing previous computation results. In order to improve deployment of computation reuse, the Reuse Deployment Buffer (RDB) is constructed to predict reuse instructions similar to a *Branch Target Buffer* (BTB). For the *active* classification, the target address field in the RS entry of a region is used to predict the target of the reuse instruction as the next instruction following the entire computation region. If the target is not set, the RS predicts not-taken, and the execution continues with the next sequential instruction. The RS prediction is used to reduce the delay in determining whether the computation reuse region has been assigned resources.

5.3.2 Evaluation Buffer

After the Reuse Sentry identifies regions as candidates, they must be evaluated to determine reuse execution behavior, i.e. whether they warrant the use of computation reuse resources. There are two criteria that a candidate region must satisfy to be transitioned from the EB (candidate state) into the CRB (active state). First, the candidate must exhibit a high percentage of successful reuses, called an *active reuse interval*, occurring over a specified minimum amount of execution time, called the *active evaluation interval*. The minimum percentage of successful reuse over the time interval is called the *active reuse threshold*, while the actual reuse percentage over the interval is the *active reuse percentage*. The number of failed reuse attempts in the active reuse interval can also be tracked to determine whether an active interval exists. Second, the candidate region must have a minimum number of active intervals before the *evaluation interval*, a predefined number of all candidate region executions generated by incrementing the Evaluation Reset Counter for every candidate region execution, expires. The evaluation interval resets the entries within the evaluation buffer, allowing new regions to be evaluated. The number of active intervals exhibited by a region during the evaluation interval is called its *active interval count*.

The Evaluation Buffer hardware is shown in Figure 5.7. EB entries contain the following fields: active interval count, execution count, miss count, activity vector, and a computation instance array. The active interval count simply maintains the number of active intervals experienced during the evaluation interval and is used in assigning activation priority for the region. The execution counter tracks the number of executions in the currently evaluated active reuse interval. Miss count represents the number of reuse misses for the current active interval. The execution counter is implemented as a roll-over counter that is initialized to the minimum value. The counter increments for each candidate region execution, and an active interval is counted if the number of failed opportunities for reuse does not exceed the *failed threshold*.

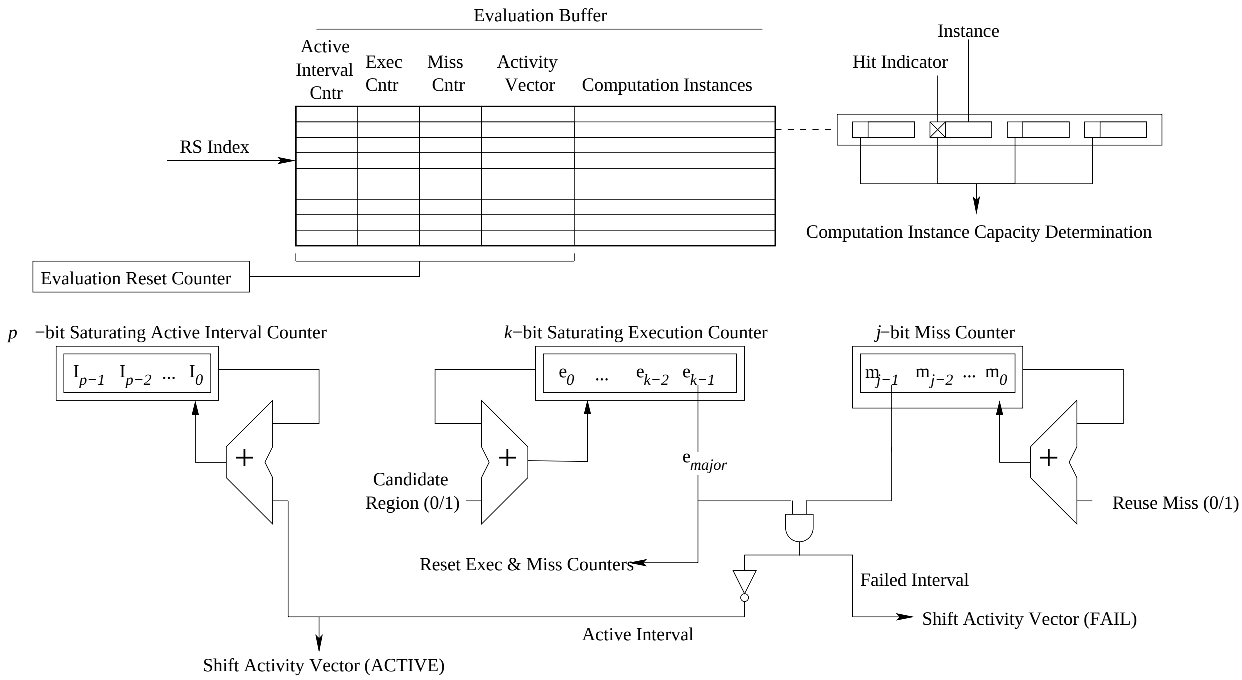


Figure 5.7 Evaluation Buffer hardware.

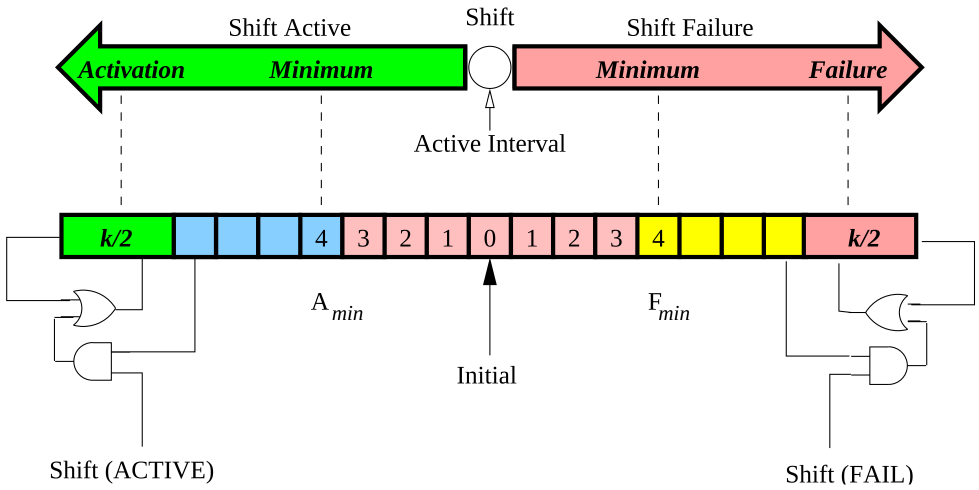


Figure 5.8 Activity vector.

The result of evaluating an active interval (active/failed) during every active evaluation interval is used to maintain an *active difference* in a special purpose shift register called the *activity vector*, shown in Figure 5.8. This vector is shifted in the active direction for every active interval achieved and shifted in the opposite direction for failed intervals. The vector maintains the approximate difference in the number of active intervals and failed intervals while aiding the evaluation of EB entries.

At the end of the evaluation interval, entries are removed from the EB under two conditions. First, the entry is removed if there is not an active interval count and a positive *active difference* has been recorded in the activity vector. This allows regions with only marginal reuse to be removed. Upon reaching the *maximum activation threshold*, attempts are made by the EB system to place the respective regions in the CRB. Similarly, at the end of the evaluation interval, region entries meeting a *minimum activation threshold* are compared with entries in the CRB for opportunities to revise CRB resource allocation. If no favorable opportunities exist, the EB entry is reset, but the region remains under evaluation (*continuation*). Second, the entry is immediately removed if the activity vector reaches the *maximum failed threshold*.

Entries with an activity vector indicating a *minimum failed threshold* are possibly replaced depending on the requests of the RS. At the end of the evaluation interval, entries with activity vectors indicating activity below the minimum active and minimum failed thresholds are also removed. Any of the above entry failures results in the region state being changed to *inactive*, while admittance to the CRB results in *active* state assignment.

The computation instance array of the EB entry differs in two ways from the traditional computation instance array. First, the input set only consists of input register operands and their respective values and the validation of memory state used by the computation. In order to minimize the dynamic management support hardware, the set of output register operands and their respective result values are not stored. Second, the array contains a single bit, called an *instance indicator*, for each computation instance activated during the evaluation process. The instance indicators serve to make an assessment of the number of computation instances necessary for the region. The EB migrates computation regions from the candidate to active state based upon availability of computation reuse buffer resources. The evaluation buffer determines the best candidate region by finding the EB entry with the largest difference between its number of active and failed intervals. Additionally, the number of set instance indicators is used to make the initial assignment to computation reuse entries. The computation reuse resources have a variable number of computation instances per computation entry, and the RMS informs the EB of available entries and their respective capacities. The EB decision is based first on matching the available capacity with the potential required capacity of entries and then on reuse evaluation (decided by the maximum activation difference).

5.3.3 Reuse Monitoring System

The Reuse Monitoring System determines if the computation resources designated for each active region are appropriate and revises the assignment of resources based on the observed run-time requirements. The computation reuse entries are divided into segments, each containing a different computation instance array capacity. Generally, the segmentation allows the RMS to match the hardware capacity with a region's requirements (adapting to irregular variation). The RMS performs four entry revisions: *demotion*, *promotion*, *eviction*, and *admittance*. The process of *demotion* transplants a region to an entry with lower computation instance capacity, while *promotion* gives a region higher computation instance capacity. The RMS is responsible for *evicting* regions from the computation reuse buffer if they do not result in successful reuse. Finally, a region be *admitted* from the EB if its reuse behavior is more favorable than existing CRB entries.

Figure 5.9 illustrates the proposed Reuse Monitoring System. Each computation entry in the CRB has a respective entry in the RMS that is used to govern the replacement policy of the entry and the replacement policy of the computation instances. The policies are able to share the same hardware components within the entries, since both operations are related to the run-time utilization and accuracy of the computation instances. To aid appropriate revision decisions, reuse behavior information is collected in the same manner as the EB entries. Figure 5.9 illustrates the additional fields for collecting activity information: activity history and computation instance activity. A *monitoring interval* is computed from the global *RMS Counter*, which is incremented for every active region execution. The monitoring interval periodically revises the existing region entries in the CRB and imports new region entries from the evaluation buffer.

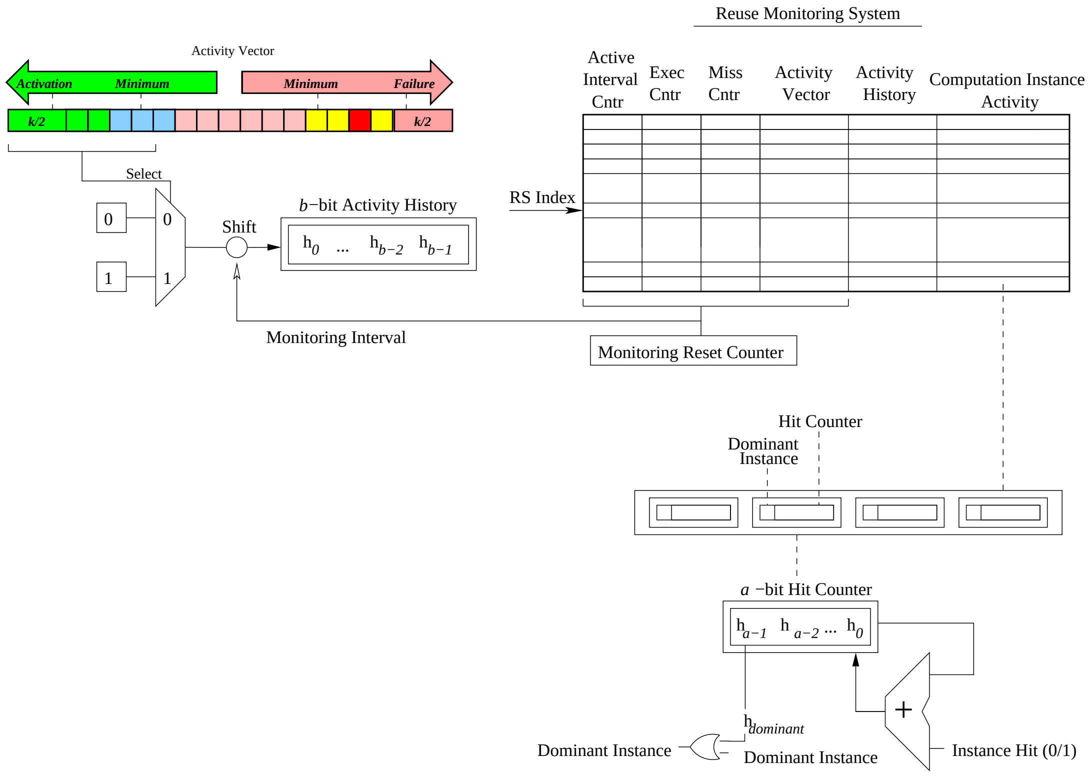


Figure 5.9 Reuse Monitoring System hardware.

Computation Instance Replacement. The RMS hardware provides high reuse accuracy for a region by tracking the activity of the computation instances within each computation entry. A traditional replacement policy for the instances, least recently used (LRU), does not perform well. However, there are some instances, called *dominant instances*, of computation that occur frequently over the lifetime of a region. The accuracy of region reuse can be greatly enhanced by allowing the dominant instances to remain in the CRB. To do this, *computation instance activity* (CIA) saturating counters are incremented for each successful reuse of an instance. When the counter saturates, the *dominant* field is set and remains set for the entire monitoring interval. On a computation instance miss, nondominant regions are selected using an LRU policy.

Computation Entry Revision. For the RMS to determine region requirements, the hardware must be aware of the reuse success rates of the computation placed in each computation entry. In addition, each entry maintains an *activity history* field that contains a b-bit shift register for recording the history of achieving the minimum activation percentage during each of the last b monitoring intervals. Bit position $history_0$ represents the current monitoring interval and position $history_{b-1}$ represents the activity level from $b-1$ monitoring intervals ago. In addition, the CIA counter information is used to estimate the needs of the computation region during revision of resource assignment.

Revisions are performed at the end of the monitoring interval and upon detection of an activity vector with excessive failures. The proposed implementation steps are followed in order of their appearance in Table 5.1. The timing column refers to revisions taking place on the reset of either the evaluation or monitoring intervals. The *segment_level* (SL) refers to the partitioned segment of the CRB; a higher segment level indicates segments with greater computation instances capacity. A *desirable candidate* (DC) refers to a region in the EB with a promising activation vector. When admitting regions from the evaluation buffer, the instance indicators are used to determine the base segment of the CRB to detect RMS entries. Admittance of a candidate region may preempt a region with a poor activity history.

Table 5.1 RMS region revision conditions.

Revision	Timing	Condition
Eviction	Asynchronous	MaxFailure Vector && !Segment_Entry_Available[SL+1→MAX]
	Synchronous	MinFailure Vector && DC && DC Vector > RMS Vector
Demotion	Synchronous	MinActive Vector && Dominant Instances < NumInstances[SL]/2
Promotion	Asynchronous	MaxFailure Vector && Segment_Available[SL+1→MAX]
	Synchronous	MinFailure Vector && Segment_Available[SL+1→MAX]
Admittance	Synchronous	DC MaxActive Vector && Segment_Available[Dominant Level→MAX]
	Asynchronous	DC MinActive Vector && Segment_Available[Dominant Level→MAX]
	Synchronous	DC MaxActive Vector && DC Vector > RMS Vector && Population(RMS History) < 1/2 History Size && Population (recent RMS History) < 1/2 Population (RMS History)

5.4 Experimental Evaluation

5.4.1 Methodology

The IMPACT compiler and emulation-driven simulator were enhanced to support a model of the proposed architecture framework and the region formation techniques respectively introduced in Section 5.3 and Chapter 3. Concerning the general compilation environment and processor model evaluated, all experiments of this section were generated using the same methods outlined in Section 4.5.

In the experimental simulations of the DCMS, the computation buffer had 32 entries with 16 computation instances (CIs) per entry. Each CI supports an input and output 8-entry register array. The DCMS was configured according to the hardware assignments and parameters listed in Table 5.2. Because the design space is complex, experimentally evaluating the individual effect of each hardware parameter was infeasible. Initial parameters were selected and optimal settings were selected based on the hardware cost and approximate performance constraints. The Reuse Sentry hardware is configured to allow regions with a dynamic execution percentage ranging from 6% (32 executions/reset interval) to 3% (16 executions/512 branches) to become candidates. The EB hardware is configured to activate regions that have more than 66% (2:1) active intervals and determine an active interval by reusing results greater than half (active threshold percentage) the executions of an interval of its 32 region executions. The RMS parameters are established identically to respective structure components in the EB, except that they are given slightly more cautious failed activity vectors to more readily adapt to changes in reuse requirements. CRB revisions required 12 cycles, representing the case when a large computation entry was migrated to an entry with reduced capacity.

Table 5.2 DCMS hardware configuration.

System	Component	Setting
Reuse Sentry (RS)	Number entries	256
	Associativity	2-way
	Exec counter size	5 bits
	Minor Candidate region threshold	16
	Major Candidate region threshold	32
	Observation reset interval	512 observed region executions
Evaluation Buffer (EB)	Number entries	8
	Associativity	Fully associative-Directly assigned
	Exec and miss counter size	5 bits
	Active interval counter	4 bits
	Activity vector size	15 bits
	MinActive vector position	4th active position
	MaxActive vector position	8th active position
	MinFailure vector position	4th failed position
	MaxFailure vector position	6th failed position
	Evaluation reset interval	1024 candidate region executions
Reuse Monitor System (RMS)	Number entries	Size of CRB
	Associativity	Fully associative-Directly assigned
	Monitoring reset interval	2048 active region executions
	Exec and miss counter size	5 bits
	Active interval counter	4 bits
	Activity vector size	13 bits
	MinActive vector position	4th active position
	MaxActive vector position	8th active position
	MinFailure vector position	2th failed position
	MaxFailure vector position	4th failed position
	Active history	8 bits
	Dominant Instance Hit Counter	5 bits
Computation Reuse Buffer (CRB)	Number entries	32
	Number computation entry segments	4
	Segment types	2,4,8,16 computation instances

5.4.2 Experimental results

Computation management results are presented here: (1) overall performance and reuse accuracy, (2) region formation characteristics, and (3) hardware costs.

Performance and Accuracy. The overall cycle-time speedups for evaluating the reference input set in the CRB and DCMS approaches are presented in Figure 5.10. The training input set is used in guiding computation region formation. The CRB design is evaluated with a direct-mapped and a two-way set associative configuration. The DCMS approach models are evaluated with a 32-entry CRB with four equal segments of computation instances respectively holding 2, 4, 8, and 16 computation instances. The two DCMS models are executed with different profile-guided reuse thresholds: 65% and 55% (rightmost bar).

Overall, Figure 5.10 illustrates the value of the DCMS approach for two reasons. First, the DCMS approach achieves higher performance due to better computation management properties than the set-associative CRB approach. Essentially the components distributed in the RS, EB, and RMS are collecting run-time information that can improve the ability to manage the CRB better than simply providing more available resources to the CRB. On average the performance speedup degrades to 15% if the regions formed with reuse threshold 55% are executed on the set-associative CRB. Second, a processor implemented for CCR with DCMS has enhanced speedups due to enabling the reuse in regions not selected for the base CRB modes. The DCMS model with the additional regions selected by lowering reuse identification to 55% is able to provide the DCMS with more opportunities to exploit dynamic computation redundancy.

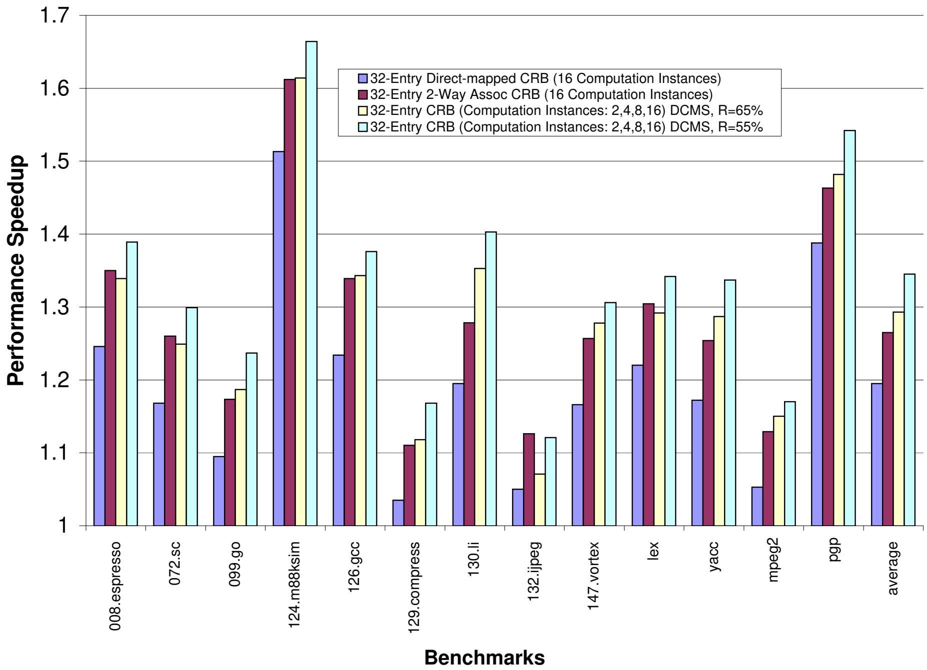


Figure 5.10 Performance for CRB and DCMS models.

The performance benefits of the DCMS approach are enabled by reducing the percentage of failed reuse attempts and increasing the percentage of successful reuse attempts. Figure 5.11 presents the distribution of reuse attempts for the DCMS relative to the base CRB approach for the lower reuse identification threshold. On average there is both a reduction in reuse failures (22%) and an improvement in successful reuse attempts (21%). The RS and EB stages of the DCMS are working in a coordinated fashion to make only confident reuse attempts. The addition of computation regions using the DCMS approach increases the percentage of failed reuse attempts for *124.m88ksim*. However, overall reuse accuracy exceeded the marginal number of failures added. Although some of the DCMS parameters did not allow regions with long intervals between executions to use the CRB resources, the DCMS had an overall positive effect on performance. Nevertheless, there is a great deal of potential in improving the management in the DCMS to use information other than reuse and execution frequency.

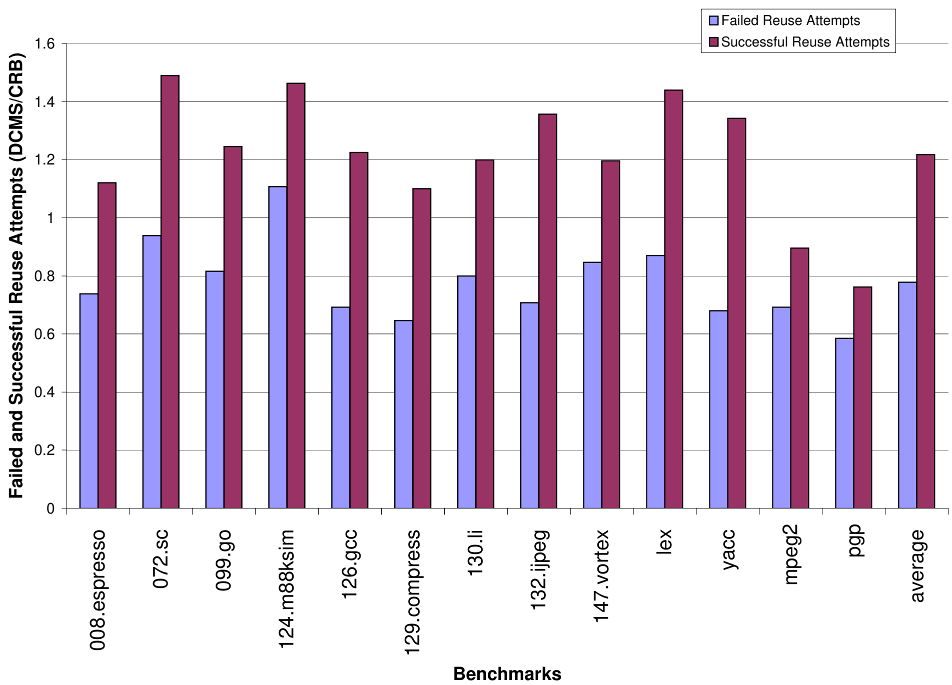


Figure 5.11 Comparison of reuse attempts for the CRB and DCMS models.

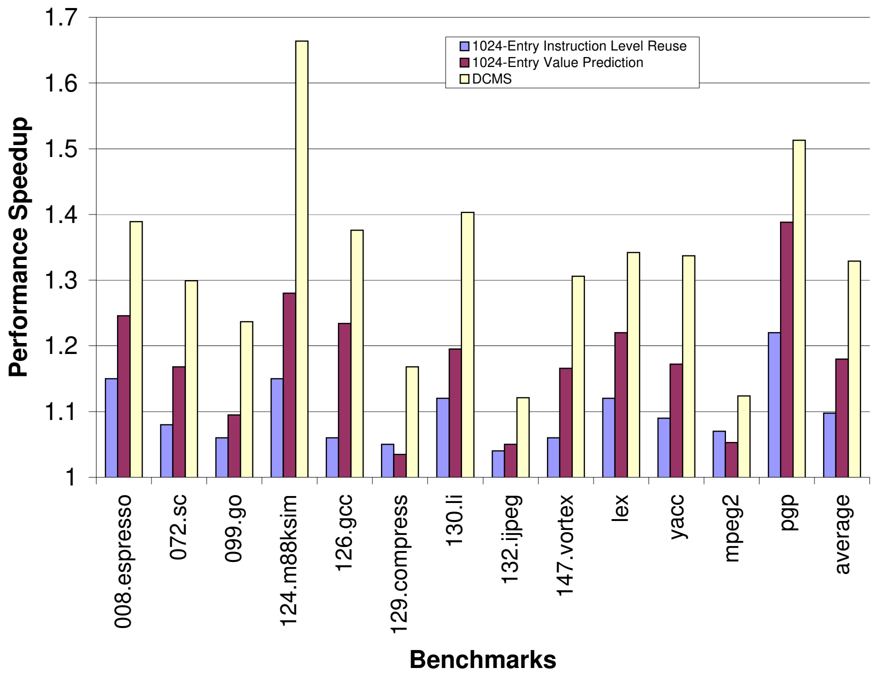


Figure 5.12 Performance for DCMS, instruction reuse (IR), and value prediction (VP).

Figure 5.12 illustrates the performance for three locality exploitation schemes: DCMS, instruction reuse (IR), and instruction-level value prediction (VP). The instruction-level techniques are provided with 1024 entries of their respective hardware resource. The IR method has a history size of eight entries. The results indicate that the dynamically managed compiler-directed scheme can achieve higher performance for all of the benchmarks studied.

Region Formation Evaluation. Although the techniques of Section 3.4 represent a first attempt at statically identifying reuse in programs for the DCMS approach, results indicate promising potential in eliminating use of value-invariant profile information in the CCR approach. Figure 5.13 illustrates the performance speedup achieved using the inferred and structured approaches relative to the profile-guided approach. A third experimental method combines the maximum number of regions found using both static approaches. On average, the individual static approaches are able to achieve 40-43% of the profile-guided speedup.

Figure 5.13 shows that the structured approach has substantially better performance for benchmarks *129.compress*, *mpeg2*, and *pgp*. These programs exploit value-invariant behavior directly related to data input at the application programming level, allowing no value-invariant inferences to be made, and thus the respective computation regions were more readily found using the structure technique. Generally such programs have natural code regions that represent fundamental tasks of inherent program algorithm. On the other hand, the inferred approach is able to better identify regions in benchmarks *124.m88ksim* and *126.gcc* which have abundant analyzable relational information found by examining individual modules in a whole-program setting. The combined approach significantly improves the percentage of profile-guided speedup achieved to nearly 55%, indicating that the best region identification scheme will most probably be composed of multiple methods.

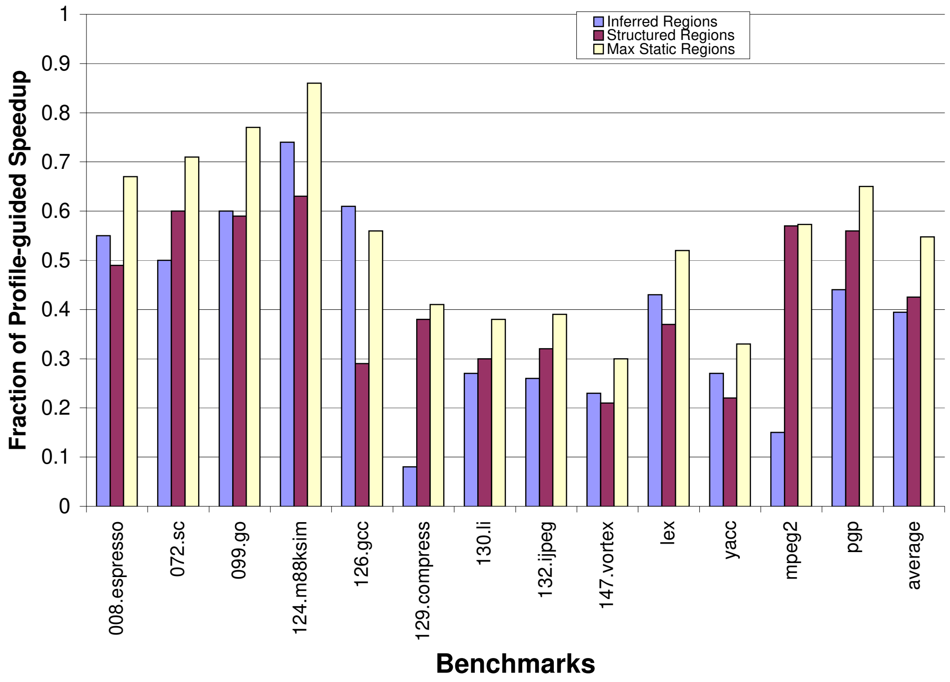


Figure 5.13 Performance for static region identification techniques.

The final important aspect of the DCMS approach investigated was the evaluation of the region formation techniques on a new untrained input. Figure 5.14 illustrates three methods of region identification: profile-guided of the training and reference inputs, the structured and inferred static approaches, and the profile-guided and static approaches. All methods generate the maximum number of regions corresponding to the multiple inputs or analysis techniques. The experiment evaluates a subset of the original benchmarks that have a third available input set. The results indicate the continued success of both the profile-guided and static region identification approaches, each respectively averaging 28% and almost 20%. The combined method employs regions identified using either method, and on average has 5-6% greater performance improvement than the profile-guided method. This result indicates that the profile-guided inputs are not indicating all of the potential program reuse behavior. Nevertheless, the DCMS improves performance by selecting the statically identified regions at run time.

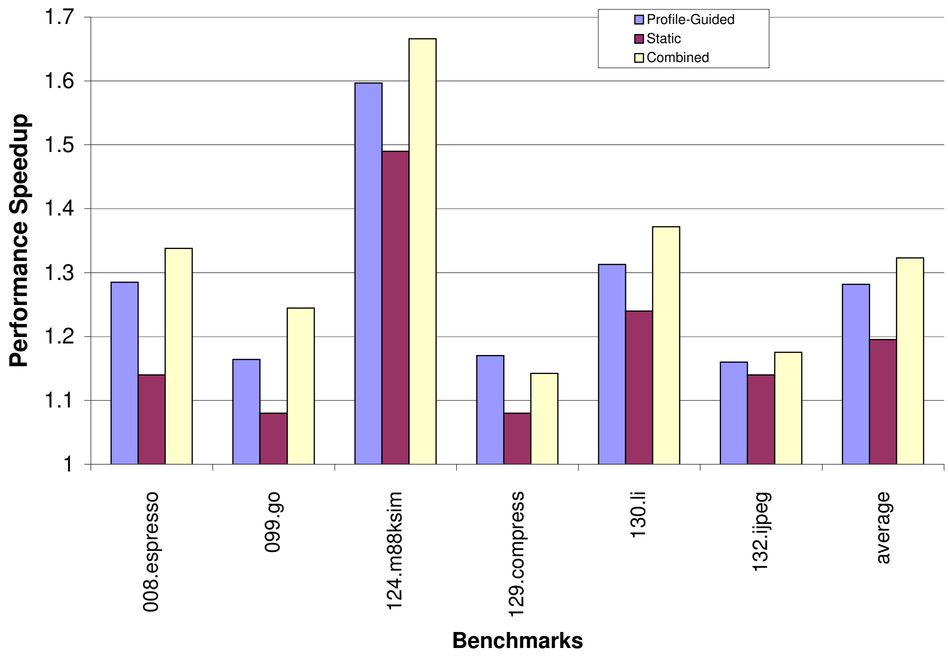


Figure 5.14 Performance for combining profile-guided and static region identification methods on untrained input set.

Table 5.3 Hardware cost expressions and cost for models evaluated.

Scheme	Component	Cost Expression (in bytes)	Cost
CRB	-	$Num\_Entries * (CI\_entry\_cost(Num\_Instances))$	42624
DCMS	CRB	$Num\_Entries\_Segment[0] * (CI\_entry\_cost(Num\_Instances\_Segment[0]))$ $+ ..... + Num\_Entries\_Segment[Segments - 1] *$ $(CI\_entry\_cost(Num\_Instances\_Segment[Segments - 1]))$	19980
	RS	$Num\_Entries * (Tag + Exec_{cntr} + State + Target_{addr} + CRB\_index)$	2240
	EB	$Num\_Entries * (Interval_{cntr} + Exec_{cntr} + Miss_{cntr}$ $+ Activity_{vector} + Instance\_Indicators(Num\_Instances))$	5276
	RMS	$Num\_Entries * (Interval_{cntr} + Exec_{cntr} + Miss_{cntr} + Activity_{vector}$ $+ Activity_{history} + Dominant\_Instance_{cntr} * Num\_Instances\_in\_CRB\_entry)$	316
	(total)	-	27812

Hardware Implementation Costs. Overall, the moderate performance speedups reinforce the strategy of using dynamic management support since the hardware requirements are significantly less than the costs of the base CRB model. Table 5.3 represents a hardware cost estimate of the CRB and DCMS models evaluated. This table accounts for bytes of hardware memory, but does not include wiring or logic gate costs. Table 5.2 defines the symbols and the respective settings found in Table 5.3. Using this examination, the DCMS requires nearly half the hardware of the traditional CRB, yet provides improved region management functionality and the ability to adapt to run-time program behaviors. Experimental results indicate that the EB hardware resources have the most influence on achieving the reported performance results.

Figure 5.15 illustrates the performance results comparing varying levels of hardware resources for the three locality schemes. Two benchmarks, *008.espresso* and *129.compress*, along with the average of all benchmarks are presented. The results indicate that the dynamically managed compiler-directed scheme can achieve higher performance even when substantial hardware resources are available for instruction-level reuse and value prediction. Only when the resources are limited to 16K are the performance results equivalent.

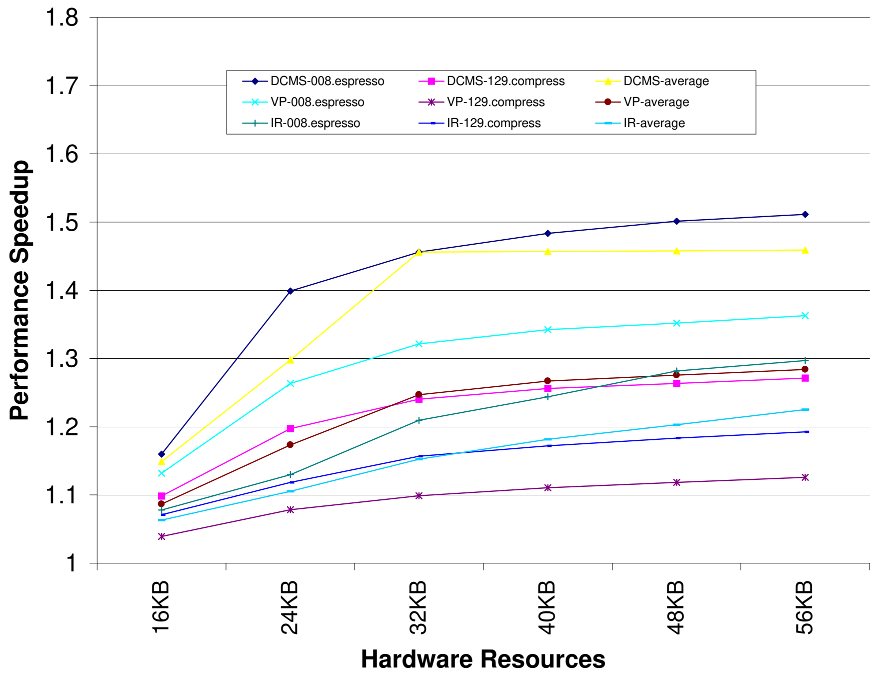


Figure 5.15 Performance when varying levels of hardware resources for DCMS, instruction reuse (IR), and value prediction (VP).

5.5 Summary

Innovations in high-performance system design and the availability of silicon resources have allowed modern processors to include mechanisms for analyzing run-time program behavior to effectively manage constrained resources. The proposed system in this chapter enhances compiler-directed computation reuse by allowing the processor to detect the most frequently executed code regions, to examine reuse execution behavior, to dynamically allocate reuse buffer resources, and to selectively deploy code regions for optimal elimination of dynamic computation redundancy. To evaluate the opportunities created by the mechanism presented in this chapter, advanced compiler techniques were employed to provide an abundant supply of potential computation regions. Results show that with little additional hardware, this combination of new region identification techniques and a dynamic management system achieves performance improvement over the traditional computation reuse framework. In addition, the chapter provides insight into opportunities for static identification of computation reuse in programs, which would eliminate dependence on value-invariance profile information in the software development process.

CHAPTER 6

VALUE OPTIMIZATION FRAMEWORK

One fundamental limitation of traditional compiler optimization techniques is that they do not significantly alter the program's original control and data flow structure to achieve high performance on wide-issue processors. By exploiting persistent value locality characteristics within programs, an enabled compiler can enhance the original computation to include new optimized control paths with value-specific contexts. Specialized code can be synthesized by analyzing the data distribution of the region. The region identification approaches used for compiler-directed computation reuse to determine redundant code regions can be used to find such opportunities for applying value-based compiler transformations.

This chapter presents an effective optimization framework that allows the compiler to transform programs to exploit value locality behavior without new hardware support. Value profiling and value analysis techniques are used to extend the data dependence representation of the compiler and are used as the basis for transforming the code in the Value Optimization Framework (VOF). Predicated execution and predicate-based optimization techniques play an imperative role in the optimization framework by allowing the newly created value-context paths to be optimally executed on modern processors designed to exploit instruction-level parallelism.

6.1 Introduction

One fundamental limitation of most traditional compiler optimization techniques is that they do not significantly alter the program's original control and data flow structure. For processors designed to exploit instruction-level parallelism (ILP) to be effective, compilers must aggressively restructure code to express program behavior characteristics and utilize micro-architecture techniques. As the compiler translates high-level language constructs into assembly instructions, it does not alter the basic control structure or synthesize instructions based on the value-locality of the code's execution behavior. Instead, most optimization techniques focus on exposing and increasing ILP within a fixed control structure.

Restricting a compiler to use the program's unaltered control and data flow structure is undesirable for several reasons. The deficiencies in executing the original control flow presented within programs have been previously studied [54]. First, a high-level language represents program control flow in an extremely sequential manner through the use of nested if-then-else statements, switch statements, and loop constructs. Each control construct is fully evaluated before proceeding to the next. This sequential computation often defines the program critical paths that constrain the available ILP. Second, programmers represent control flow for understandability or for ease of debugging rather than for efficient execution on the target architecture. As a result, software often contains redundant control constructs that are difficult to detect with traditional compiler techniques. These may involve evaluating the same conditions multiple times or evaluating conditions that partially overlap. For such reasons, an effective ILP compiler should be capable of transforming the program control structure to eliminate these problems.

The original dataflow structure of a program also presents similar limitations to the performance of modern processors. The inherent structure as described in the source within a program becomes the major factor in determining the efficiency of wide-issue processors since the existence of multiple independence instructions is a property of the program algorithm and not of the design technology. In order to enable their cost-effective performance potential, these processors demand that increasing levels of ILP be exposed in programs. The ability to restructure code aggressively is a critical feature of an effective optimizing compiler. The most obvious situation where aggressive transformation is regularly applied is on arithmetic expressions. Compilers often completely restructure the programmer's arithmetic computations into more parallel forms using a variety of transformations. These include expression re-association, tree height reduction [55], and blocked back substitution [56]. Although ILP compilers may aggressively restructure computation, they typically apply only a limited set of arithmetic identities, theorems, and rules to the program's original dataflow structure. Thus any reformulation approach that operates by exploiting natural characteristics of a program has the potential to dramatically enhance the level of efficiency as well as the level of ILP achieved in programs. Only when the limit of the ILP is completely determined and the fundamental dataflow limitation imposed by data dependences remains can the ILP paradigm be fairly evaluated.

Recent studies on value locality reveal that many code sequences frequently execute with a small variety of inputs [5], [9]. Value locality, the repetition of a set of values executed by an instruction, is one program characteristic with substantial potential for enabling new methods of program reformulation. The existence of a limited set of value characteristics for a code segment essentially represent an opportunity to develop new value-specific transformations that surpass the limitations imposed by the data dependences that remain after all fundamental arithmetic computations have been applied. To exploit dynamic redundancy of value sets in programs, most proposed models are divided into two hardware strategies, speculative value prediction [16] and dynamic instruction reuse [17]. However, many hardware complexity and system design limitations exist in the detection of value locality within a program at run time. A more aggressive alternative, the compiler-directed reuse approach described in the previous chapters of this thesis and also published in [38], is to allow the compiler to partition the program into potentially reusable regions of computation whose results are then dynamically recorded in hardware for future reuse. Many code regions exhibiting value locality and identified through the value profiling and analysis techniques of compiler-directed reuse approach can be reformulated to use existing instruction execution capabilities rather than require dedicated hardware buffers. These value-based reformulation opportunities can be expressed to existing microarchitectures that support predicated execution, eliminating the need for special hardware.

Predication has become an effective instruction set architecture feature for expressing program control by conditionally executing instructions [57], [58]. A compiler can employ if-conversion to convert a sequence of code containing branches into an equivalent branch-free sequence of conditionally executed instructions [59]. Predicated execution increases ILP by allowing the compiler to schedule operations from multiple paths of control for simultaneous execution. The relation between the predicated instruction representation and value-based transformations is that predication support within the microarchitecture allows multiple paths to be executed simultaneously. These multiple paths, although traditionally representing the control flow of a program, can also be the reformulated control paths synthesized from dataflow sequences of a program. By utilizing predication support for dataflow sequences with value locality, new opportunities for applying predication support within programs are created. At the same time, value-based reformulation improves the performance efficiency of the compiler-directed reuse approach since regions with low variation can be reformulated rather than directed to the limited computation reuse buffer hardware resources.

Value-Based Optimization Approach. Motivated by the potential of aggressive techniques for transforming arithmetic expressions, this chapter introduces a new approach to optimizing program dataflow structures. The goal of this work is to develop a systematic methodology for reformulating specific regions of program data flow for more efficient exploitation of their inherent value locality characteristics on an ILP processor. Dynamic value locality information is first gathered and represented as a *value dependence graph*. A new, more parallel computation is synthesized with the goal of reducing dependence height. To accomplish the desired optimization and synthesis, the parallel computation is modeled as new control flow structures. In turn the control structures are converted to a predicate representation using if-conversion techniques and predicated-based optimization techniques are then employed.

VOF explores applying two advanced predicate reformulation techniques, Predicate Decision Logic Optimization (PDLO) [54] and Partial Reverse If-Conversion (PRIC) [60], to the value-synthesized code region. Using PDLO, the control operations of the region are first systematically extracted and represented as a *program decision logic network*. Then, a more efficient control network is re-synthesized with the goals of reducing dependence height and redundancy. To accomplish the desired optimization and synthesis, the program decision logic network is modeled as a Boolean equation. Boolean minimization techniques are then applied to simplify and optimize the equation. Finally, the optimized network is re-expressed in the form of predicated assembly code. After predicate-based optimization, the PRIC framework is enabled to extract the full benefits of applying value-based transformations using the predicate representation. The *partial reverse if-conversion* technique operates at schedule time to balance the amount of predicated parallel computation code with any untransformed components of the original code. The balance is based on the characteristics of the target processor.

Figure 6.1 illustrates the program transformation paths and compiler techniques presented in this section. There are three potential program representation paths: traditional control flow (CF), predicate dataflow (PF), and value-inserted control flow (VF). The predicate representation of code effectively designates the execution conditions of the newly created program sequences. The key idea behind the value-based optimization framework is to synthesize optimized code structures by analyzing the original program value characteristics and applying a series of control flow and data flow transformations using both a predicate-based compiler representation and predicated execution microarchitecture support. Experimental results indicate that programmatic logic optimization and predicate-based transformation can result in substantial performance improvements in program regions where value locality is persistent.

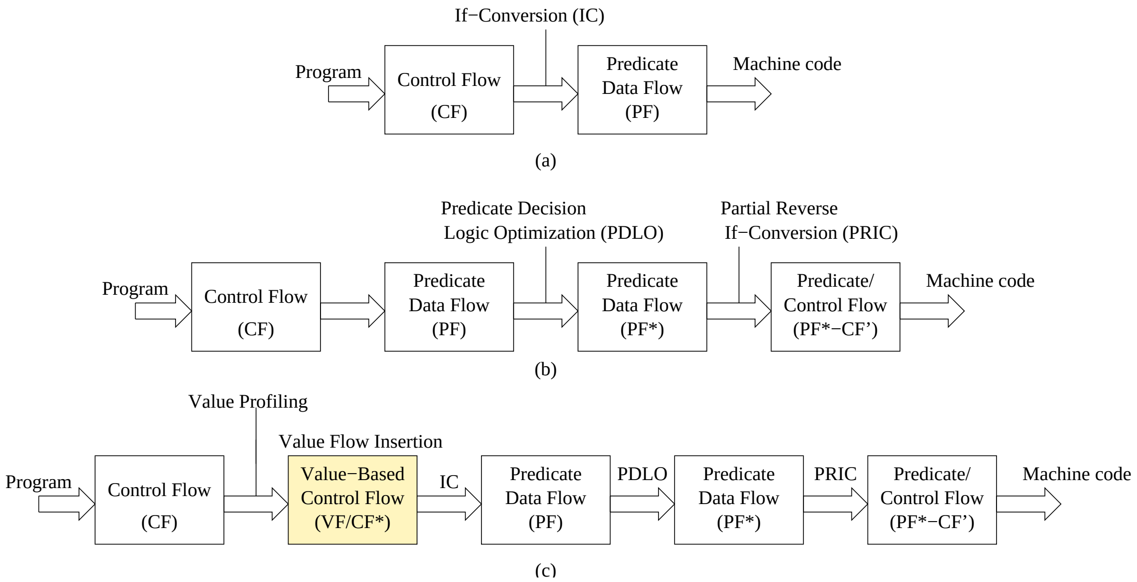


Figure 6.1 Program flow-graph transformation paths.

The remainder of this chapter is organized as follows. Section 6.2 provides a brief overview of related methods of exploiting value locality and an overview of predicated execution support that provides the underlying microarchitecture mechanism for the proposed optimization framework. Next, Section 6.3 presents the compiler optimization opportunities that motivate reformulating regions of code to predicate-based sequences that can effectively exploit the value locality characteristics. Section 6.4 details the compilation framework for establishing the value-based optimizations. The effectiveness of the proposed approach in improving performance and exploiting value-locality behavior is presented in Section 6.5. Finally, the chapter is summarized in Section 6.6.

6.2 Related Work and Architecture Support

Related Work. Previous research in the area of value locality and redundancy exploitation can be classified into two major categories, value prediction and dynamic instruction reuse. Value prediction and dynamic instruction reuse are two important strategies that attempt to reduce the execution time of programs by alleviating the dataflow constraints using hardware. Although alternative schemes use profiling information to guide the detection mechanism [18], the performance improvement of these proposed approaches is often limited by the ability of the hardware mechanism to effectively transform the observed value locality into a form that can reduce the processor execution time. However, compiler techniques offer a wide range of transformations of code that have yet to be fully applied to exploit value locality behaviors in codes.

In addition to the arithmetic reformulation work mentioned in the previous section, a large body of related work in the area of control flow optimization has also been carried out. These methods can be classified into three major categories: branch elimination, branch reordering, and control height reduction. Branch elimination techniques identify and remove those branches whose direction is known at compile time. The simplest form of branch elimination is loop unrolling, in which instances of back-edge branches are removed by replicating the body of the loop. More sophisticated techniques examine program control flow and data flow simultaneously to identify correlations among branches [61], [62]. When a correlation is detected, a branch direction is determinable by the compiler along one or more paths, and the branch can be eliminated. In [62], an algorithm is developed to identify correlations and to perform the necessary code replication to remove branches within a local scope. This approach is generalized and extended to the program-level scope in [61]. The second category of control flow optimization work is branch reordering. In this work, the order in which branches are evaluated is changed to reduce the average depth traversed through a network of branches [63].

Another category of program reformulation is control flow optimization research, which focuses on the reduction of control dependence height. This work attempts to collapse the sequential evaluation of linear chains of branches in order to reduce the height of program critical paths [64]. In an approach analogous to a carry lookahead adder, a lookahead branch is used to calculate the taken condition of a series of branches in a parallel form. Subsequent operations dependent on any of the branches in the series need only to wait for the lookahead branch to complete. The control dependence height of the branch series is thus reduced to that of a single branch. The mechanisms introduced herein also serve to reduce control dependence height. This chapter, however, leverages an approach [54] to minimization and re-expression of control flow networks that is far more general than those proposed in previous work.

The final category of related work is value-based compiler transformations. Value-based optimizations include all optimizations based on a predictable value or range of values for a variable or instruction at run time. These include constant propagation, code specialization [65], optimizations assuming the value predictability of an instruction, continuous optimization, and partial evaluation. Variables and instructions that have invariant or predictable values at run time, but cannot be identified as such using compiler analysis, can benefit from value-based compiler optimizations. Dynamic compilation and run-time optimization are emerging directions for computer system research which provide improved execution performance by performing some transformations that exploit variable invariance behavior. Dynamic-compilation process [24], [26] segments the stages of compilation to both traditional compile time and run time by using a form of binding time analysis to generate templates for code sequences that have been identified as semi-invariant. *Dyc* [23] is a dynamic compilation system in which the programmer annotates regions of the programs that should be compiled dynamically and variables for which to specialize the regions. A static optimizing compiler automatically identifies which data will have known values at run time, given the values of the annotated variables, then creates and optimizes machine-code templates.

Architecture Support. Predicated execution, the main architectural feature utilized in this work, is a mechanism that facilitates the conditional execution of individual instructions [58]. Predicates are registers that store a single bit value, representing either TRUE or FALSE. Each instruction is associated with a particular predicate, known as its guard predicate, that determines its execution. In the case when an instruction's guard predicate is TRUE, it executes normally. Conversely, when an instruction's guard predicate is FALSE, it is nullified.

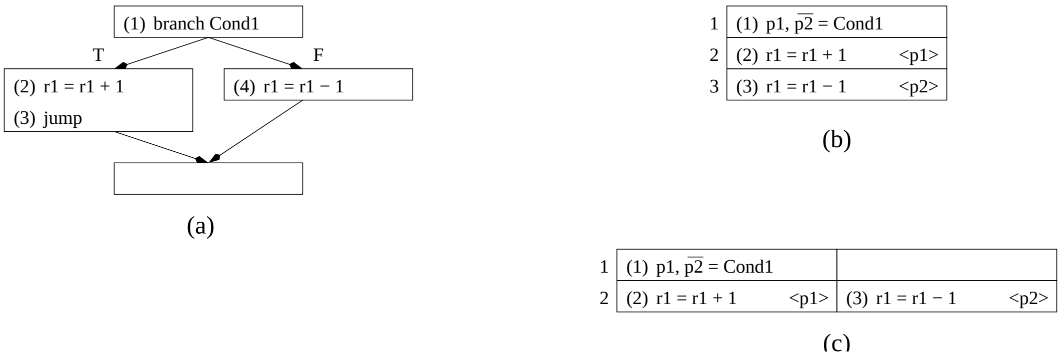


Figure 6.2 An example code segment (a), after if-conversion (b), and after scheduling (c).

Figure 6.2 contains a simple example illustrating the concept of predicated execution. Figure 6.2(a) shows sequential if-then-else constructs, called *hammocks*. The branch outcomes are determined by the evaluation of the branch condition $Cond1$. Depending on the outcome of the first branch register, $r1$ is either incremented or decremented. Figure 6.2(b) shows the code segment after if-conversion, and Figure 6.2(c) shows the code after scheduling. Here the two branch conditions have been transformed into comparison instructions that define predicate destinations. The example illustrates how support for predicated execution allows the multiple path contexts to be executed in parallel on a wide-issue machine.

The most important component of a predicate architecture is the instruction set support for computing predicates, or the *predicate define instructions*. Predicate defines are inserted by the compiler to generate values for control of conditional execution. The PlayDoh predicate define instruction set [66] provides a baseline predicated execution support. An additional strategy for enhancing the generation of predicated code identifies new Boolean optimization predicate define extensions to the PlayDoh instruction set [54]. For each destination predicate register, a predicate define instruction can either deposit a 1, deposit a 0, or leave the contents unchanged. The predicate type specifies a function of the source predicate and the result of the comparison that is applied to derive the resultant predicate.

Table 6.1 Predicate definition truth table.

		Base compare types						Predicate optimization types			
pSRC	Comp	UT	UF	OT	OF	AT	AF	$\vee$T	$\vee$F	$\wedge$T	$\wedge$F
0	0	0	0	-	-	-	-	-	1	0	0
0	1	0	0	-	-	-	-	1	-	0	0
1	0	0	1	-	1	0	-	1	1	0	-
1	1	1	0	1	-	-	0	1	1	-	0

Table 6.1 (left-hand portion) shows the deposit rules for each of the PlayDoh predicate types in both normal and complement modes. PlayDoh defines three predicate types, *unconditional* (UT or UF), *wired-or* (OT or OF), and *wired-and* (AT or AF). Each type can be in either normal mode or complement mode, as distinguished by the T or F appended to the type specifier (U, O, or A). Complement mode differs from normal mode only in that the condition evaluation is treated in the opposite logical sense. Each entry corresponds to the result assigned to the destination predicate. Note that a "-" means that the destination is left unchanged.

The Boolean predicate types introduced to facilitate generating efficient predicate decision logic are referred to as *disjunctive-type* ($\vee$T or $\vee$F) and *conjunctive-type* ($\wedge$T or $\wedge$F). Table 6.1 (right-hand portion) shows the deposit rules for the new predicate types. The $\wedge$T-type define clears the destination predicate to 0 if either the source predicate is FALSE or the comparison result is FALSE. Otherwise, the destination is left unchanged. Note that this behavior differs from that of the and-type predicate define, in that the and-type define leaves the destination unaltered when the source predicate evaluates to FALSE. The conjunctive-type thus enables the compiler easily and efficiently to form the logical conjunction of an arbitrary set of conditions and predicates. The disjunctive-type behavior is analogous to that of the conjunctive-type.

$$pD_0\_type_0,\ pD_1\_type_1 = (src_0\ \mathbf{cond}\ \mathbf{src_1})\ \langle pSRC \rangle.$$

The instruction is interpreted as follows: pD_0 and pD_1 are the destination predicate registers; $type_0$ and $type_1$ are the predicate types of each destination; src_0 **cond** src_1 is the comparison, where **cond** can be *equal (==)*, *not equal (! =)*, *greater than (>)*, etc.; and $pSRC$ is the source predicate register. The value assigned to each destination is dependent on the predicate type.

6.3 Value-Based Compiler Reformulation Motivation

This section presents an overview of the value-based program optimization, starting with the conversion of dataflow sequences to the control representation for subsequent predicate-based conversion and reformulation. In order to simplify the expression of code paths with exploitable value locality characteristics, the compiler applies if-conversion to transform all potential data flow into the predicated representation. In the IMPACT compiler, this conversion is traditionally performed within code regions formed using *hyperblock* formation heuristics [67] which determine complex control block structures with frequently executed but unbiased dynamic paths. To a great extent, the designation of new control structure relates to the existing hyperblock structure, as codes with a high level of control parallelism are often the lead selection criteria for forming hyperblocks. In the case of new value-based codes, the number of different value behavior contexts reformulated corresponds to the level of control parallelism.

The compilation techniques for detecting the code regions executing with value locality characteristics are based on the concept of *deterministic* computation regions. A deterministic computation region is an arbitrary, connected subgraph of the program control flow graph that can be analyzed to determine the location of all input operands and memory locations that affect the region's computation. The nature of detecting a repeatable sequence requires some estimation of the run-time execution behavior of a program. The most direct method is value profiling [2]. Value profiling is an effective method of finding the value recurrence and potential reuse of instructions. Applying dynamic information with formal analysis [4] can also help in finding relationships among computations and understanding the fundamental source of program redundancy and predictability. Studies [3] have supported the existence of regions of code with strong value locality that persist over multiple program input evaluations. To illustrating the opportunities of program reformulation on these established characteristics, three concepts are explored in this section. First, a base example of value-based dataflow conversion for a single operand is displayed, illustrates the initial potential of using predicated-execution support to synthesize new code from program value behaviors. Next, the inherent memory flow of a program is transformed using the outlined techniques to eliminate redundancy in program execution. Finally, value-based dataflow transformation is shown for a case with multiple operands, demonstrating the power of applying predicate optimization and reformulation techniques to code originally containing only computation instructions.

6.3.1 Dataflow transformation

The process of converting the dataflow representation of a program into the context-based predicate dataflow representation as described in Figure 6.1(c) is motivated and explained through a simple example. Figure 6.3 illustrates a three-step process being applied to a simple straight-line sequence of two instructions. The first step, shown in Figure 6.3(a), illustrates the starting code. For the purpose of this example, value analysis may have yielded results indicating that the value of variable x may occur with only two possible values, 1 or 2. By applying the *Value Control Insertion (VCI)* transform, the dataflow of the sequence is identically represented by the code shown in Figure 6.3(b). The two values are used in formulating a branch instruction, and the two possible paths include the context of information supplies by the transform.[1] By applying a series of traditional compiler optimizations to each value path, more efficient sequences can be created. The third step applies *if-conversion* to the value paths, removing the intermediate control representation and replacing the control structure with predicate dataflow structure, called the *value sequence*. The final code for the transform is shown in Figure 6.3(c). The value sequence is composed of two components, the *decision component* and the *residue output computation component*. The predicate definition instructions synthesized by the intermediate control flow are referred to as the decision component of the new dataflow computation. Similarly the predicated instructions, in this case move instructions, are referred to as the residue output computation component, since the purpose of the instruction is to move the results into the respective live-out registers of the original code sequence.

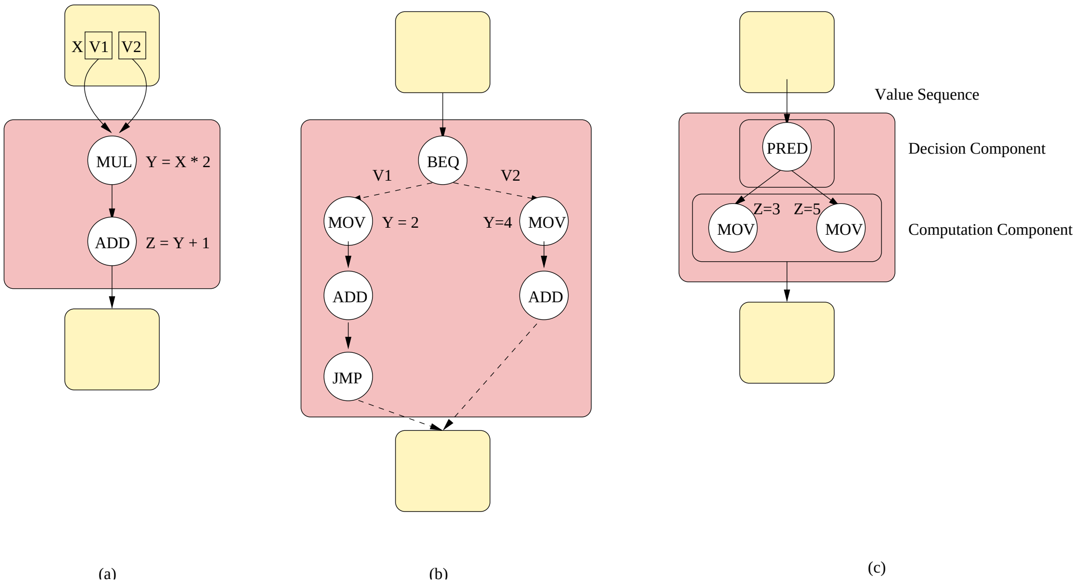


Figure 6.3 Code for one variable *closed* region: dataflow representation (a), value-transformed control flow representation (b), and value-transformed dataflow representation (c).

[1]Namely that value of X is 1 for the left-hand path, and the value of X is 2 for the right hand path.

There are several advantages to the code represented by the value-based predicate form. First, the long latency multiply instruction is strength reduced, allowing dependent instructions to execute early. The parallel code created is likely to execute efficiently on a machine model design to execute multiple instructions per cycle. Generally, the advantages are that the optimizations may yield shorter code sequences at very little additional code size. The dataflow transformation supplied in this first example is for a single variable dependence chain region, and is said to be of *degree-1*. Similarly, the example's variation is referred to as *order-2 closed* since in the example only two values are possible for the case. Several variations of the final code can be caused by the form of the original code and the value information used. Figure 6.4 illustrates an important aspect of the value transform for the case of a *degree-1, order-3 open* region. Figure 6.4(a) shows the original code with the description of the sequence order. Since the order is open, an unknown value may appear for the value of X that is not 1, 2, or 3. The order in this example may have been generated by profile information. In fact the profile information can simply summarize a variable as open to include multiple values that have very infrequent weight to create very efficient code for the value scenarios that occur often.

Figure 6.4(b) shows the form of the code after the VCI transform in which four value paths correspond to the ordered-values and open value. Again, optimization is applied to each path, although the open path is only reduced in scenarios where the value context of creating the other path affords the open path some usefulness; in all other cases, the open path code is identical to the original code. Figure 6.4(c) shows the code after *if-conversion*. For an open value transform, instead of including the open path with the value paths, a control instruction is created to relocate the execution to the original instruction sequence. This target location is called the *Escape Sequence* for an open transformation. Generally the open path is not included

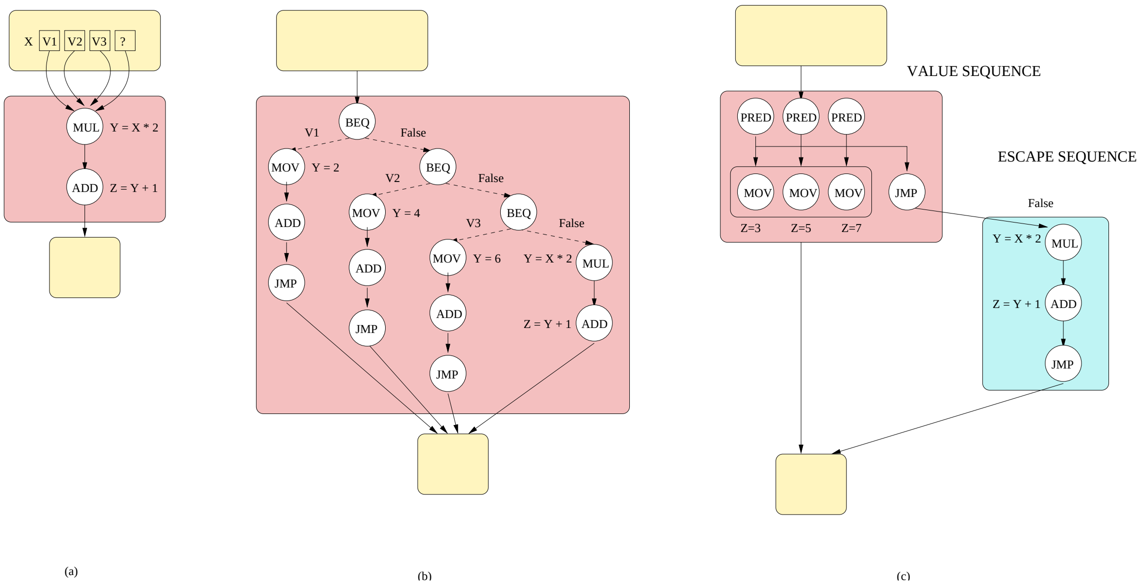


Figure 6.4 Code for one variable *open* region: dataflow representation (a), value-transformed control flow representation (b), and value-transformed dataflow representation (c).

as it represents the fundamental dataflow limit in the original sequence and because in the case of using value profiling, the open sequence is not the most frequently occurring path. Later in this chapter, an alternative is shown that does allow the instructions of the escape sequence to be combined with the value sequences.

6.3.2 Memory flow transformation

In addition to the recurrence of previously-seen program values in dataflow, a similar characterization of program behavior has been observed for memory instructions [9], [68]. The notion of *silent stores* has been introduced to describe store instructions with the dynamic execution behavior of writing values that match the exact value already stored at that memory location. Thus a silent store is defined as one that does not change the memory system state. Essentially the memory instructions express redundant computation that could be eliminated without changing the program correctness. There are several reasons for exploiting the behavior of silent store instructions. Eliminating silent stores can benefit the efficiency of a processor's memory hierarchy by reducing the number of dirty cache lines that require main memory write backs. Similarly, the operations computing the results of the silent stores can also be eliminated when the silent value property can be determined. Finally, other techniques such as computation reuse will benefit from communicating only true state changes to the memory system.

Figure 6.5 reports the fraction of all stores executed by each benchmark that are effectively silent. When an executed load instruction occurs in the same control block of a silent store, the store instruction is said to be a *paired silent store*. For these stores, there is more opportunity to initiate a memory flow transformation that eliminates the process of redundant store. Clearly from the results of Figure 6.5, there is a substantial opportunity to modify the information sent to the memory system from within the processor.

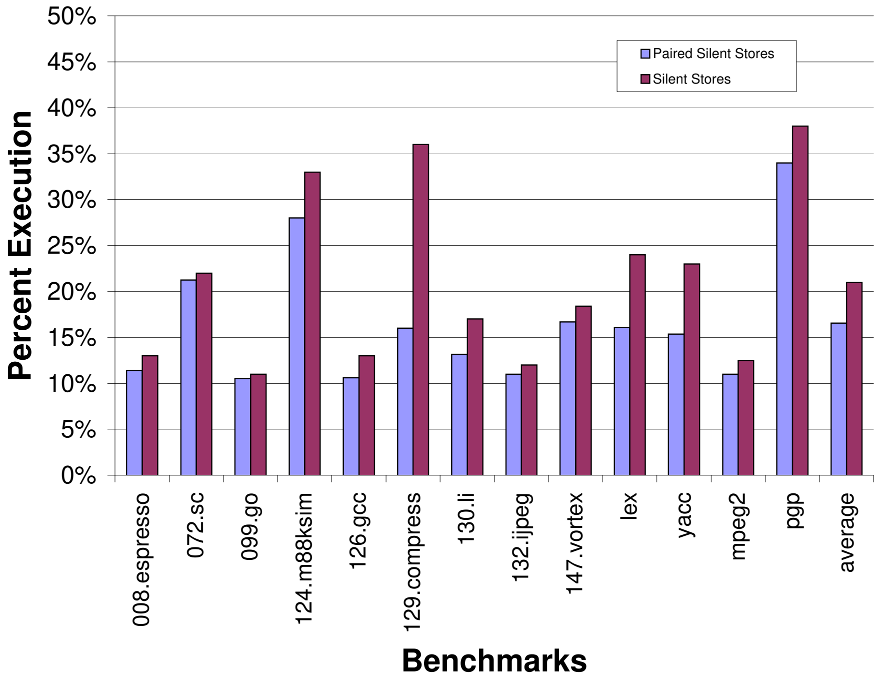


Figure 6.5 Execution distribution of silent store instructions.

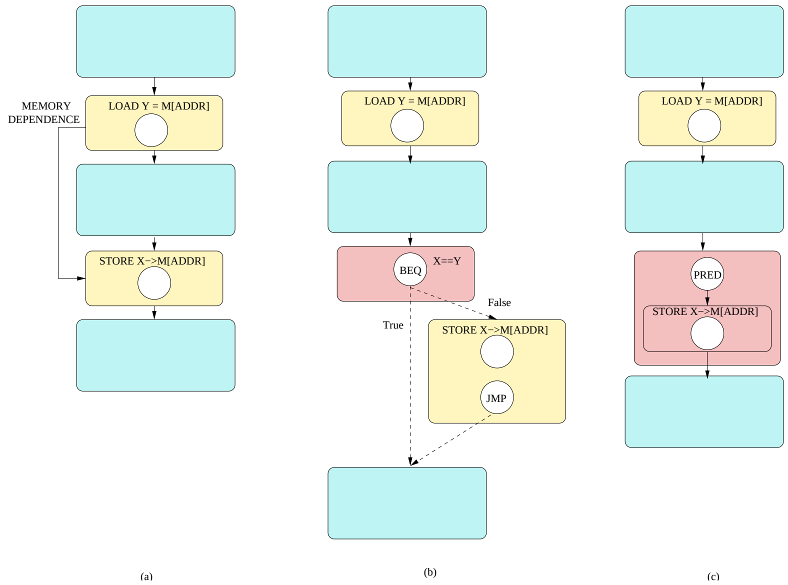


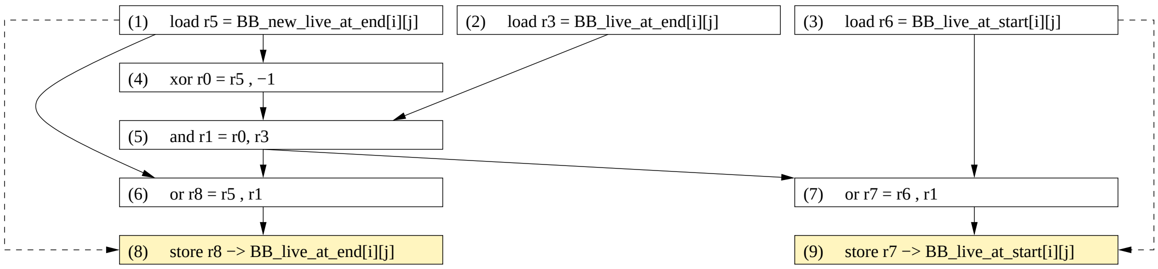
Figure 6.6 Fundamental conversion of memory flow (a) to control flow representation (b), and then to value-based predicate dataflow (c).

Figure 6.6 illustrates the general steps of the transformation. The first code sequence of Figure 6.6(a) shows the original sequence that includes a load instruction followed by a store instruction that communicates with the identical memory location for all invocations of the sequence. When value profiling or value analysis determines that the store is a silent store, then the control flow of Figure 6.6(b) can be inserted using a form of VCI. By creating the two paths, the store instruction is performed only when changing the resident value of the respective memory location. The code is if-converted and Figure 6.6(c) results, having a predicated store. The predicate condition verifies that the memory data is unchanged and squashes the store's execution on that occurrence.

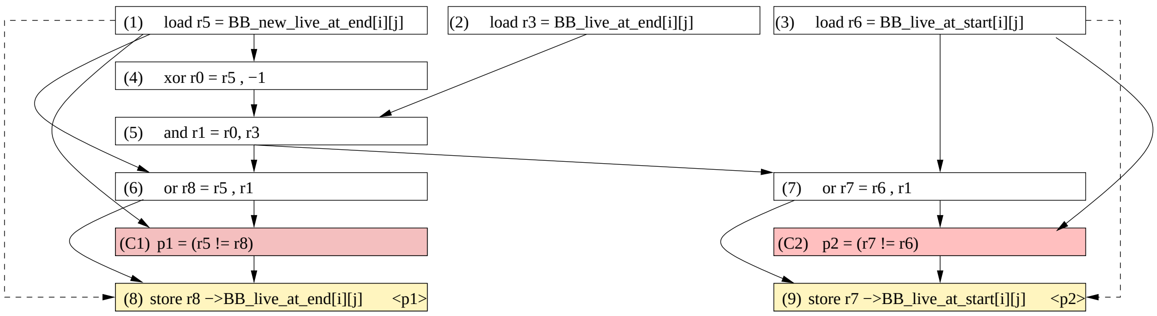
A memory transformation opportunity extracted from the application SPECINT95 benchmark suite *126.gcc* illustrates the primary benefit of the optimization and motivates the liklihood of finding other opportunities in applications. Figure 6.7 shows the code segment in various stages of program reformulation.

```
for (j = 0; j < regset_size; j++) {
    x = (BB_new_live_at_end[i][j] & ~BB_live_at_end[i][j]);
    BB_live_at_start[i][j] = BB_live_at_start[i][j] | x;
    BB_live_at_end[i][j] = BB_live_at_end[i][j] | x;
}
```

(a)



(b)



(c)

Figure 6.7 Example of memory flow transformation for *126.gcc*, showing the original code (a), the original dependence graph (b), the value-transformed predicate-asserted dependence graph (c).

The original source code for a loop is illustrated in Figure 6.7(a). The code invokes a subroutine that calculates the live variable analysis within a compiler. The routine is constructed using an iterative algorithm that updates information along all necessary paths until the information reaches all of its possible locations and the algorithm stops. It is well understood [69] that such compiler routines often compute sparsely distributed information such that many stores will frequently continue to write the same data. This example is rewarding since the inherent value locality of a subroutine technique coded for an application is identified, and a nontraditional solution to improving the code's efficiency is readily available through value-based transforms. Figure 6.7(b) shows the dependence graph prior to transformation. The highlighted instructions indicate the silent potential silent stores of the subroutine. The code after value transformation is shown in Figure 6.7(c) which includes new predicate definition instructions and predicated store instructions. Observed execution behavior of the sequence indicates that 60% of the store executions are redundant and do not need to update memory.

Overall, as stated there are benefits to performing this optimization within the memory system. On the other hand, the register live-range of the loaded value may possibly be extended to perform the necessary computation. Similarly, the store instruction may possibly be delayed in order to perform the necessary predicate comparison. Finally, to apply the optimization it is necessary to have precise memory disambiguation and analysis. On a positive note, there are substantial future opportunities that remain outside the scope of this thesis. Essentially, a similar form of the predicate-based memory flow transformation can be applied to aid in run-time memory disambiguation [70] and eliminate some dependence on data speculation hardware [48].

6.3.3 Advanced predicate optimization

Several advanced compilation techniques have been developed to enable the consistent application of efficient predicated code [54], [60], [71]. The two existing predicated optimization technologies supported within the IMPACT framework are the Predicate Decision Logic Optimization (PDLO) and Partial Reverse If-Conversion (PRIC) framework. Traditionally these techniques operate on predicated code that represented the original program control flow structure. The utility of these techniques is significantly enhanced by the value-based techniques that create new intermediate control flow structures. PDLO and PRIC techniques are able to operate on the original dataflow computation of the program rather then strictly adjusting the predicated representation of the control flow. Likewise, in turn these techniques complement the efficiency of the code produced using value transformation techniques.

Predicate Decision Logic Optimization. Motivated by the design of the Shared-Instance CRB, several opportunities exist to enhance the generation of both the decision and residue computation components of the value-transformed dataflow sequence. Namely, the design of the SI-CRB was conceived from the overlap of register value comparison within the hardware structure of the base CRB. Similar characteristics occur for the value-transformed codes, making any analogous predicate reformulation techniques equally applicable. The Predicate Decision Logic Optimization [54] is a method of simplifying the program decision logic by means of sharing value comparisons. The basis for the technique is that since predicate registers represent Boolean values, predicate registers and the respective predicate definition instructions can be represented using combinational logic. Once the logical representation is derived for a sequence of predicate definition instructions, circuit optimization techniques in the computer-aided design (CAD) field can be utilized to reduce the decision logic guarding the residue computation.

In the IMPACT compiler, the derived Boolean function is represented with a binary decision diagram (BDD) [72]. The BDD algorithms used are described in [73]. The *predicate BDD* contains the relationship among predicates as defined by the network of predicate define operations. The predicate BDD is used throughout the compiler as a database for queries made by optimizations when operating on predicated code. Queries to the BDD are made in IMPACT by the optimizer, the scheduler, and dataflow analysis. The BDD maintains a canonical representation of the decision logic functions, from which a Boolean sum-of-products expression can be produced for any represented function. Note that the expression thus generated reflects the canonical nature of the BDD's internal representation, and is usually not optimal for expressions with multiple product terms. Therefore, it is necessary to optimize the derived expression before attempting to synthesize a predicate defining structure.

Value-transformed code segments that can leverage the Boolean minimization techniques are referred to as *Vectored Computation Regions.* Each vector represents a computation instance which, as described, is a set of register indexes each with a respective value. Multiple vectors for a computations region are indicators of the computation variation. Similar to the *order* of value information, the *order* of a vectored computation region refers to the number of computation instances. In addition to requiring overlap of register comparison between the vectors (instances), for effective performance improvement, the computation vectors must also contain persistent values. Since the predicate reformulation synthesizes code based on data values, the effectiveness remains as long as the values observed during execution are those synthesized using the VOF techniques.

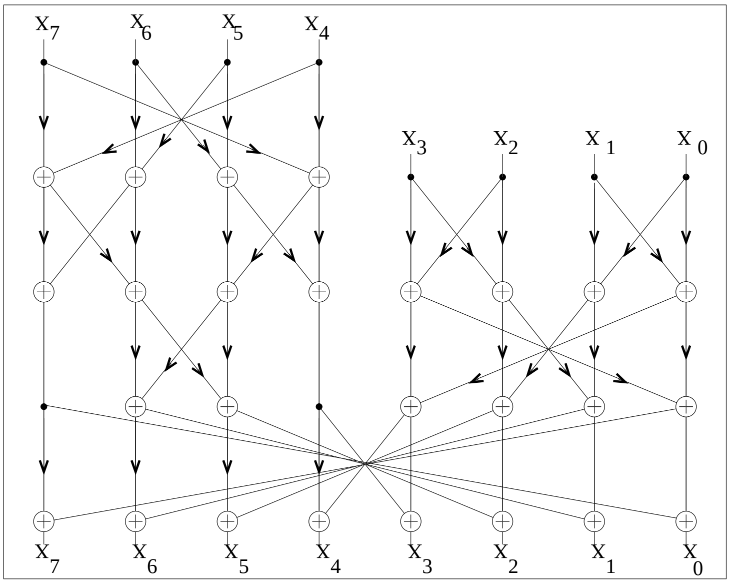


Figure 6.8 MPEG2 IDCT dependence graph.

Given the observed opportunities to utilize the SI-CRB, many relevant opportunities for the Boolean minimization were found. Figure 6.8 illustrates an example previously shown in Section 3.4. This example shows a computation region formed from the row transform of an MPEG2 inverse discrete cosine transform (IDCT) using the Chen IDCT algorithm [46].

As previous noted, typically DCT blocks of MPEG-compressed video sequences have only five to six nonzero coefficients, mainly located in the low spatial frequency positions [47]. These execution properties are generally known characteristics of the transform and represent persistent values. Figure 6.9 illustrates the average value execution for an 8×8 coefficient block of the IDCT transform. For the *mpeg2* application, each eight-element row is processed and then the series of each eight-element column is processed. Each execution begins by placing eight memory values into eight register operands. In the case of the CCR approach, the IDCT computation region has significant reuse locality (80%) for a history of 16 instances of the row computation. As can be observed in the data, there is high probability that value comparisons can be shared since only the data location $Block[0][0]$ has a significant execution percentage

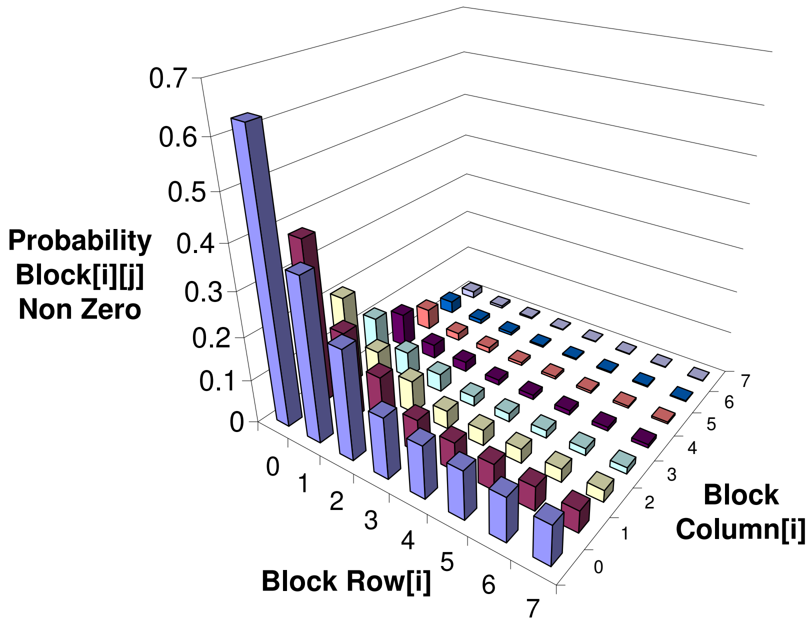


Figure 6.9 Nonzero value distribution of DCT blocks of MPEG compressed video.

of nonzero values. Identically, the register operands establish the point of specializing a code sequence prior to initiating the IDCT computation

Figure 6.10 illustrates the case of applying the base degree-1 techniques of Section 6.3.1 to the MPEG IDCT code region. Figure 6.10(a) shows the original code and Figure 6.10(b) depicts the generated decision components and the vector escape sequence to the original code region. The vector decision components can be optimized using the Boolean predicate minimization techniques [54]. The initial decision components of the transformed code correspond to logic diagrams. A two-level network composed of the input decision component and the output result move component is presented in Figure 6.11(a). The two-level network is analogous to a sum-of-products (SOP) expression. Figure 6.11(b) illustrates the expression after the two components have been analyzed, optimized, and reformulated to eliminate redundancy and inefficiency in the original value-transformed code sequence.

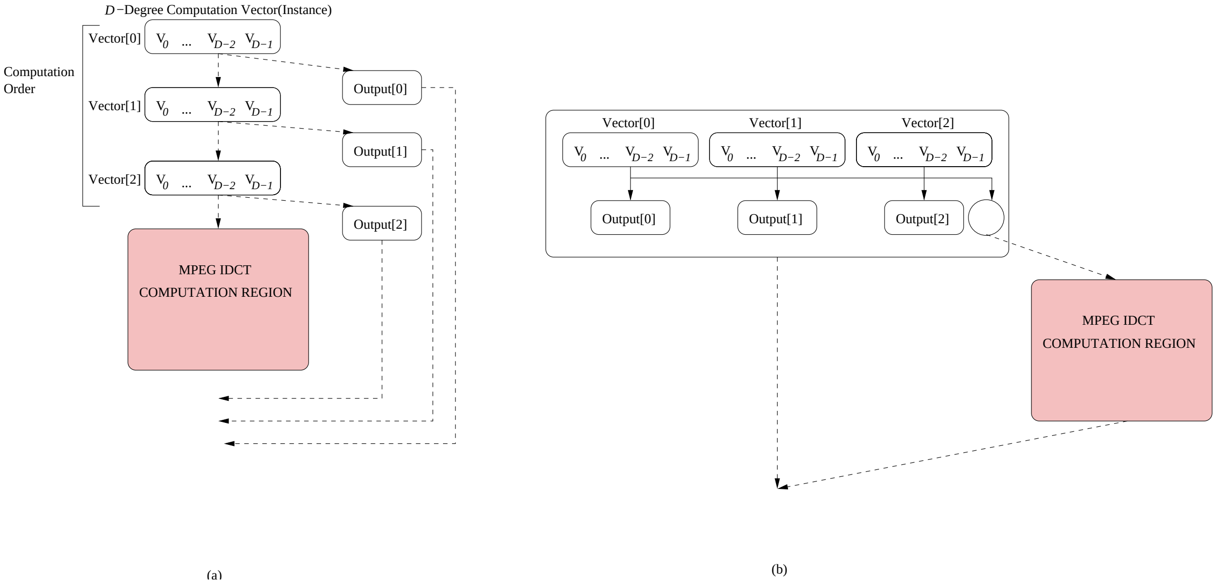


Figure 6.10 MPEG reformulation of degree-8 vectors with the three open value vectors, control flow representation of value-transformed code (a), and dataflow representation of value-transformed code (b).

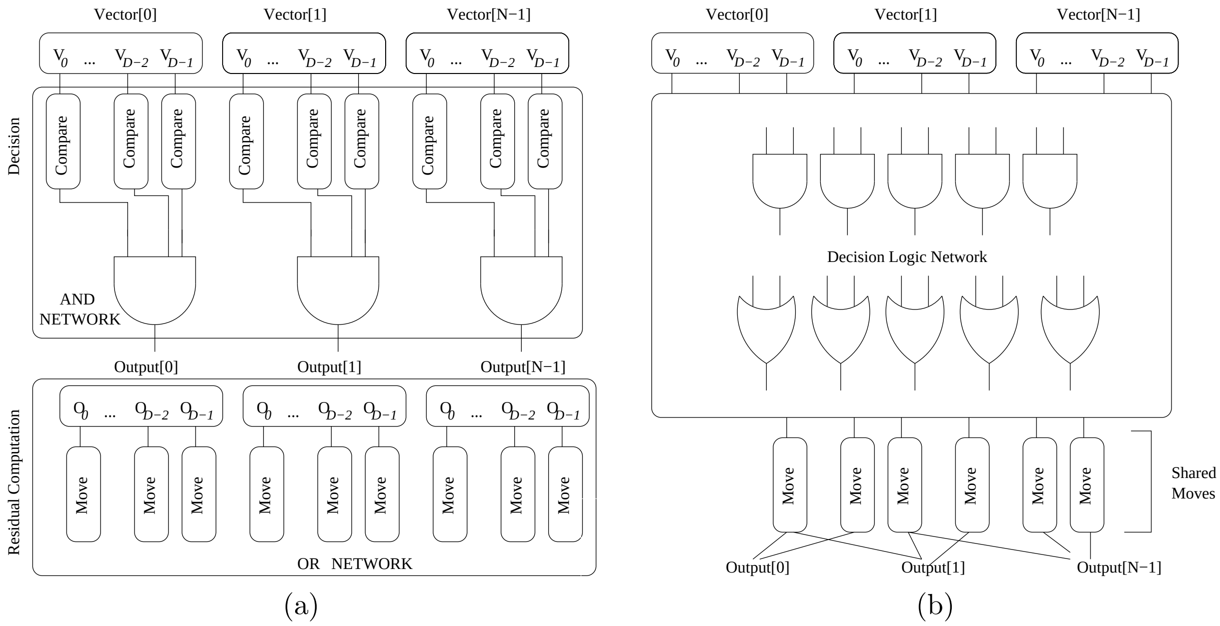


Figure 6.11 Process of decision logic optimization, original logic network (a) and optimized decision logic network(b).

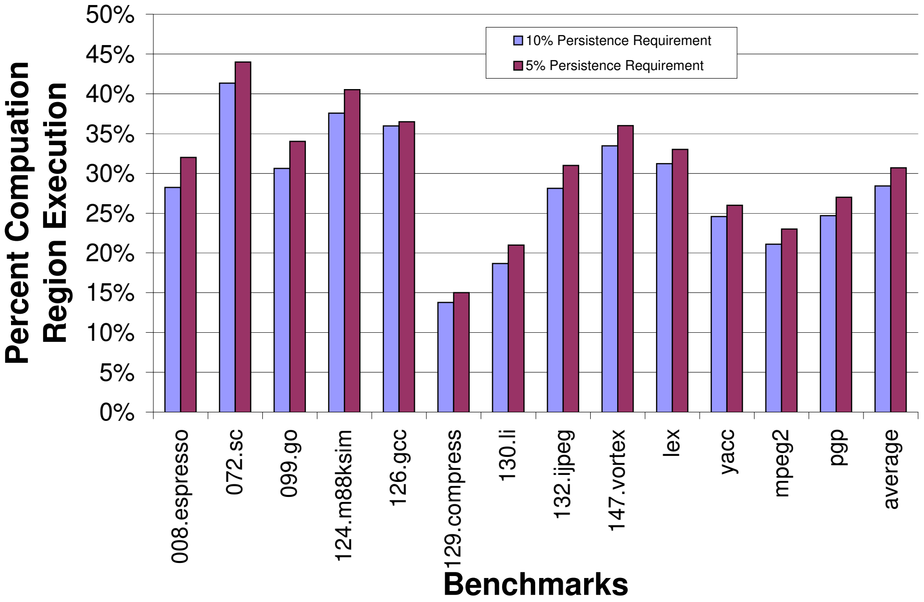


Figure 6.12 Computation region execution with persistent vector characteristics.

Figure 6.12 indicates the amount of region execution observed during profiling that qualifies for potential classification as persistent value vectors. To gauge the persistence of a computation region, profile information was gathered for both the training and the reference input. Value vectors overlapped in both input sets and accounting for greater than 5% of the respective region execution. This setting was determined by experiments to find the maximum percentage of persistent value vectors. Overall, about 20% of region execution shows promise in being reformulated using either the dataflow techniques. The results of evaluating the SI-CRB also help conclude that there is significant overlap of computation comparisons and output values to allow the Boolean optimization to enhance value vector reformulation.

Partial Reverse If-Conversion. As discussed in Section 6.3.1 the escape sequence of an open compiler-synthesized region is typically not included with the value paths when generating the new dataflow trace of instructions. Essentially if it were, the original dependence height would establish the same fundamental limit on execution as in the original nonvalue transformed representation. However, the concept embodied in the *Partial Reverse If-Conversion* technique [60] can be used to improve the execution efficiency of the escape sequence. The partial reverse if-conversion framework operates at schedule time to balance the amount of control flow and predicated present in the code originally generated using hyperblock formation techniques. For the VOF techniques, the decision computation and residue computation components can be formed with a balance of some components from the original code sequence. This allows effective generation of value-based sequences that match the characteristics of the target processor and reduce the penalty of generating value-based sequences that do have differing run-time value profile characteristics than those observed during the value profiling. Figure 6.13 illustrates the computation region that performs the population count of the SPECINT95 application *008.espresso* that was presented in Chapter 3. The code has one input register operand, and the value profile directed the generation of three value-based context paths.

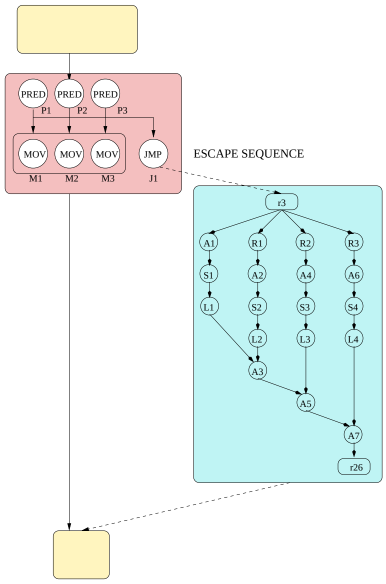


Figure 6.13 Value-transformed sequence for population count example of *008.espresso*, a degree-1 region with an open order-3 value transform.

Cycle	Value sequence - Instructions Issued					
0	op P1	op P2	op P3			
1	op M1	op M2	op M3			op J1

(a) Schedule for the value sequence in Figure 6.13.

Cycle	Escape Sequence - Instructions Issues					
0	op A1	op R1	op R2	op R3		
1	op S1	op A2	op A4	op A6		
2	op L1	op S2	op S3	op S4		
3	op L2	op L3				
4	op A3	op L4				
5	op A5					
6	op A7					

(b) Schedule for the escape sequence in Figure 6.13.

Figure 6.14 Static schedules for the 008.espresso value and escape sequence.

The transformed sequence includes an escape sequence to guard against any potential execution condition not matching the value-inserted predicate paths. For the basic optimization path, Figure 6.14 illustrates the static code schedules for both the value and escape sequences for a general machine model of a six-issue processor. When the execution of the original code matches one of value paths, a two-cycle latency known as the *value sequence latency (VSL)*, shown in Figure 6.14(a), is required. The original code, shown as the escape sequence of Figure 6.14(b), requires seven cycles for execution, known as the *escape sequence latency (ESL)*. When the execution of the original code does not match one of the value paths, the latency $VSL + ESL$ is incurred, an occurrence known as the *max escape latency (MEL)*. Nine cycles in this case. The static IPC[2] for the code sequences are 3.5 and 2.6 respectively. Although there is high utilization of the processor resources for the value sequence, there is also an opportunity to migrate instructions from the escape sequence to the position of the value sequence.

Applying partial reverse if-conversion techniques to balance instructions between the escape sequence with the value sequence creates the two schedules shown in Figure 6.15. In this

[2]Calculating IPC with predicated execution is obscure since instructions are conditionally executed.

Cycle	Value sequence - Instructions Issued					
0	op P1	op P2	op P3	**op A1**	**op R1**	**op R2**
1	op M1	op M2	op M3	**op R3**	**op A2**	op J1

(a) Schedule for the value sequence in Figure 6.13 using PRIC.

Cycle	Escape Sequence - Instructions Issues					
0	op L1	op S2	op A4	op A6		
1	op L2	op S3	op S4			
2	op A3	op L3	op L4			
3	op A5					
4	op A7					

(b) Schedule for the escape sequence in Figure 6.13 using PRIC.

Figure 6.15 Static schedules for the 008.espresso using PRIC techniques.

example, the utilization of machine resource during the value sequence has improved since the instructions shown in bold type have been placed with the value sequences. Likewise, by initiating instructions from the escape sequence concurrently with the value sequence, the max escape latency is reduced to $MEL(7) = VSL(2) + ESL(5)$. This example motivates the use of compiler support for PRIC to enhance the VOF techniques. To perform partial if-conversion of this form on the value-based transformed codes, it is necessary to determine how many instructions from the escape sequence can be included with the value sequence without harming the performance efficiency of the value-transformed code.

Although the PRIC technique offers an array of methods for determining when to reverse if-convert a hyperblock, in predicated codes of VOF techniques applicable to reverse if-conversion, there are fewer decisions on where to initiate the control insertion. As such, only a simple form of PRIC was implemented and is discussed in the following section.

6.4 Compiler Framework

The support for compiler-directed computation reuse and value-based program reformulation are related in many ways. Figure 6.16 illustrates an overview of the compiler technologies applied to eliminate dynamic computation redundancy. Both methods share the RCRF techniques for identifying regions of computation with exploitable value behavior. The RCRF is in turn based on the RPS for profile-guided region generation and the the RAS static region formation. The partitioning of regions for either compiler reformulation or compiler-directed reuse is performed using region heuristics in the Computation Region Assessment (CRA) phase.

The application of the compiler value-based optimization framework interfaces to several existing ILP components. At run-time, these regions require predicated execution support. The regions assigned to the CCR approach utilize compiler support for constructing the computation regions (RCS), which in turn use the DCF for inserting invalidation instructions for memory dependent regions. The CRB and DCMS provide run-time support for the CCR approach. The following sections describe the compiler support for the VOF optimization techniques and the overall method for coordinating the assignment of regions for VOF reformulation or compiler-directed redundancy elimination.

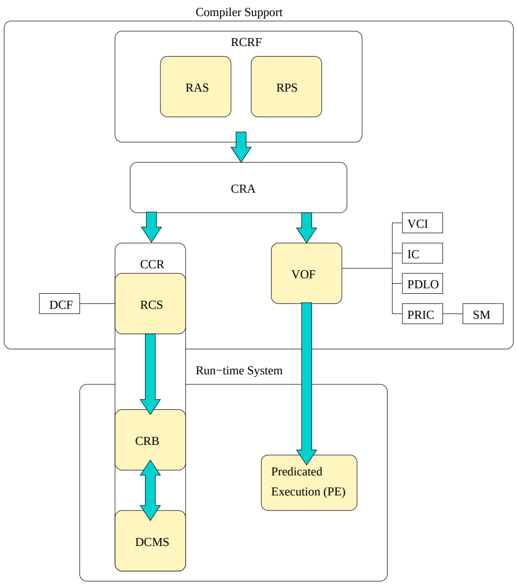


Figure 6.16 Overview of applying compiler-based and compiler-directed techniques.

6.4.1 Coordination of compiler-based and compiler-directed techniques

As shown in the results of the previous section, upwards of 30% of computation region executions exhibit stability of values being executed. The persistence of values is imperative for compiler-based value transformations, since only the compiler-directed technique that uses hardware buffers can capture large variations in dynamic redundancy. The Computation Region Assessment (CRA) performs a profile-based check of the conditions between the multiple input evaluations to detect these regions. The region persistence is judged based on the following criteria. First, when examining the training value invariance, the execution weights of the top values are collected. The invariance is calculated on an instruction-by-instruction basis and is averaged over the region. It is necessary for the value invariance to exist for the individual instructions and for the composite region metric to also indicate sufficient value persistence. Equation 6.1 gives the basic persistence relation.

$$Reuse\_Persistence(i) = \left(\frac{Invariance[k](i)}{Exec(i)} \geq R_p\right) \tag{6.1}$$

The setting of R_p is varied depending on its use in compiler-based code reformulation or compiler-directed computation reuse. Setting R_p to .60 and the number of invariant values to from 5-10 typically accounted for good persistent candidate region behaviors. When examining the invariance between the two profiles, the overlap of the invariance in the top values collected is used to judge the overall program persistence for those regions. The overlap in invariance is calculated by determining the weighted average of execution for the top invariance values that occur in both profiles. The Reuse_Persistence_Overlap for each profile input must be greater than $R_{p\_overlap}$ for the the specified top input m values of each input. Essentially, the top

m invariant values must account for greater than $R_{p\_overlap}$ of the program execution for the other input evaluation. Generally $R_{p\_overlap}$ is set to .50 and m to 10 to determine persistence between the two input set evaluations. There are several other cases that amount to deciding on the inherent persistence of a region. In the case of a region being executed a large portion of time for only one input set, then heuristically, the persistence characteristic is attributed based only on the single input set. Another case is when there are few invariant values for both input sets, yet there is little overlap between the invariant values. This behavior is captured as a persistent opportunity since the compiler reformulation transformations likely to enable a small number of multiple concurrent value-based sequences can be performed at no cost. When performing the transformations of the following sections, value vectors are constructed from both input sets and ranked according to a weighted distribution based on the total execution of both input sets.

6.4.2 Value Optimization Framework

The previous section provided a motivating overview of the process of value transformation and predicate optimization using advanced optimization techniques. This section describes in detail the mechanisms by which the value-transformations are applied.

Once vectored computation regions have been identified, value-based predicate transformations are applied in several steps. First, value vectors are generated to represent the value execution with the highest priority of executing. These vectors are then converted using the VCI and if-conversion systems. The program decision logic of the vectored regions can then be presented by the predicate BDD and the condition BDD, and the SOP expression is generated for the region's decision and output components. The sum-of-products expressions are

```
VOF_Transform_Regions (Vectored_Region_Candidates)
FOREACH region IN Vectored_Region_Candidates(Program) DO
    // Determine vectors to exploit for region
    region.vectors = Generate_Vectors_for_Region(region);
    // Insert value control flow
    region = VCI(region, region.vectors);
    // If-Convertion to convert to predicated representation
    region.ic = IC(region);
    // PDLO-Extract Sum-Of-Products Expression
    region.sop_info = SOP(region);
    // PDLO-Optimize the predicte boolean representation
    region.pred_opti = Pred_Opti(region.sop_info);
    // Generate predicate definitions
    region.code = Predicate_Optimization(region.pred_code);
    // PRIC-Partial reverse code
    region.code = PRIC(region.code);

Generate_Vectors_for_Region (region)
Region_Vectors[N] = Sort (region , Profile_Information)
FOREACH vector IN Region_Vectors DO
    vector.path = Optimize(vector);
    vector.count = Count(vector.path);
    vector.height = Dependence_Height(vector.path);
    // Apply vector selection priority function
    vector.priority= Vector_Selection_Priority(vector,VSP);
Region_Vectors[N] = Sort (Region_Vectors , VSP) // Sort based on VSP
Selected_Vectors = NULL;
FOREACH vector IN Region_Vectors DO
    IF (Vector_Count(vector,Selected_Vectors,region) == FALSE) THEN
        continue;
    IF (Vector_Performance(vector,Selected_Vectors,region) == FALSE) THEN
        continue;
    Selected_Vectors = Selected_Vectors ∪ vector;
return (Selected_Vectors)
```

Figure 6.17 Algorithms for value optimization techniques.

represented by the artificially inserted value conditions. These expressions are then optimized using condition analysis and traditional Boolean logic minimization techniques. The resulting decision component is expressed using predicate definition instructions. Finally, partial reverse if-conversion can be employed to create efficient balance between the value sequences and the escape code sequence. Figure 6.17 illustrates the process of applying the value-based transformations and the predicate-based optimizations. The function $VOF\_Transform\_Regions$ performs the activation of the steps of the value-based transformations.

Value-vector Generation. The generation of value vectors for the selected code region begins with the extraction of the prioritized vectors for the computation region. A two-step process is used to generate vectors for a computation region. First, vectors are prioritized according to the potential benefit of the value sequences they may generate and their cost. The priority is based on the form of the heuristic function shown below:

$$VSP_{vector} = Frequency_{vector} \times \left(\frac{count\_region}{count\_pathvector} \times \frac{heightregion}{height\_path_{vector}}\right) \tag{6.2}$$

The vector selection priority (VSP) is calculated for each vector prior to constructing a set of value vectors for the value transformations. The frequency factor is the amount of region execution incurred by the value vector. The count and height terms indicate the number of instructions and the dependence height of the code region prior to transformation. The elements that include a path term are for the optimized path created for evaluating the benefit of the value transform to the code region. After the construction of a region's prioritized vectors, generation of the region's order is performed by gathering the composite set of vectors that can potentially transform the dataflow region. The function $Generate\_Vectors\_for\_Region$ in Figure 6.17 illustrates the general form of the generation technique. Essentially from the top vectors ordered by the VSP priority, vectors are selected allowing the composite set to maintain the following relationships with the original code.

A relation that limits the code size of the set of value-vectors creation is applied to reduce the final code size of the generated sequence. An estimate of the number of operations in the composite set of computation regions is acquired by calculating the operations in each value path. The total count of operations of the value sequence (computation and predicate decision instructions of all vectors) is summed, and an estimate of the overlap between each value sequence is then subtracted. The overlap terms account for the reduction of operations through predicate sharing, operation sharing, and output sharing. The estimated composite set may only exceed a multiplied factor of the original operation count.

A second relationship, shown in Equation (6.3), is applied to estimate the performance of the value sequence. The performance is calculated by adding the the frequency of executing of each potential value vector in the composite set as the estimate of value-sequence path execution. The value execution weight is then used to determine if the execution of the value sequence will make a significant performance improvement for the original code sequence. The dependence height for the original sequence and value sequence are used for estimating the performance of the sequence. Essentially two path estimations are made. First is the execution time of the value sequence $VSL \times Freq_{VS}$; then the escape sequence is calculated as $ESL \times (1 - Freq_{VS})$. A penalty for the branched escape sequence is made based on the frequency of escape. A factor $K_{performance}$ is used to force computation regions to show improvement over the original code sequence.

$$ESL > K_{performance} \times (VSL * Freq_{VS} + (VSL + ESL + Penalty_{escape}) \times (1 - Freq_{VS})) \quad (6.3)$$

Memory flow transformations to eliminate redundant memory system activity are identified by searching for individual instructions that exhibit silent store behaviors. The VOF system treats these opportunities as degree-1 closed order-2 variation regions since the transformed store instruction either does or does not match an earlier load instruction. After value vectors for a computation have been selected using both the operation count and the operation, value-based control insertion for the value and escape sequences is performed using the VCI process.

The value-based control flow is annotated for the if-conversion (IC) process to transform the representation to the predicate dataflow domain. Essentially, an artificial hyperblock is formed from the original dataflow, and the dataflow is receptive to to predicate optimization techniques.

Predicate Decision Logic Optimization. Once the program decision logic has been extracted in the SOP form based on program conditions, predicate optimization techniques can be applied. Predicate expressions are optimized in two steps. First, expressions are reduced using condition BDD information. Conditions such as redundant literals in product terms are removed in this phase. This is analogous to the hardware of the SI-CRB sharing comparison operations. Once redundant products and literals have been removed from the predicate expressions, the iterative-consensus method is applied to produce a complete sum, and then to select a subset of prime implicants for a simplified two-level logic implementation [74]. The focus of this iterative algorithm is the consensus-taking routine, which applies the Boolean theorem $x+\overline{x}y \rightarrow x+y$. After each pass through the product list, products subsumed by other products are removed. The iterative-consensus algorithm generates a complete sum for the input expression. Nonessential products can then be removed to generate a minimal covering sum. Within the IMPACT PDLO system, a heuristic-based mechanism of the iterative-consensus application is used to prevent the time of the algorithm from requiring an excess of compile time.

Following optimization of the predicate expressions, the control logic can be synthesized most intuitively as a two-level predicate define network which directly evaluates the minimized sum-of-products expression. Two levels of predicate define instructions are used for each predicate. The first level consists of and-type predicate defines of the form $p_i\_at = C_i\langle T\rangle$, where one predicate p_i is defined for each product term in the predicate expression, and T is the TRUE predicate, which always has the value 1. The second level consists of or-type predicate defines of the form $p_j\_ot = (condT)\langle p_i \rangle$, where there is one such predicate define for each product (p_i) and $condT$ is an invariant TRUE condition (e.g., $(0 == 0)$). Predicates which have products in common can share intermediate predicates, allowing for some savings through reuse. In most cases, redundancy between products is reduced due to the characteristics of value locality for the dataflow sequences creating many shared value comparisons. Other techniques to perform factoring on the logic expressions are possible. However, since the predicate expressions are generally strongly connected, the resulting predicate definition structure can be quite manageable. Techniques [54] showed that this is not the case with predicate expressions originally created from control flow structures.

Partial Reverse If-Conversion. Once predicate code has been created and optimized for the value sequences generated from the code exhibiting value locality, a form of partial reverse if-conversion is performed. To do this, the escape sequence is predicated upon the condition of the escape branch instruction condition. The process determines whether instructions of the escape sequence should be executed concurrently with the predicated instructions of the value sequences. Generally the advantage of initiating the escape sequence in parallel with the value sequences must be weighed in terms of delaying the effectiveness of the value sequence. In the originally proposed PRIC framework applied to hyperblocks, there were many decisions on where to initiate the control insertion. Instead for the value-based if-converted region of code, there is a limited number since it is only desirable to migrate escape sequence instructions between the value sequence block and the the escape sequence location.

The subroutine code of *VOF_PRIC_schedule* in Figure 6.18 illustrates the method of applying the partial reverse if-conversion principle at schedule time. To begin, after creating the combined predicate region, composed of both the value and escape sequences, the compiler makes the decision on which instructions of the escape sequence to remain in the value sequence block. To determine the appropriate amount of code to reserve if-convert from the entire region, the compiler considers only the n instruction of the escape path sequence. To choose from any of the instructions of the value sequence would contradict the original generation of the value-vectors. While future heuristic-based techniques may investigate opportunities to perform other reverse if-conversion policies, the current vector generation process achieves an effective balance between the escape and value sequences.

```
VOF_PRIC_Schedule (Region)
Initialize block_schedule; // Schedule value sequences first
Insert ( block_schedule, Region.value_ops);
Prioritized_List_Schedule(block_schedule);
// Each iteration schedules an instruction
pric_schedule = TRUE;
WHILE pric_schedule != NULL DO
  pric_schedule = FALSE;
  // Handle escape sequence instructions
  FOREACH escape_op IN region DO
    // Allow escape instruction that can be scheduled (dependents have been scheduled)
    IF Can_Schedule_Op(escape_op, block_schedule) THEN
      //Compute schedule for escape op;
      SM[escape_op].value_block = Insert(block_schedule, escape_op);
      //Insert reverse if-converting branch at earliest location
      SM[escape_op].value_block = Insert_Branch_in_Schedule(SM[escape_op].block, ric_op);
      //Schedule for value sequence block
      SM[escape_op].value_block = SM_Schedule(SM[escape_op].value_block);
      //Create and Schedule escape sequence block
      SM[escape_op].escape_block = Create_Block(region.unschedule_escape_ops);
      SM[escape_op].escape_block = SM_Schedule(SM[escape_op].escape_block);
      //Compute scheduling information for the value block
      SM[escape_op].schedule_value = Compute_cycle_time(SM[escape_op].value_block);
      SM[escape_op].schedule_escape = Compute_cycle_time(SM[escape_op].escape_block);
      //Compute schedule priority for the PRIC sequences
      SM[escape_op].priority_pric = PRIC_Priority;
  candidate_ops = Sort (SM);
  top_candidate_op = Sort (candidate_ops, (PRIC_Priority)); //Sort schedules on pric priority value
  IF (SM[top_candidate_op].schedule_priority ¡ Threshold) THEN
    pric_schedule = TRUE;
// Place top candidate op in block
Insert ( escape_block_schedule, Region.unscheduled_escape_ops);
Prioritized_List_Schedule(escape_block_schedule);
```

Figure 6.18 Method of scheduling escape sequence instruction using partial reverse if-conversion methodology.

The process of reverse if-converting the computation region is developed using the list scheduler and the *Schedule Manager (SM)* [75] of the IMPACT compiler. In fact, the hyperblock-based PRIC framework within IMPACT also integrates SM technology. The first step of the process performs list-based scheduling on the instructions of the value sequence. The list scheduler performs a cycle-by-cycle schedule using an instruction priority function. The second step performs prioritized operation scheduling for the partial reverse if-conversion framework on first schedule. This allows the escape sequence instructions to be integrated with the value sequence schedule and assessments can be made over the efficiency of the reverse if-conversion scenario.

The process begins by taking all of the instructions that are available for scheduling and that are parts of the escape sequence and scheduling each of them using the scheduling manager. Different schedules are created for each candidate escape instruction. Each schedule includes the scheduling of a reverse if-converting branch to account for the branch resource necessary to perform escape generation. For every candidate instruction scheduled, the schedule cycle times of both the value sequence and escape sequences are computed. The cycle times are used to compute a prioritized list of the schedules.

To make the decision on whether each candidate instruction should be included with the value sequence, the estimated profitability for the resulting code schedule is evaluated. For each candidate instruction, three schedule times are considered. First, the original code (both value and escape sequences prior to PRIC) schedule is used as a baseline of the performance that should be improved. This schedule is the exact formation of the $VSL(Original) + ESL(Original)$. The second schedule is the value sequence code schedule with the candidate instruction inserted. The third schedule is the escape code sequence schedule remaining after the candidate insertion with the value sequence. Each of these three schedules needs to be compared to determine if a reverse if-conversion of the value-transformed code is profitable. This relation computed is: $schedule\_cycles_{original} > K_{pric} \times (schedule\_cycles_{value\_sequence} + schedule\_cycles_{escape\_sequence+escape\_penalty})$. This operation calculates the cycles by using the schedule height and the frequency of execution of the respective sequences. The frequency of executing the value sequence VS is estimated by the information used to generate the initial value vectors. The term *escape_penalty* is related to the penalty of the processor in having to execute the branch instruction and the respective number of mispredictions introduced by the reverse if-conversion branch. Generally an additional factor K_{pric} can be applied to make sure that there is sufficient slack in the profitability calculation at run time. As such, the required relation of the two potential reverse if-conversion schedules (apply/no apply) is set to force the application to have no less than a K_{pric} speedup. Thus the candidate selection must offer the greatest improvement of schedules and show marked improvement over the original code. An additional constraint is that the new schedule for the value sequence cannot be greater than 30% of the original value sequence schedule time. This maintains the achieved speedup of the value-based sequences.

The process of scheduling operations and finding a single candidate escape instruction to be placed with the instruction sequence is continued until the PRIC scheduling relation can no longer be matched. The scheduling of the code can be expensive; however, the schedule manager framework is extremely efficient in performing incremental operation scheduling, allowing many scheduling variations to be performed, each in near constant time.

6.5 Experimental Evaluation

6.5.1 Methodology

The IMPACT compiler support was enhanced to include the transformation framework techniques proposed in Section 6.4. In the experimental evaluations of the compiler transformations, the base processor modeled can issue in-order six operations up to the limit of the available functional units: four integer ALU's, two memory ports, two floating point ALU's, and one branch unit. To study the benefits of applying the predicate-based transformations, as well as the computation reuse effect on the fundamental dataflow limitations, several other machine models were investigated. Each model has an increased number of functional units over the base processor modeled. In addition to the 6-wide machine, a 9-wide, 12-wide, and 16-wide machine were also evaluated. Table 6.2 illustrates the breakdown of functional unit resources for the models investigated. For all other purposes, the architecture resources and compilation environment explored with experiments in this section were generated using the same methods outlined in Section 4.5. Additionally, two perfect models of processor were explored. The first has infinite functional unit resources and an instruction is never stalled in the process of waiting for a function unit. The second perfect model concerns the CRB design; in some experiments, an ideal CRB capable of recording all computation regions and computation instances is modeled.

Table 6.2 Evaluated processor models.

Name	Issue-Width	Function Units (Int, Mem, Float, Branch)
Issue-6	6	4, 2, 2, 1
Issue-9	9	8, 4, 3, 1
Issue-12	12	12, 6, 4, 1
Issue-16	16	16, 8, 5, 1

Table 6.3 VOF optimization parameters.

Support	Relation	Policy
Vector Generation	Reuse Persistence R_p	.60 for max 10 elements
	Persistence Overlap $R_{p-overlap}$	50% for 10 elements
	$Penalty_{escape}$	8
	$K_{performance}$	1.3
PRIC	$Penalty_{escape}$	8

A number of variations for the different compiler support relations were evaluated during the investigation of the techniques for eliminating dynamic redundancy. Table 6.3 illustrates the parameters used for the experiments of this section.

6.5.2 Results and analysis

Value-based Reformulation. Figure 6.19 illustrates the height reduction of applying the value-based transformation to the computation regions. The height reduction is computed by taking the dependence height (control and data) of the original computation region and dividing it by the height using value-transformations. Two columns are included to indicate the height calculation with the computation and the computation and decision elements (predicate instructions). The results indicate that the value-transformed codes are able to effectively transform the original dataflow representation of the computation regions to new predicate-enabled dataflow regions with substantially reduced dependence heights. Generally the original dependence heights are three times the size of the respective value-transformed codes. An additional amount of height is reduced by simply eliminating the control instructions from the height calculation. When this is evaluated, the average result for transformed regions is that the original code exceeds 3.25 times the value-transformed codes. This result indicates that compiler can effectively use value distributions to synthesize efficient code sequences.

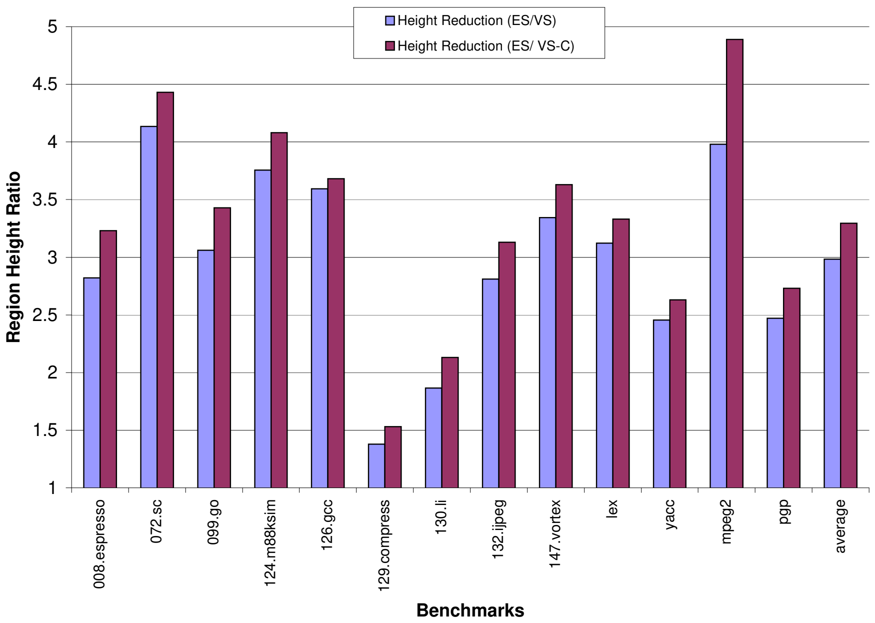


Figure 6.19 Average relative height reduction for dataflow transformed regions.

Results for performing VOF techniques for the 16-issue machine are presented in Figure 6.20. Performance is shown with and without the help of the PDLO support, and the base model being compared is also a 16-issue processor model. Generally the results indicate that the performance of compiler-based reformulation can achieve an average improvement when given sufficient machine resources. On average a performance improvement of nearly 6% is observed for the 16-issue machine. The modest result is achieved in comparison with the 16-issue machine without value-transformed code and thus indicates overall success in extending the effectiveness of wide issue processors. Namely, the results indicate that transforming computations regions can obtain good performance even when the regions are consistently limited by data dependence height. Since parallel resources reach hard limits in improving dependence height limited regions, the modest performance improvement of the value-based transforms become significant.

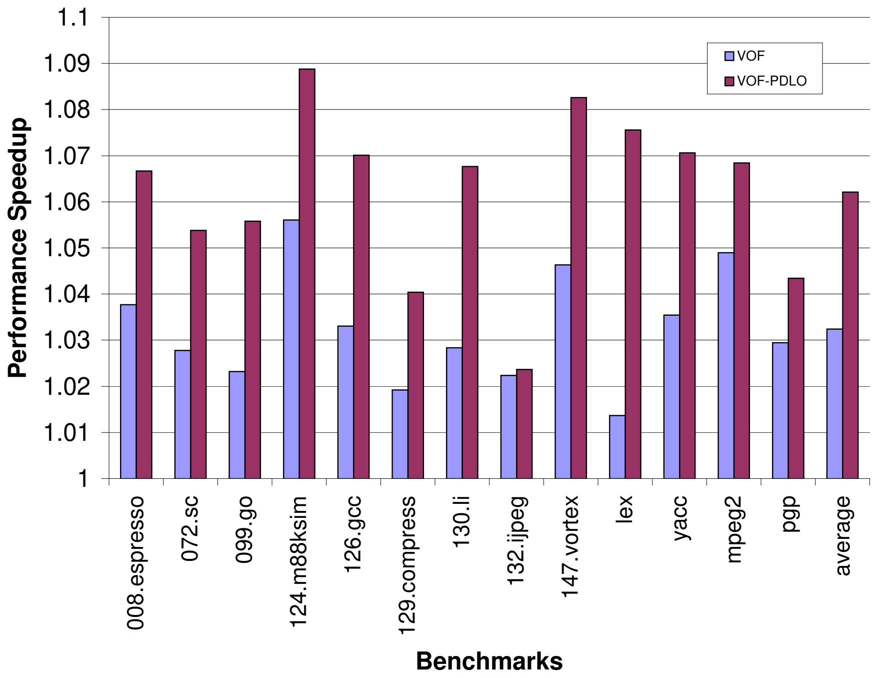


Figure 6.20 Performance for VOF and VOF-PDLO optimization.

An evaluation of VOF and PDLO compiler techniques for different machine models is shown in Figure 6.21; in this case, the limited machine models are not able to aggressively improve performance. Instead, for the six-issue model, the parallel code synthesized of predicate definition instructions is over-running the limited execution resources of the processors. By improving the functional unit capacities, performance can be achieved using the PDLO framework.

The performance of VOF-PDLO optimization for two input sets is shown in Figure 6.22. As a proof of concept of the policies used to gather persistent regions, it can be seen that the regions reformulated by the input sets are relatively similar in performance. This has significant meaning in understanding that invariant program behavior of some computation regions are likely to have reuse exploited by the CCR approach, while others are invariant and can be reformulated. Further study to evaluate these regions based on their reuse characteristics would yield interesting insight into different strategies of exploiting reuse opportunities.

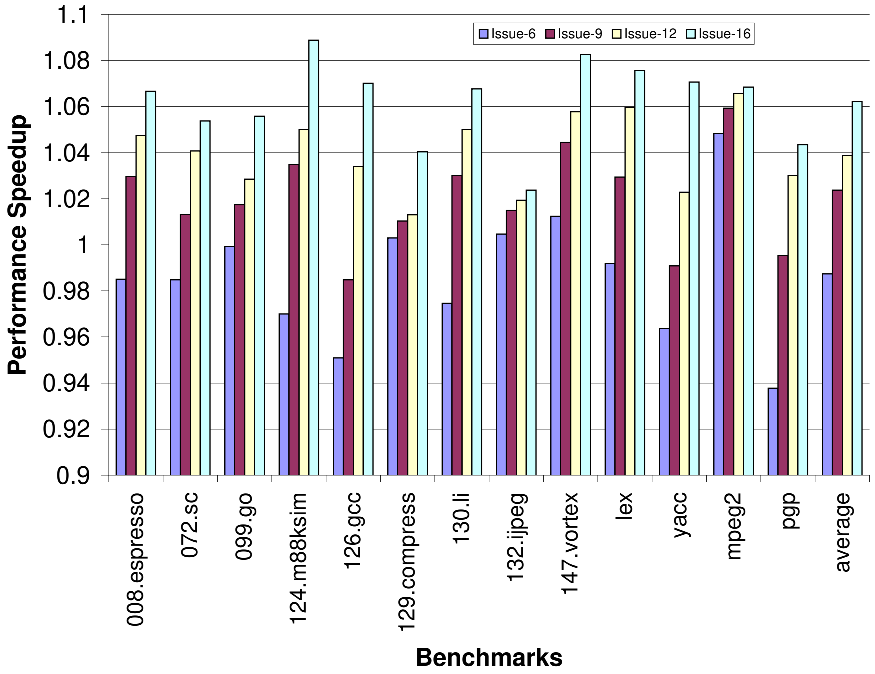


Figure 6.21 Evaluation of PDLO speedup for different machine models.

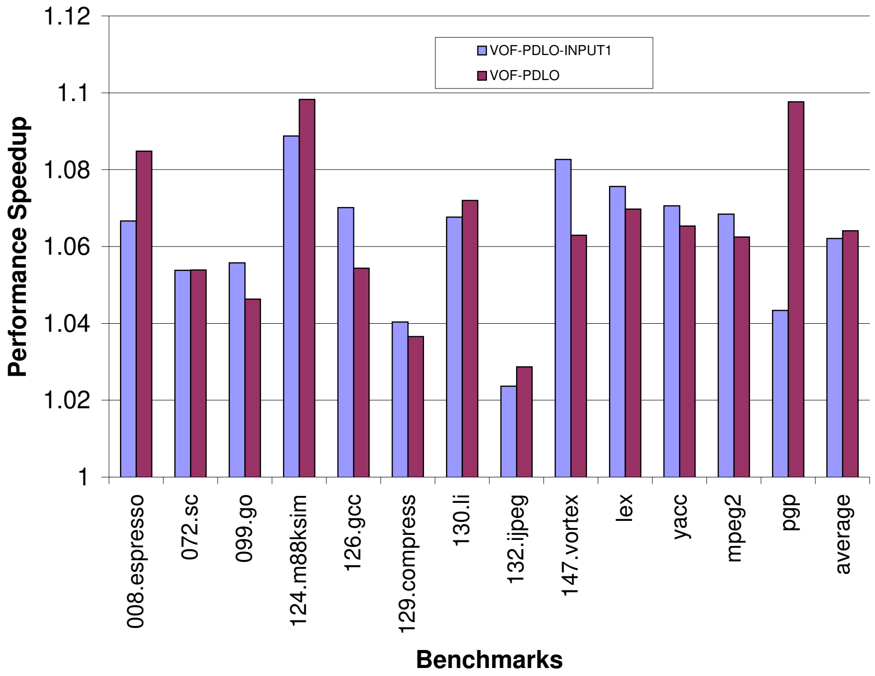


Figure 6.22 Evaluation of VOF-PDLO speedup for different input sets.

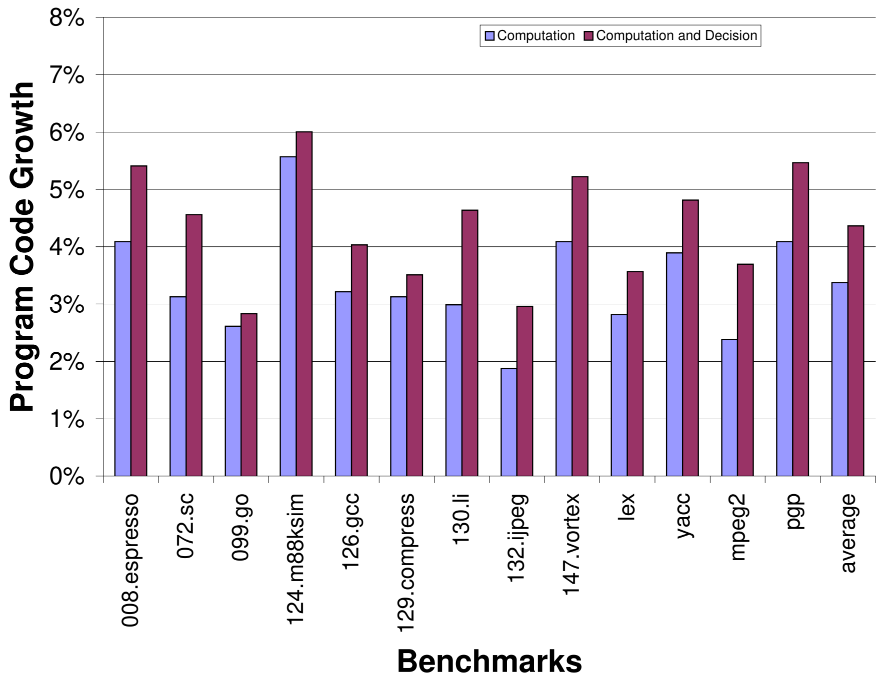


Figure 6.23 Code growth for applying value transforms.

Code growth is one side effect of the value-based transformations. Figure 6.23 illustrates the code growth due to VOF-PDLO, for the different components that compose the transformed code. Since each value context generated during the value vector generation phase represents a path of replicated instructions, there is code growth due to assigning the results of the region and the predicate decision instructions. The code growth is about 5% on average, which may be an acceptable level for the achieved performance of the reformulation technique. Future techniques may be able to selectively apply the transformation with code considerations as a factor in determining the level of transformation and whether to use reformulation more selectively.

The performance of partial reverse if-conversion of the escape sequences of the value-based transformation is shown in Figure 6.24 for the 16-issue processor model. In these cases the different levels of compiler parameters indicate the threshold attempting to create well balanced

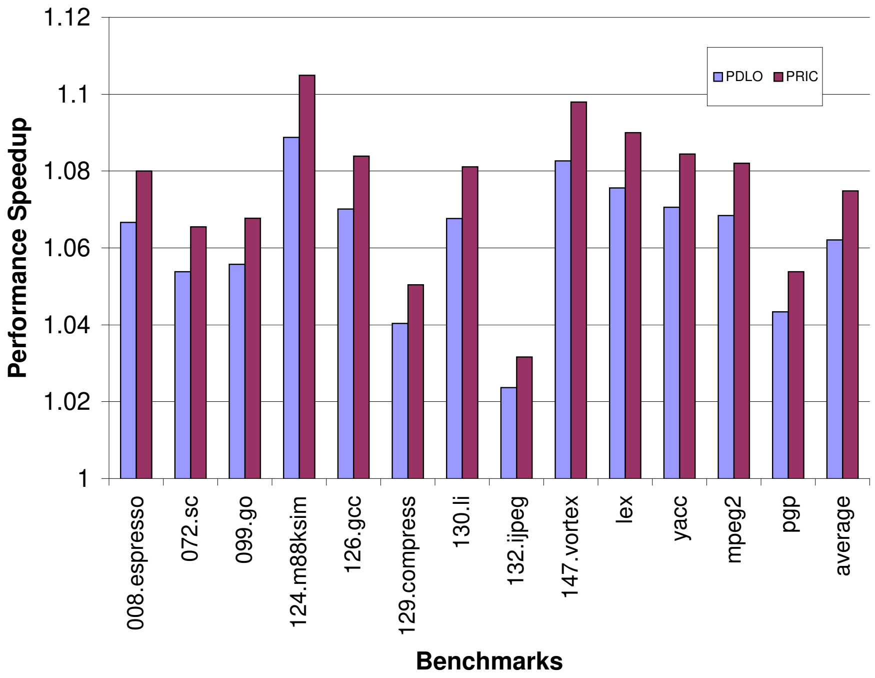


Figure 6.24 Performance for VOF-PDLO and VOF-PDLO-PRIC.

code migration between the escape sequence and the value sequence of the value-based codes. By allowing escape sequences to be partitioned with the value sequences, some degree of insurance is being performed since the value vectors institute the most likely paths. But more performance can be achieved by initiating the escape sequence in parallel for a wide machine such as the 16-issue model.

Coordinating Redundancy Elimination Techniques. By coordinating the redundancy elimination techniques proposed in this thesis, a cost effective design of the CRB can be achieved. Figure 6.25 shows the performance for three models of execution, a 32-entry CRB design, a 32-entry CRB design with compiler reformulation of vectored regions, and a 128-entry CRB design. Clearing by utilizing compiler support to reformulate the computation regions with invariance, then the CRB resources can be freed of some access contention. In the case of

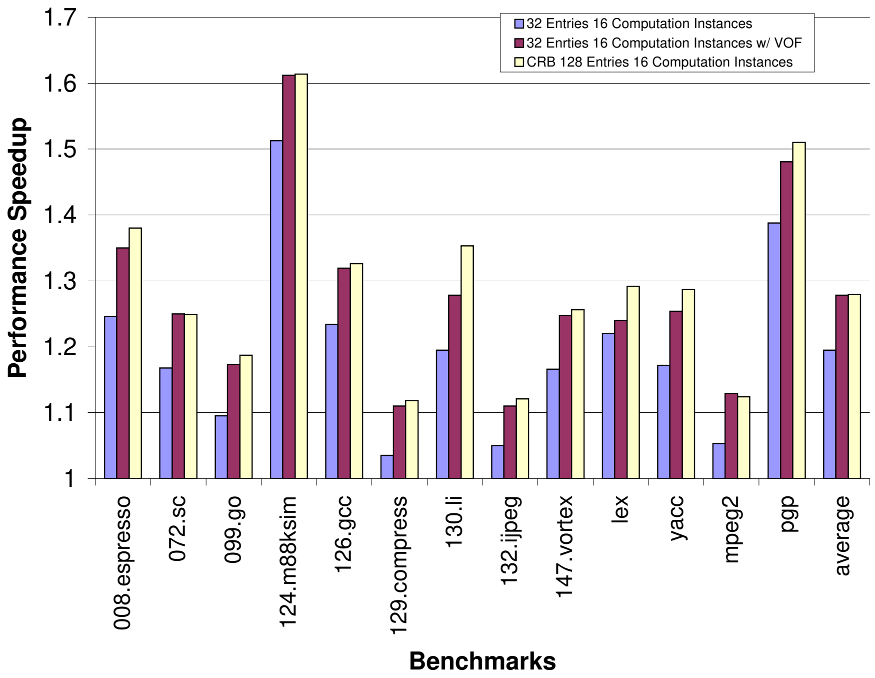


Figure 6.25 Performance comparing CCR models when coordinating compiler-based and compiler-directed techniques.

Figure 6.25, the coordination can amount to using a 32-entry computation buffer rather than a 128-entry buffer while achieving very similar performance.

ILP Paradigm Evaluation. One of the most important aspects of future compiler transformations is their ability to enable the cost effective design strategy of ILP machines. Optimizations that improve code efficiency by using many concurrent processor resources reduce the limitations of instruction parallelism naturally found in program codes. Figure 6.26 is used to motivate the application of silicon resources to the elimination of computation redundancy. Three models of execution are shown: base processor model/system with 32-entry CRB, base system model with perfect machine resources, and the base model with perfect system resources, and finally the base model with the CRB and perfect system resources. Generally the fundamental performance of applications can still be improved even for perfect system resources

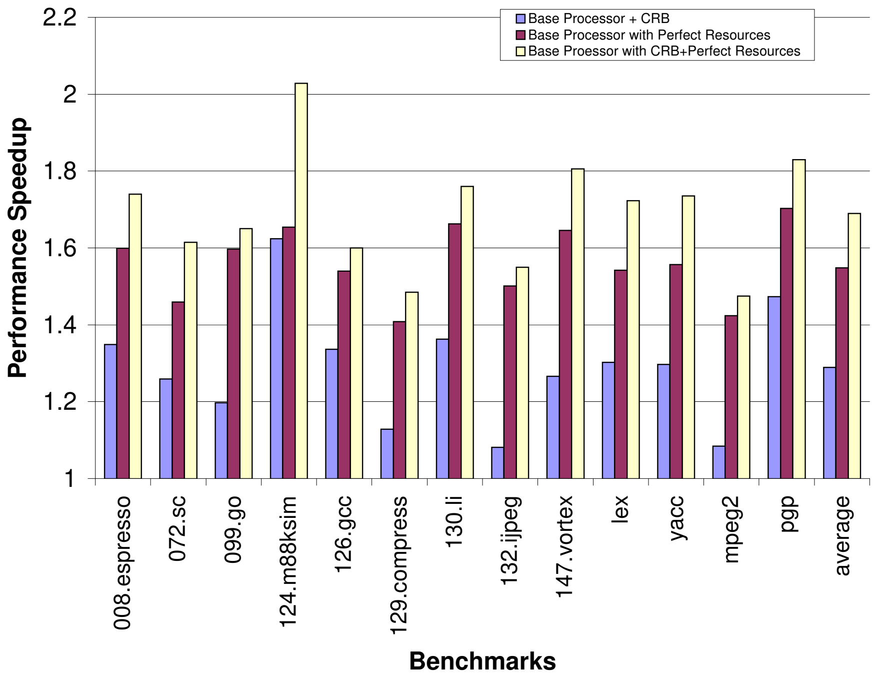


Figure 6.26 Performance comparing benefits of compiler-based and compiler-directed techniques when varying machine resources.

such as functional unit cache memory and branch prediction accuracy. In this case, computation reuse maintains some performance improvement over a machine with perfect ILP resources. This is due to the fact that both the ILP compiler and hardware technique do not properly exploit the value locality characteristics of code. As Figures 6.26 and 6.25 indicate, there are excellent opportunities to improve processor performance by eliminating dynamic computation redundancy.

6.6 Summary

In this chapter, new methods for transforming dataflow and memory flow expressions were investigated. The transforms of the VOF system effectively illustrate how value sequences could be created from the value profiling information. Subsequently, the paths could be represented using predicated execution and expressed in more parallel forms than the original codes. Although the predicated representation of the generated value-based sequenced was necessary for the value-based transformation, it is also a prefered method of exploitation. Through predicated execution, a series of advanced predicate transformation techniques can be applied to improve the efficiency of value-based transformations. The two techniques studied in this chapter are the predicate decision logic optimization and partial reverse if-conversion transformation. At the heart of the decision logic optimization technique is the existence of value locality and the opportunity to devise effient decision logic for the regions with computation instances that match closely in their value conditions. The optimization technique reduces the decision height of transformed sequences, allowing the predetermined computation results to be quickly introduced into the machine pipeline. The reverse if-conversion framework enables the predicate-based value transformed sequences to effectively balance the amount of speculation performed on escape sequence of value-synthesized codes. Even though predicated execution has been shown to be a powerful tool for enhancing ILP by reducing the detrimental effects of branching control flow, value was added to the predicated execution mechanism by illustrating that other spaces of opportunities exits.

Predicate-enabled program reformulation in conjunction with the compiler-directed computation reuse approach also showed interesting results. By coordinating actions of these techniques, improved behavior of the hardware resources of the CCR approach was observed. This improvement is enabled by eliminating invariant program behavior that is more efficiently handled using program reformulation techniques than hardware resources. The hardware is best constructed to exploit computation with dynamic redundancy whereas compiler-based techniques are more suited for regions with persistently invariant value execution.

CHAPTER 7

CONCLUSION

7.1 Summary

Compiler-directed computation reuse and compiler-based value transformations are novel approaches proposed in this thesis for exploiting program value locality. Effective exploitation of both technologies requires the compiler to understand the run-time invariance behavior within programs. This dissertation presented the Reusable Computation Region Framework, constructed within the IMPACT compiler, which provides a method for exploring the proposed techniques. Two compiler-directed hardware techniques, Compiler-directed Computation Reuse Approach and the Dynamic Computation Management System, provide significant advantages over previous methods of eliminating redundancy at the instruction level. Based on value execution behaviors, the compiler transformation techniques of the Value Optimization Framework reformulate the original program to highly-parallel instruction sequences that can be efficiently executed with predicated-execution support. The techniques enable cost effective performance on modern processors designed to exploit instruction-level parallelism.

The Compiler-directed Computation Reuse Approach uses the compiler to identify code regions whose computation can be reused during dynamic execution to eliminate dynamic redundancy. The instruction set architecture provides a natural and effective interface for the compiler to communicate the scope of each region to the hardware. During run time, the microarchitectural components of the approach record the execution results of the computation regions. Compiler-directed reuse has more potential to eliminate redundant computation by identifying regions in programs at compile time, which eliminates reuse detection at run time and implements efficient reuse. Several alternative hardware designs and optimization techniques were explored to determine which areas of technology offered the most promise in improving the performance and cost effectiveness of the CCR approach.

A major limitation of the compiler-directed approach is that the compiler relies on value profiling to identify reusable regions, making it difficult, if not impossible to deploy the scheme in many software production environments. The Dynamic Computation Management System is a novel hardware mechanism that alleviates the need for value profiling at compile time and enhances the assignment of computation regions in the hardware resources. The management system monitors the dynamic behavior of compiler-designated regions and selectively activates the profitable ones at run time. The proposed design makes more effective utilization of hardware buffer resources, achieves rapid employment of computation regions, and improves reuse accuracy, all of which promote more flexible compiler methods of identifying reusable computation regions.

By exploiting persistent value locality characteristics within programs, the transformations of the Value Optimization Framework enhance the original program code structure by synthesizing new optimized paths with value-specific contexts. The context-based sequences assert new dependence representations for the codes that overcome the fundamental dataflow limitations imposed by the original data and memory dependences. Predicated execution and predicate-based compilation techniques play an imperative role in the optimization framework as the newly created value contexts are translated to the predicate dataflow representation and efficiently executed on a wide-issue processor.

Eliminating dynamic computation redundancy has been presented as an important method of enhancing program execution speed and efficiency. The proposed compiler-directed and compiler-based techniques presented in this dissertation work in a coordinated manner to completely eliminate dynamic redundancy and fully exploit program recurrence. The computation redundancy of regions with high variation is captured by the hardware support in the CCR approach. However, since the compile-time extraction of computation regions cannot have absolute knowledge of the program execution conditions, the DCMS is required to make accurate run-time decisions about applying the computation reuse hardware resources. Code regions with persistent value locality are restructured using the VOF transformations rather than the hardware support of the CCR approach. Effective application of these technologies is enabled by the computation region formation framework that provides the identification of program regions with exploitable behaviors. The proposed coordinated compiler techniques and compiler-directed systems for eliminating dynamic computation redundancy allow a high degree of program efficiency to be achieved for a diverse range of applications.

7.2 Future Research

The work proposed in this dissertation promotes a number of future research directions in both compiler and architecture domains. The work still needs to be validated by a real hardware implementation and evaluation. Namely, the trade-offs between the cost of the instruction annotation overhead of the CCR approach and the evaluated benefits need significant investigation. In addition to hardware evaluations, alternative embodiments of the approach, such as eliminating the dependence of compiler-annotated region identifiers and compile-time alias analysis are the most likely future research directions.

The CCR approach operates by insuring that recorded results of designated regions of code are valid for reuse. However, the technique is generally limited in eliminating redundancy in the memory-dependent regions that access the same data values from different memory locations. It is often the case that two independent sequences of memory accesses result in the same computation results because the respective source memory values are identical. The CCR approach can be extended to further incorporate this form of memory redundancy by communicating each computation region's memory expression to the memory system. The memory system could evaluate the memory expressions and detect whether future memory sequences are redundant. Reuse availability information would be coordinated between the memory system and the processor when a computation region was initiated. Overall, the future approach would further integrate the compiler's role in the performance of advanced microarchitectures and high-performance systems.

As a mechanism for enhancing program execution, the compiler-directed approach has been thoroughly investigated. However, computation reuse offers other advantages such as the potential to reduce processor power consumption. Instructions that are bypassed using the proposed methods of this thesis are not fetched, do not execute in the microarchitecture pipeline, and do not access the memory system. As such, computation reuse has the potential to reduce the amount of power consumed. Further developments in system design include exploring concepts to achieve optimal solutions to the power/performance and cost/performance trade-offs for active computation regions.

The techniques of the VOF framework transform regions of code based upon the analyzed results of program analysis and value locality profiling. However, the techniques developed in the VOF could be applied to program-level reformulation. As future trends in software production environments continue, the responsibilities of the compiler to enable efficient code execution will continue to increase. Currently, optimizing compilers are only able to make existing programs operate more efficiently on processors, and fail to reformulate the program and transform the original algorithm to the most effective form. Many opportunities exist to identify problematic code structures with invariant execution behaviors, and the results presented for the VOF techniques motivate several promising opportunities for future work in systematic program reformulation. In addition, new programming environments that provide opportunities through the high-level language to further expose opportunities for reformulation are interesting directions of research created by the dynamic management system and are being studied.

Significant research in high-performance systems exists on using multiple threads to improve computing efficiency. These systems could benefit in many ways by expanding the work of compiler-directed computation reuse. First, in multithreaded systems, each thread operates on an individual task and later synchronizes results with one or more threads. It is often the case that separate threads have significant overlap in their working set and generate redundant computation sequences. The compiler-directed computation reuse approach could communicate shared computation instances for computation regions processed by different threads, enhancing performance by generating new thread-level efficiencies. For a subordinate-threaded system in which minor threads execute to optimize a main thread, the computation reuse buffer provides a likely system to relay interthread communication of execution results.

The DCMS model operates through the use of information collected at run time to improve program performance, specifically enhancing the efficiency of the CCR approach. As the inflexible ISA boundary separating software and hardware systems becomes more permeable with the construction of run-time systems, the fundamentals of the CCR approach will not be limited to only its compiler-directed embodiment. Run-time specific schemes of the CCR approach are possible through dynamic optimization systems that enable adaptive execution, dynamic compilation/translation, and continuous profiling/optimization. In such systems, the hardware passes information up to a layer of software. In turn the software uses that information to direct optimization and specialization, and employs global analysis and program knowledge to direct the new opportunities. There is a significant amount of future work in investigating dynamic systems that coordinate run-time program information with both the power of compiler transformation and compiler-directed hardware technologies.

REFERENCES

[1] R. Bodik, R. Gupta, and M. L. Soffa, "Complete removal of redundant computation," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pp. 1–14, June 1998.

[2] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 259–269, December 1997.

[3] A. Gonzalez, J. Tubella, and C. Molina, "Trace-level reuse," Tech. Rep. 47, Departament d' Arquitectura de Computadors, Universitat Politecnixa de Catalunya, October 1998.

[4] Y. Sazeides and J. E. Smith, "Modeling program predictability," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, June 1998.

[5] A. Sodani and G. S. Sohi, "An empirical analysis of instruction repetition," in *Proceedings of 8th International Conference on Architectual Support for Programming Languages and Operating Systems*, pp. 35–45, October 1998.

[6] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[7] D. B. Alpert and M. J. Flynn, "Performance trade-offs for microprocessor cache memories," in *IEEE Micro*, pp. 44–53, August 1988.

[8] T. Chen and J. Baer, "Reducing memory latency via non-blocking and prefetching caches," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 51–61, October 1992.

[9] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," in *Proceedings of 7th International Conference on Architectual Support for Programming Languages and Operating Systems*, pp. 138–147, September 1996.

[10] Y. Sazeides and J. E. Smith, "The predictability of data values," in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 248–258, December 1997.

[11] R. Johnson, "Efficient program analysis using dependence flow graphs," Ph.D. dissertation, Cornell University, Ithaca, NY, 1994.

[12] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers, "Annotation-directed run-time specialization in C," in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pp. 163–178, June 1997.

[13] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms.* Cambridge, MA: The MIT Press, 1986.

[14] D. A. Connors, "Memory profiling for directing data speculative optimizations and scheduling," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1997.

[15] T. Way and L. L. Pollock, "Using path spectra to direct function cloning," in *Proceedings of the 1998 Workshop on Profile and Feedback-directed Compilation*, October 1998.

[16] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 226–237, December 1996.

[17] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 194–205, June 1998.

[18] F. Gabbay and A. Mendelson, "Can program profiling support value prediction?" in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 270–280, December 1997.

[19] A. Sodani and G. S. Sohi, "Understanding the differences between value prediction and instruction reuse," in *Proceedings of the 31st International Symposium on Computer Architecture*, pp. 205–215, December 1998.

[20] J. Huang and D. J. Lilja, "Exploiting basic block value locality with block reuse," in *The 5th International Symposium on High-Performance Computer Architecture*, January 1999.

[21] S. P. Harbison, "An architectural alternative to optimizing compilers," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 57–65, March 1982.

[22] S. E. Richardson, "Caching function results: Faster arithmetic by avoiding unnecessary computation," Tech. Rep. 92-1, Sun Microsystems Laboratories, 2550 Garcia Avenue, Mountain VIew, CA 94304, September 1992.

[23] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers, "An evaluation of staged run-time optimizations in DyC," in *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pp. 293–304, May 1999.

[24] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad, "Fast, effective dynamic compilation," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, vol. 31, pp. 149–159, June 1996.

[25] C. Consel and F. Noel, "A general approach for run-time specialization and its application to c," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 145–156, January 1996.

[26] T. B. Knoblock and E. Ruf, "Data specialization," in *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, January 1996.

[27] T. Autrey and M. Wolfe, "Initial results for glacial variable analysis," in *Proceedings of the 9th International workshop on Languages and Compilers for Parallel Computing*, pp. 120–134, August 1996.

[28] M. Mock, M. Berryman, C. Chambers, and S. Eggers, "Calpa: A tool for automating dynamic compilation," in *Proceedings of the Second Workshop on Feedback-Directed Optimization*, pp. 100–109, November 1999.

[29] V. Bala, E. Duesterwald, and S. Banerjia, "Transparent dynamic optimization: The design and implementation of dynamo," Tech. Rep. HPL-1999-78, Hewlett-Packard Laboratories Cambridge, June 1999.

[30] K. Ebcioglu and E. R. Altman, "Daisy: Dynamic compilation for 100% architectural compatibility," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 26–37, June 1997.

[31] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24–34, December 1996.

[32] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Evaluation of design options for the trace cache fetch mechanism," *IEEE Transactions on Computers, Special Issue on Cache Memory and Related Problems*, February 1999.

[33] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 242–256, June 1994.

[34] B. C. Cheng, "Compile-time memory disambiguation for c programs," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 2000.

[35] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235–248, June 1992.

[36] D. M. Gallagher, "Memory disambiguation to facilitate instruction-level parallelism compilation," Ph.D. dissertation, University of Illinois, Urbana, IL, 1995.

[37] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 19th International Conference on Software Engineering*, pp. 213–224, May 1999.

[38] D. A. Connors and W. W. Hwu, "Compiler-directed computation reuse: Rationale and initial results," in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pp. 158–169, November 1999.

[39] 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.

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

[41] B. C. Cheng and W. W. Hwu, "Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation," in *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

[42] T. Autrey and M. Wolfe, "Initial results for glacial variable analysis," *International Journal of Parallel Programming*, vol. 26, February 1998.

[43] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," in *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, (New Orleans, LA), pp. 291–299, January 1985.

[44] W. H. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Transactions on Software Engineering*, vol. 3, May 1977.

[45] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," in *Proceedings of the ACM Symposium on Compiler Construction*, 1986.

[46] W. H. Chen, C. H. Smith, and S. Fralick, "A fast computational algorithm for the discrete cosine transform," *IEEE Transactions on Communications*, vol. COM-25, pp. 1004–1009, September 1977.

[47] T. Xanthopoulos and A. P. Chandrakasan, "A low-power IDCT macrocell for MPEG-2 MP@ML exploiting data distribution properties for minimal activity," *IEEE Journal of Solid-State Circuits*, vol. 34, pp. 693–703, may 1999.

[48] 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.

[49] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Value speculation scheduling for high performance processors," in *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 51–61, October 1998.

[50] A. Klauster, A. Paithankar, and D. Grunwald, "Selective eager execution on the PolyPath architecture," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 250–259, June 1998.

[51] A. Uht and V. Sindagi, "Disjoint eager execution: An optimal form of speculative execution," in *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 313–325, November 1995.

[52] D. C. Lee, P. J. Crowley, J. L. Baer, T. E. Anderson, and B. N. Bershad, "Execution characteristics of desktop applications on windows nt," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 27–38, June 1998.

[53] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization," in *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 136–147, May 1999.

[54] D. I. August, J. W. Sias, J. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu, "The program decision logic approach to predicated execution," in *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 208–219, May 1999.

[55] D. J. Kuck, *The Structure of Computers and Computations.* New York, NY: John Wiley and Sons, 1978.

[56] M. Schlansker and V. Kathail, "Acceleration of first and higher order recurrences on processors with instruction level parallelism," in *Proceedings of Languages and Compilers for Parallel Computing, 6th International Workshop*, August 1993.

[57] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[58] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

[59] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.

[60] D. I. August, W. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 92–103, December 1997.

[61] R. Bodik, R. Gupta, and M. L. Soffa, "Interprocedural conditional branch elimination," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pp. 146–158, June 1997.

[62] F. Mueller and D. B. Whalley, "Avoiding conditional branches by code replication," in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 55–66, June 1995.

[63] M. Yang, G.-R. Uh, and D. B. Whalley, "Improving performance by branch reordering," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 130–141, June 1998.

[64] M. Schlansker and V. Kathail, "Critical path reduction for scalar programs," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 57–69, December 1995.

[65] B. Calder, P. Feller, and A. Eustace, "Value profiling and optimization," *The Journal of Instruction-Level Parallelism*, vol. 1, March 1999. http://www.jilp.org/vol1.

[66] 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.

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

[68] K. M. Lepak and M. H. Lipasti, "On the value locality of store instructions," in *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 182–191, June 2000.

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

[70] A. Nicolau, "Run-time disambiguation: Coping with statically unpredictable dependencies," *IEEE Transactions on Computers*, vol. 38, pp. 663–678, May 1989.

[71] D. August, "Systematic compilation for predicated execution," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 2000.

[72] S. B. Akers, "Binary decision diagrams," *IEEE Transaction on Computers*, vol. C-27, pp. 509–516, June 1978.

[73] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transaction on Computers*, vol. C-35, pp. 677–691, August 1986.

[74] J. F. Wakerly, *Digital Design: Principles and Practices.* Englewood Cliffs, NJ: Prentice Hall, 1994.

[75] J. C. Gyllenhaal, "An efficient framework for performing execution-constraint-sensitive transformations that increase instruction-level parallelism," Ph.D. dissertation, University of Illinois, Urbana, IL, 1997.

VITA

Daniel Alexander Connors was born on October 16, 1972, in Chicago, Illinois. He pursued his undergraduate studies at Purdue University in West Lafayette, Indiana, where he received the Bachelor of Science degree in Electrical Engineering in May of 1994. After receiving the B.S. degree, he continued his studies in Electrical and Computer Engineering at the University of Illinois at Urbana, Illinois. During his graduate tenure, he was a member of the Center for Reliable and High-Performance Computing and the IMPACT project directed by Professor Wen-mei Hwu. He completed the M.S. degree in Electrical Engineering in 1997. After completing the Ph.D. work in 2000, he joined the faculty at the University of Colorado in Boulder, Colorado. At the University of Colorado, he holds an appointment with both the Department of Electrical and Computer Engineering and the Department of Computer Science. Currently he is developing an optimizing compiler and computer architecture research group to study high-performance and embedded systems.