Abstract

SATHAYE, SUMEDH WASUDEO

Evolutionary Compilation for

Object Code Compatibility and Performance.

(Under the direction of Prof. Thomas M. Conte)

Optimized code for high-performance systems can be generated using (1) the knowledge of processor microarchitecture, and, (2) the use of representative program profiles for profile-based optimization. Decisions made during compilation using such information may turn out to be erroneous when the code is run on a different machine model, or when program behavior changes due to a change in program inputs. The topic of this thesis, *Evolutionary Compilation*, is suggested as a solution. It helps the code adapt to such variations and thus preserves program performance.

The evolutionary compiler (EC) runs as a part of the operating system. Program behavior is monitored using unobtrusive, fast, hardware-based profiling techniques. A change in the program profile or a difference in the machine model assumed during compilation causes EC to be invoked on the object-code. One kind of variation in the machine model assumptions is the inter-generation compatibility problem in statically scheduled VLIW architectures. Dynamic Rescheduling (DR) is the technique used to reschedule code pages at first-time page faults to overcome this problem. Any changes in code size while using such a scheme are avoided by using the TINKER encoding. Caching techniques to reduce the overhead incurred due to rescheduling

are presented and studied.

In order to adapt to variations in profile information, recompilation requires more information than that available in the object code. Such information, e.g. live-outs across side-exits, is stored in the form of a non-loadable object-file annotation segment in the binary. The evolutionary compilation framework, the TINKER encoding, rescheduling algorithms and the properties of regions over which rescheduling is performed are discussed in detail.

Evolutionary Compilation for Object Code Compatibility and Performance

by

Sumedh Wasudeo Sathaye

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh

1998

Approved By:

Prof. Thomas M. Conte Chair of Advisory Committee	Prof. D. P. Agrawal
Prof. Paul D. Franzon	Prof. E. W. Davis

Biography

Sumedh Sathaye was born on August 25, 1968 in Mumbai (known previously as Bombay), India. Upon receiving a Bachelor of Technology degree in Computer Science from REC Warangal in 1989, he worked as an Associate System Engineer in the Research and Development arm of CMC Limited, for two years. In 1991, he began his graduate studies at the University of South Carolina, Columbia, and was awarded an MS in Computer Engineering in 1994. He continued his graduate education in the doctoral program in Computer Engineering at the NC State University, Raleigh. Sumedh joined the T. J. Watson Research Center of the International Business Machines Corporation in Yorktown Heights, NY, in February 1998.

Acknowledgments

First and foremost, I acknowledge the contribution of Prof. Thomas M. Conte, my thesis advisor, to this thesis. Evolutionary Compilation, as described in this thesis, is based on his vision. Tom has been a wonderful advisor and a friend. He has guided me very well through tedious parts of my graduate research career. I am fortunate to have worked under his direction.

I would like to thank my Dissertation Committee: Prof. Dharma P. Agrawal, Prof. Thomas M. Conte (Chair), Prof. Edward Davis, and Prof. Paul D. Franzon. Thanks also to Prof. Alexandre Eichenberger for agreeing to substitute on my Final at the last moment. Thanks to Prof. Agrawal for his guidance, his patience, and understanding.

I would like to thank Dr. B. Ramakrishna Rau and Dr. Michael Schlansker for their words of advice during the summer of 1995.

Thanks to fellow TINKER group members: Kishore Menezes, Sanjeev Banerjia, Emre Ozer, Matt Jennings, Roger Isaac, Bill Havanki, Sergei Larin, Mary Ann Hirsch and Willie Glover. Bill did a lot of LEGO compiler development. Sanjeev made substantial contributions to the schedulers in LEGO.

Thanks to June Phillips for helping me wade my way through the myriad paper-

work in graduate school.

Many results presented in this thesis were generated with help from the IMPACT compiler developed at the University of Illinois, Urbana, by the group led by Prof. Wen-mei W. Hwu. I acknowledge help from the IMPACT group– special thanks to Scott Mahlke and John Gyllenhaal for helping me out with numerous questions.

I was blessed with many good friends at various times in my life. I would like to thank them for all the great times and all that I have learnt from them. (In no particular order): A. Kameshwar Rao, Manoj Kalra, Donald Lobo, Diganto Gogoi, and Janak Patel for the fun times in REC Warangal; Kalpesh Unadkat for some crazy night-outs; Catherine Linder Conte and Tom Conte for Sunday afternoon cook-outs on their patio; Nitin Deshmukh, Manish Vichare, Harshad Bakshi, Chetan Bhat, Swati Deval, Parag Shukla, Ziauddin Sayed, and Dr. Sharad Wagle and other fellow CommGroup members for the wonderful times while I was with CMC Communication Group in Bombay; Troy Travis, Srinadh Godavarthy, Sadhana Karandikar, Ashish Patel, Raju and Radha Marathe; Kishore Menezes, Sanjeev Banerjia and Matt Jennings at NCSU.

I would like to thank my father Adv. W. S. Sathaye and my mother Mrs. Shubhada Sathaye for their patience, understanding, and support. Thanks also to my sister Seema and brother-in-law Sanjay for their love.

Last, but not the least, I would like to express my gratitude towards my wife Swati. She has been a wonderful and cheerful partner throughout the three grueling

years of my Ph.D. I really am fortunate to have found her love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern high-performance architectures depend heavily upon the compiler to extract instruction-level parallelism in common scalar programs. In these programs, available parallelism is typically not explicit in the source code. Rather, the compiler must use aids such as the program *profile* to infer the nature of parallelism that may be available in the code. For example, the profile information is used to detect independent flows of control through the code, independent memory access instructions, locality of memory accesses, and the like [1] [2]. This information is later used in program optimization.

The profile of a program is an imprint of its dynamic behavior, and is generated using typical, representative inputs to the program. (A detailed description of the techniques to extract profile of a program can be found in [1] and [3]). The technique of program optimization and instruction scheduling which uses profile information has been referred to as profile-based optimization [4], [5], [6], [7], or PBO. The main

idea behind PBO is to spend maximum effort optimizing the most frequently executed code. Examples of optimizations which aggressively use profile information are classic code optimizations [6], instruction scheduling [8], [9], [10], [11], [12], [13], code layout for improved ICache performance [7], [14], and speculative execution [15], [16]. In the past, the VLIW architectures such as the MultiFlow TRACE [17], [18] and the Cydrome Cydra 5 [19], [20] have used the PBO techniques extensively. An example of a recent superscalar for which the use of PBO has been described is the Hewlett-Packard PA8000 processor [21], [22].

During profile-based optimization, the profile is assumed to be a fair characterization of the program behavior. The object-code generated as a result of PBO is well-tailored for execution of the program on the input originally used to collect the profile information. The usage pattern of an arbitrary program, however, differs vastly from user to user, and hence the "sample, representative" input which forms the basis of these optimizations may not be representative of all the cases. This observation points to a deep (and often ignored) flaw of the PBO technique: how would the program perform if run on an alternative input which differs substantially from the one which was used to optimize the code? It is only natural to expect that the performance would degrade when compared to the program tailored for the alternative input. In the extreme, if the input chosen for PBO is not representative of the *most common* of the inputs to the program, a degradation in performance will be evident almost every time the program runs. Inability of an arbitrary input to capture

the "representativeness" of the most common inputs of a program is the principal shortcoming of the profile-based optimization technique.

Another shortcoming of the technique is its dependence on the machine microarchitecture for which the instruction stream is scheduled. Even when the input to the program remains the same, any differences in the microarchitecture could lead to a drop in performance. For example, consider the case of Pentium Pro and the Pentium, the two latest processor offerings from Intel. Because the microarchitectures of Pentium Pro and its predecessor Pentium are substantially different (Pentium Pro is an out-of-order superscalar, whereas the Pentium is an in-order superscalar), an old binary optimized for execution on the Pentium cannot take advantage of the architectural advances in Pentium Pro. A program binary scheduled for the Pentium microarchitecture could perform worse on the Pentium Pro microarchitecture, if the semiconductor implementation technology differences between the two are reconciled for purposes of the comparison. When the instruction scheduler within the compiler targets a *microarchitecture* (as opposed to targeting an ISA), it uses a combination of the profile information and the machine model to prioritize the *ready* instructions to be scheduled. The code would perform as expected only so long as the machine model assumptions are precisely met. When *both* the profile and the machine model assumptions change, their impact on the performance is usually detrimental. As another example, when instructions are scheduled for a wide-issue machine, the scheduler attempts to take advantage of the issue width by aggressively speculating the

operations above branches. If the program behavior changed due to changes in the input *and* the code is executed on a narrow-issue machine, not only would the speculated operations produce largely useless computation, but also take away valuable resources (such as issue slots) from critical operations. The critical operations would be delayed unnecessarily, resulting in performance degradation.

Correctness of code generated for statically scheduled machines such as VLIWs is another important issue that arises when a change in machine model assumptions occurs. The VLIWs depend on the compiler to present a stream of instructions so crafted that the correctness of computation is guaranteed without any runtime checking hardware. Due to this extreme dependence on static scheduling, code scheduled for one VLIW generation *cannot be guaranteed* to execute correctly on another generation. This is a serious problem because it leads to binary inter-generation incompatibility in VLIW architectures. The implications of the problem are enormous: it is a principle reason why VLIWs have not been accepted as a general-purpose computing paradigm by the marketplace, in spite of their simplicity and performance potential [23]. An effective solution which sacrifices little performance yet maintains binary compatibility in VLIWs is essential to resolve this problem.

The topic of this thesis, *Evolutionary Compilation*, addresses these shortcomings of profile-based optimization. Evolutionary Compilation transforms the program object-code to enhance the performance when profile shifts occur, or to guarantee compatibility when code incompatibility is detected. The overhead of evolutionary

compilation is amortized over multiple runs of a program using caching techniques. The evolutionary compilation framework is described in detail in the next section (Section 1.1).

1.1 Evolutionary Compilation

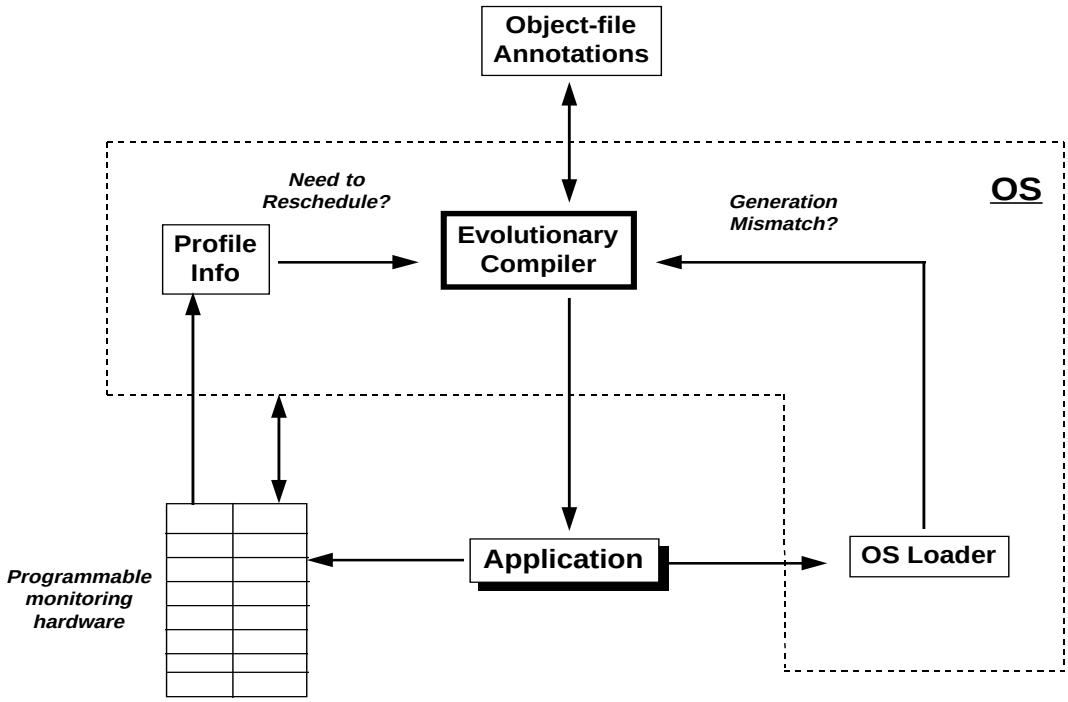


Figure 1.1: The Evolutionary Compilation framework.

The proposed Evolutionary Compilation framework is shown in Figure 1.1. The framework is designed to be a component of the OS. At the heart of the framework is the evolutionary compiler (EC). The EC performs the task of object-code transformations based on the evolution *stimuli* generated during the execution of an application.

The two stimuli shown here are (1) detection of a machine generation incompatibility in the code, and, (2) a shift in performance of the code across inputs. The incompatibility stimulus is generated by the virtual memory subsystem OS, which loads program code pages on demand. It compares the code type against the generation of the machine for this purpose, and invokes the EC to perform rescheduling of the page if needed. The performance shift stimulus is generated using the programmable monitoring hardware for hardware-based profiling. Dedicated hardware support is assumed in the processor microarchitecture for unobtrusive, fast, and effective collection of profile information[1]. Based on past observations or using the "hints" provided by the initial compiler, the EC programs the profiling hardware to monitor specific events during execution. For example, the hardware may be programmed to monitor and count occurrences of *aliasing* between a pair of `store` and a following `load` instructions. Similarly, the taken-frequency of a branch marked by the initial compiler to be *volatile* may be counted. Using the information thus collected, the EC invokes a decision making oracle to decide if the profile changes are significant to warrant a rescheduling transformation.

If rescheduling is warranted, the EC operates on *regions* of object-code (region boundaries are marked in the ISA encoding). Since the EC operates on object-code, it lacks the semantic information about the program that would otherwise be available

[1]Study of hardware-based profiling techniques has been documented elsewhere [1] [2], and will not be discussed in this dissertation. For the purposes of this dissertation, it is assumed that the necessary profile information is available to the evolutionary compilation framework, on demand, from the hardware-based profiling modules in the machine hardware and supporting modules in the OS.

to the initial compiler. For example, set of variables *live* across a side-exit, or the data- and control-dependence graph of the region code could be valuable for the transformations of object-code. Lack of this information can significantly restrict the abilities of the evolutionary compilation framework. To remedy this, a significant amount of information about the object-code is saved during initial compile. This information is stored along with the executable in a non-loadable segment called object-code annotations segment.

Overhead of the evolutionary transformations is a concern, because if the overhead is high, the gains of object-code evolution will be lost in paying for them. To address this concern, transformed sections of code are cached for future use, thus amortizing the overhead over the entire run of the program or multiple runs of the program.

1.2 Research contributions

This thesis presents and evaluates the technique of Evolutionary Compilation for (1) object-code compatibility in VLIW architectures and, (2) for performance improvement of object-code in the presence of profile shifts. The following are the research contributions of this thesis. The problem of inter-generation object-code compatibility in statically scheduled VLIW architectures is examined, and a scheme called *Dynamic Rescheduling* is proposed. Dynamic Rescheduling performs object-code rescheduling of program code pages at *first-time* page faults. The rescheduler is invoked when a page page with generation mismatch is presented for execution. Prob-

lems associated with dynamic rescheduling, *viz.* speculative-code motion, code-size change, and code correctness issues with live-variables along side-exits, are examined and solutions are presented. Dynamic rescheduling is evaluated for different generations of the TINKER machine, for various scalar programs. It is shown that the technique suffers from the drawback of overhead of rescheduling. To reduce the impact of this overhead, a rescheduled-page caching technique called a *Persistent Reschedule-Page Cache (PRC)* is introduced. Various organizations and page replacement schemes for the PRC are evaluated empirically.

Solution to the problem of code-size changes during object-code rescheduling is based on a specialized encoding for the object-code, called the TINKER encoding. Formal definition of the encoding is presented, and the properties of rescheduling size-invariance exhibited by the encoding are introduced.

The other focus of this thesis is evolutionary compilation of object-code to improve performance in the presence of profile shifts. It is shown that substantial shifts in performance occur for scalar programs across inputs which have different nature. The performance shift data is presented for superblock code and treegionized code, for several machine models. This data is used to make a case for evolutionary compilation for performance.

Object-code evolution techniques to achieve performance in the presence of profile shifts are outlined. Methods to detect the performance shift, and an *oracle* to make the rescheduling decision are presented. Algorithms for object-code rescheduling are

also described.

Evolutionary compilation techniques operate on the object-code representation of the program. The information that the original compiler has when it parses the program in the form of source code written in a high-level language is lost during translation. Lack of this information can impact the effectiveness of evolutionary compilation. Proposal to save some of this information in form of a non-loadable program object-annotation segments is made. Detailed description of various pieces of information which can be useful in improving the performance of the program and/or the evolutionary compiler is presented. Space requirements for the object-annotations are studied and evaluated.

1.3 Outline of the thesis

The organization of this thesis is as follows. Chapter 2 presents the technique of Dynamic Rescheduling for inter-generation compatibility in VLIW architectures. Chapter 3 provides empirical evaluation of the dynamic rescheduling scheme. The problem of overhead due to rescheduling is discussed, and a solution is presented which is called a Persistent Rescheduled-Page Cache. Chapter 4 formally introduces the TINKER object-code encoding, and its properties of size-invariance. Chapter 5 presents an empirical study of the shift in performance of ordinary scalar programs induced by shifts in program behavior with varying inputs, known as the profile shift. Chapter 6 discusses the application of evolutionary compilation for improving program perfor-

mance in the wake of profile shifts. Chapter 7 presents the object-code annotations that are saved during initial compile of the program, and used during object-code evolution. Chapter 8 presents conclusions of this thesis and discusses a few possible directions for future research.

Chapter 2

Evolutionary Compilation for Object-Code Compatibility in VLIW architectures

Lack of object-code compatibility across generations of a VLIW architecture is an often raised objection to its use as a general-purpose computing paradigm [23]. A program binary compiled for VLIW generation x cannot be guaranteed to execute correctly on generations $x+n$ or $x-n$, for a reasonable value of n. This means that an installed software base of binaries cannot be built around a family of VLIW generations. The economic implications of this problem are enormous, and an efficient solution is necessary if VLIW architectures are to succeed. Two classes of approaches to this problem have been reported in the literature: hardware approaches and soft-

ware approaches. The hardware approaches include *split-issue* proposed by Rau [24], and the *fill-unit* proposed by Melvin, Shebenow, and Patt [25] and later extended by Franklin and Smotherman [26]. Although these techniques provide compatibility, they do so at the expense of additional hardware complexity that can potentially impact cycle time. A typical software approach is to statically recompile the VLIW program from the source code, or from the object-code. This approach requires generation of multiple executables, which poses difficulties for commercial copy protection and system administration. This chapter proposes a new scheme called *Dynamic Rescheduling* to achieve object-code compatibility between VLIW generations. Dynamic rescheduling applies a limited version of software scheduling during first-time page-faults, requiring no additional hardware support. Making this practical requires support from the compiler, the ISA, and the operating system. These topics are discussed in detail below. Results which suggest that dynamic rescheduling has potential to effectively solve the compatibility problem in VLIW architectures are presented in Chapter 3.

2.1 The VLIW compatibility problem

The compatibility problem is illustrated in the following example. Figure 2.1 shows an example VLIW schedule for a machine with two integer ALUs, and one unit each of Multiply, Load, and Store. The latencies of the units are as shown. Assume that this represents first generation of the machine. Figure 2.2 shows the next-generation

IALU			IALU			MUL			LD		ST	
1 cycle latency			1 cycle latency			3 cycle latency			2 cycle latency		1 cycle latency	
d	*s*	*s*	*d*	*s*	*s*	*d*	*s*	*s*	*d*	*addr*	*addr*	*s*
A:R1	R2	R3				D:R6	R2	R3	B:R4	X		
						E:R7	R1	R3				
C:R5	R4	R3	G:R8	R4	R3							
H:R9	R6	R4										
											F: X	R7

Figure 2.1: Scheduled code for the original VLIW machine.

VLIW where the Multiply and Load latencies have changed to 4 and 3 cycles respectively. The old schedule cannot be guaranteed to execute correctly on this machine due to the flow dependence between operations B and C, between D and H, and between E and F.

Figure 2.3 shows the schedule for the next-next-generation machine that includes an additional multiplier. The latencies of all FU*s* remain as shown in Figure 2.1. Code scheduled for this new machine would not execute correctly on the older machines because the scheduler has moved operations in order to take advantage of the additional multiplier. (In particular, operations E and F have been moved.) There is no trivial way to adapt this schedule to the older machines. This is the case of downward incompatibility between generations. In this situation, if different generations of machines share binaries (e.g., via a file server), compatibility requires either

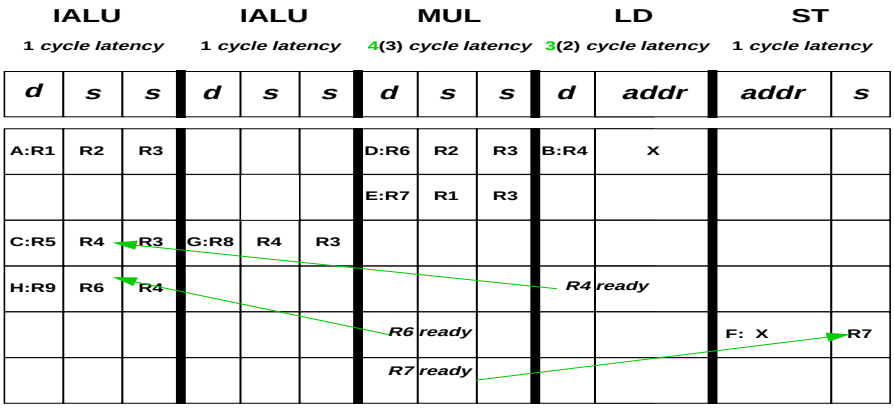


Figure 2.2: Next generation VLIW machine: Incompatibility due to changes in functional unit latencies (shown by arrows). The old latencies are shown in parentheses. Operations C, H, and, F are now produce incorrect results because of the new latencies for operations B, D, and E.

a mechanism to adjust the schedule or a different set of binaries for each generation.

A scheme which would guarantee correct execution of a VLIW binary on any generation of the machine would suffice to solve the compatibility problem. Such a solution must be efficient in order to be viable. Also, it must be implemented from the very first generation to ensure upward compatibility with future generations.

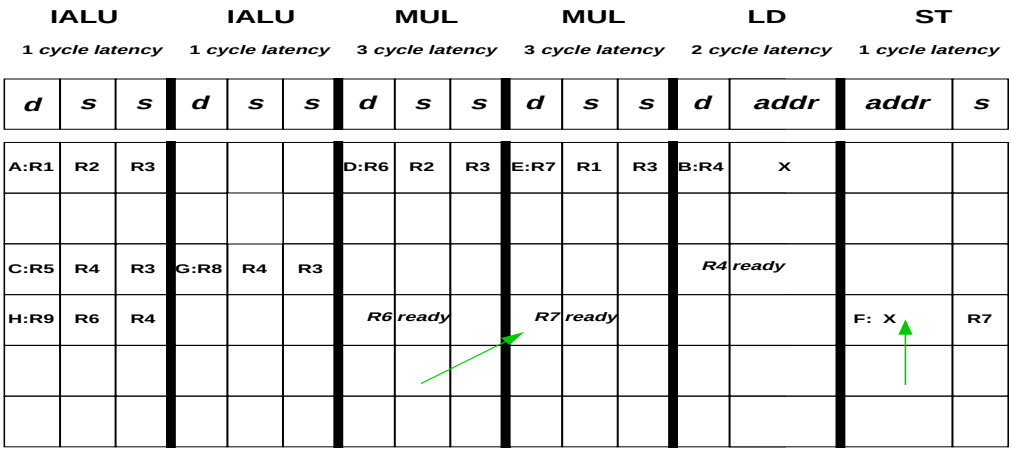


Figure 2.3: Downward incompatibility due to change in the VLIW machine organization: no trivial way to translate new schedule to older machine.

2.2 Related Work

2.2.1 Terminology

The following terminology is from Rau [24], and will be used in this chapter and the rest of this thesis. VLIW architectures are horizontal machines, with each wide instruction-word, or *MultiOp*, consisting of several operations, or *Ops*. All Ops in a MultiOp are issued in the same execution cycle. VLIW programs are latency-cognizant, meaning that they are scheduled with knowledge of the functional unit latencies. A VLIW architecture which runs latency-cognizant programs is termed a *Non-Unit Assumed Latency* (NUAL) architecture. non-unit latency. A *Unit Assumed Latency* (UAL) architecture assumes unit latencies for all functional units. Many

superscalar architectures are UAL.

There are two scheduling models for latency-cognizant programs: the *Equals* model and the Less-Than-or-Equals (LTE) model [24]. An *Equals* model schedules for a VLIW architecture on which each operation takes exactly its specified execution latency. In contrast, an LTE model schedules assuming any operation may take less than or equal to its specified latency. The Equals model produces slightly shorter schedules than the LTE model, mainly due to register reuse. However, the LTE model simplifies the implementation of precise interrupts and provides binary compatibility when latencies are reduced. The scheduler in the back-end of the compiler and the dynamic rescheduler presented in this chapter follow the LTE scheduling model.

2.2.2 Previous work

The working principle behind hardware techniques used to support object-code compatibility in VLIW machines is shown in Figure 2.4. It is similar to superscalars in that both perform run-time scheduling in hardware. The difference, however, is that the schedule presented to superscalar dynamic scheduling hardware is UAL, whereas the scheduling hardware in a dynamically scheduled VLIW processor is presented with a NUAL schedule.

Rau [24] presented a hardware technique, called Split-Issue, for dynamic scheduling in VLIW processors. In order to handle NUAL programs, it provides hardware capable of splitting each Op into an Op-pair: $(read\_and\_execute, destination\_writeback)$.

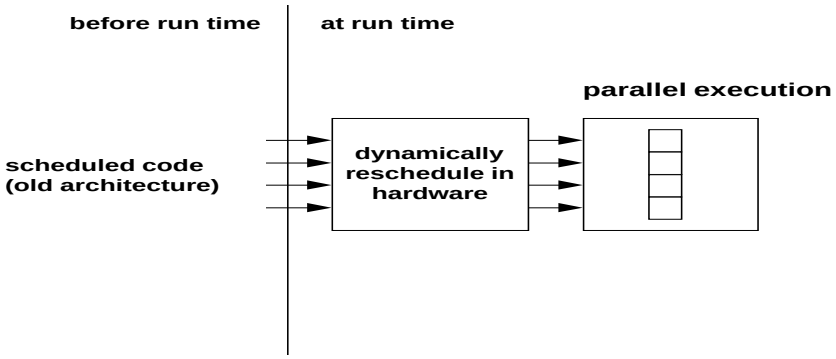


Figure 2.4: Hardware approach to compatibility.

Read_and_execute uses an anonymous (i.e. a non-architected) register as its destination, whereas *destination_writeback* copies the destination of *read_and_execute* to the destination specified in the original Op. *Read_and_execute* operation is issued in the next available cycle, provided there are no dependence or resource constraints. The *destination_writeback* operation is scheduled to be issued in the latest cycle after ($issue\_cycle$ ($read\_and\_execute$) + $original\_operation\_latency - 1$). To ensure that the *destination_writeback* operation is not issued before the *read_and_execute* completes, support in the form of hardware flags is provided. The splitting of operations and issuing them in the correct time order preserves the program semantics, and correct program execution is guaranteed.

The concept of fill-unit was originally proposed by Melvin, Shebanow, and Patt in [25], and was extended in [26]. Although it was originally not aimed at achiev-

ing compatibility in VLIWs, it can be adapted to fulfill this goal. Special hardware consisting of the fill-unit and a shadow cache is used in this technique. It works as follows: the processor routinely executes a UAL program operation stream. Concurrent to the execution, the fill-unit compacts these operations into VLIW-like MultiOps. These newly formed MultiOps are stored in the shadow cache. When an operation requested by the fetch unit is available in the shadow cache, all the operations in the MultiOp containing this operation are issued. The formation of a new MultiOp by the fill-unit is terminated when a branch instruction is encountered.

A limitation of the hardware approaches is that the scope for scheduling is limited to the window of Ops seen at run-time, hence available ILP is relatively less than what can be exploited by a compiler. These schemes also may result in cycle time stretch, a phenomenon due to which many are considering the VLIW paradigm over superscalar for future generation machines.

Static recompilation is the most prevalent software technique (illustrated in Figure 2.5). It recompiles the entire program off-line, and hence can take advantage of sophisticated compiler optimizations to attain superior performance. Alternatively, complete recompilation of the program may be avoided by maintaining multiple copies of the program for various target architectures in a partitioned object file. An appropriate module can be scheduled at installation time. The main drawback of these methods is that they involve an extra step to achieve code compatibility. This introduces a deviation from the normal development process for the developer, and from

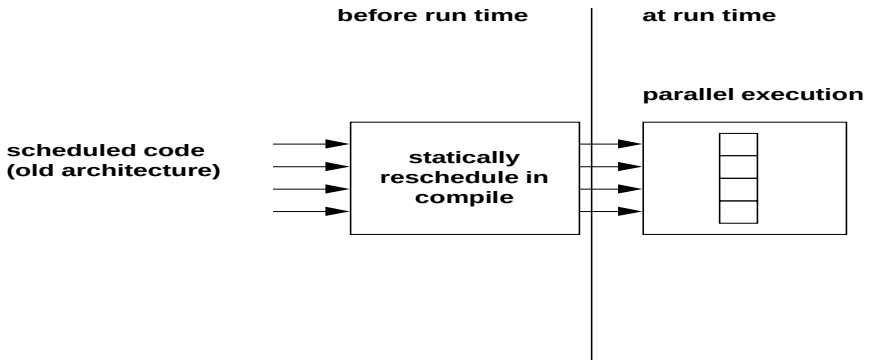


Figure 2.5: Rescheduling the program off-line for compatibility.

the routine installation process for the user. Also related is the issue of potential copy protection violations. Software licensing is done on a per-copy basis; having multiple specialized copies of the same program, even though the user plans to use only the one for his machine, may well become an expensive proposition. Another problem is that the storage space requirements of multiple copies may be excessive. These problems suggest that compatibility via off-line recompilation may not be easy to commercialize.

Of related interest are the techniques used to migrate the software across to a new machine architecture. Silberman and Ebcioglu [27] describe in detail an effort to gain performance advantage by translating and running old CISC object code on RISC, Superscalar and VLIW machines. *Binary Translation* used by Digital Equipment Corp. to migrate from the VAX/VMS environment to its newer Alpha architecture is

documented in [28]. Their latest effort is called FX!32 [29], which uses a combination of translation and emulation, and caches frequently executed pieces of code for future use. Apple Computers used the technique of *emulation* to migrate the software compiled for Motorola 680x0 to the PowerPC. Insignia Solutions has presented the effort of emulating Intel x86 architectures on modern RISC machines in [30]. A short survey of related techniques and issues can be found in [31]. Researchers from IBM recently described a project called DAISY [32], which attempts to provide 100% architectural compatibility, between generations of the same ISA, or across heterogeneous ISAs.

2.3 Dynamic Rescheduling

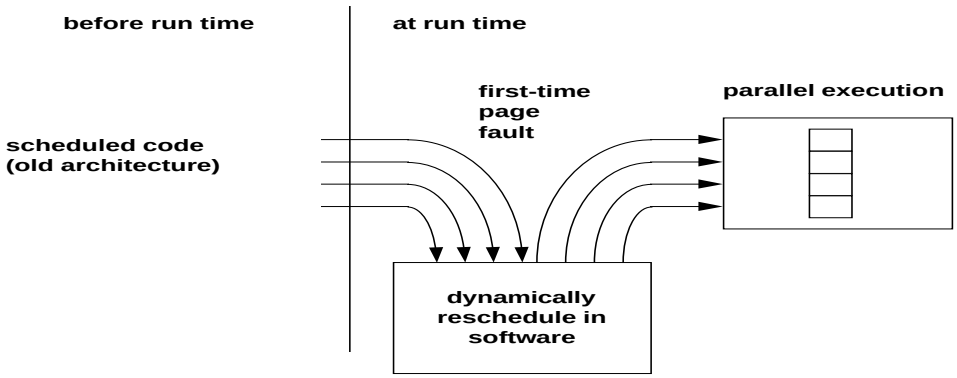


Figure 2.6: Dynamic Rescheduling.

Dynamic rescheduling is illustrated in Figure 2.6. When a program is executed on a machine generation other than what it was scheduled for, the dynamic rescheduler is invoked. The exact sequence of events is as follows: the OS loader reads the program binary header and detects the generation mismatch.[1] After the first page of the program is loaded for execution, the page fault handler invokes the dynamic rescheduler module. The rescheduler reschedules the page for execution on the current host. This process is repeated each time a new page fault occurs. Translated pages are saved to swap space on replacement. Only the pages which are executed during the life-span of the program are rescheduled. The knowledge of architectural details of the executable's VLIW generation is necessary for the dynamic rescheduler to operate, and is retained in the executable image.

Dynamic rescheduling poses some interesting problems which can reduce its effectiveness as a run-time technique. The rest of this section discusses these problems in detail and presents solutions. This thesis assumes that the code scheduled for a VLIW machine is logically organized as a sequence of scheduling structures called superblocks [11], hyperblocks [33], or treegions [34] [35]. Superblocks, hyperblocks, and treegions belong to a category of scheduling structures called SEARs (Single-entry Acyclic Regions [36]). Construction of superblocks and hyperblocks is shown in Figures 2.7 and 2.8, respectively; Treegions will be discussed at length later in Chapter 5. The implementation of the dynamic rescheduling algorithm uses the TINKER

[1] The version information is retained as part of the process table entry for the process.

microarchitecture [37]. TINKER is based on the parametric PlayDoh VLIW architecture from Hewlett-Packard Laboratories [38]. Some of the features in TINKER have been designed specifically to solve the problems faced in dynamic rescheduling. For example, the TINKER bit-encoding for MultiOps provides a *Block-Bit* in an Op to mark a entry point (merge point) of a SEAR. This information is used by the rescheduler to define the scope of rescheduling. More examples are presented in later sections of this chapter and in Chapter 7. the chapter.

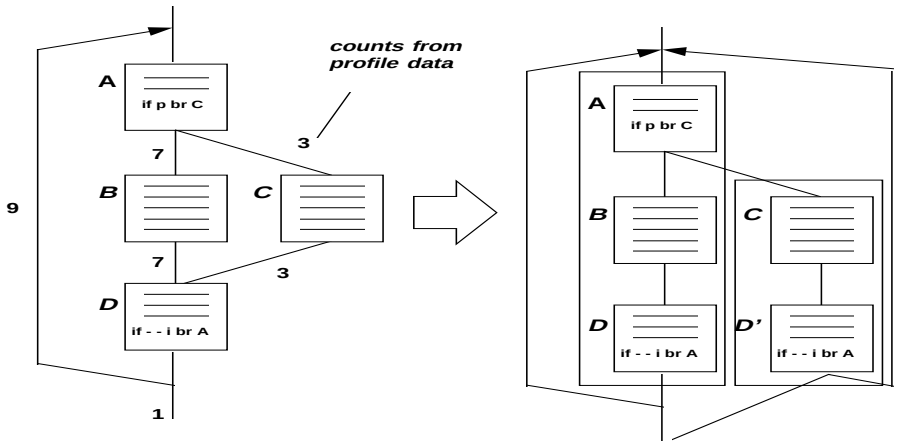


Figure 2.7: Example of Superblock formation. Note that both Superblocks and Hyperblocks have a single entry, multiple exits and no side entrances. In general, Hyperblocks provide larger scope for ILP than the Superblocks.

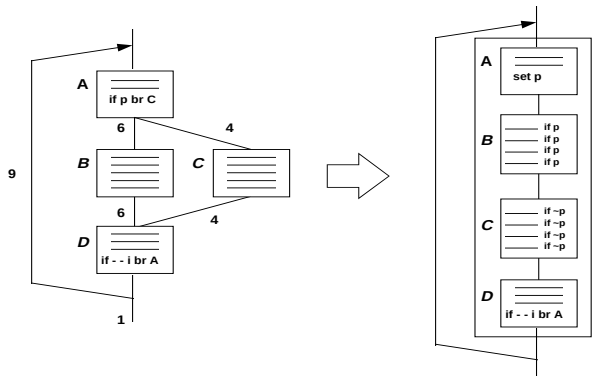


Figure 2.8: Example of Hyperblock formation.

2.4 Problems and their solutions

2.4.1 Changes in Code size

The dynamic rescheduling algorithm constructs a new schedule from the old schedule, using the knowledge of the old and new machine organizations and the execution latencies of the functional units. The new schedule may grow larger in size due to the insertion of empty cycles, or may shrink in size due to the deletion of empty cycles from the old schedule. This insertion or deletion of empty cycles introduces variation in code size. This phenomenon is illustrated in Figure 2.9, which assumes two simple machines each having integer ALU (IALU), FP add, FP multiply (FPMul), load, store, branch, and predicate comparison (Cmpp) units. Further, each Op is assumed to be 8-bytes wide. The number of `nop`s in the upper left sample schedule of Figure 2.9 is 24. After the code is rescheduled for a machine having one less IALU and increased

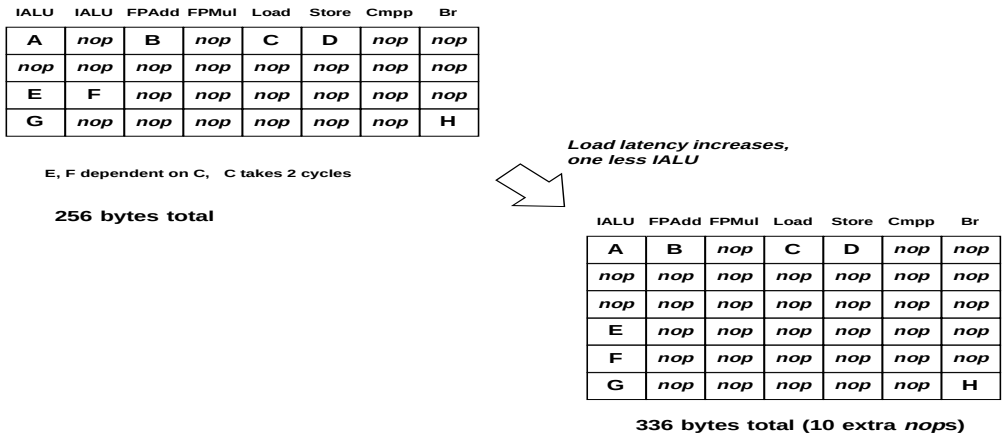


Figure 2.9: Example illustrating the insertion/deletion of NOPs and empty MultiOps due to Dynamic Rescheduling.

Load latency, the number of `nop`s in the rescheduled code becomes 34, resulting in a code size increase of $(34 - 24) * 8 = 80$ bytes. As this example illustrates, any change in the size of the program would cause an overflow or underflow at the page boundary. It is neither easy nor practical to handle such changes in the page boundaries at run time. Hence, changes in code size due to rescheduling must be avoided.

This problem is solved via efficient encoding provided in TINKER (see Figure 2.10). The first three fields of a TINKER Op are: `Header bit`, `Optype`, and the `Pause`. An Op with `Header Bit` = 1 signifies the beginning of a new MultiOp, and it is called a *Header Op*. `Optype` indicates the type of the functional unit in which the Op will execute, thus bypassing the need for `nop`s within a MultiOp. The `pause` field in each Header Op encodes the number of empty cycles, if any, that would

follow the current MultiOp. (No meaning is attached to the value of `pause` in a non-Header Op). The Op encoding in TINKER thus *hides* the *nop*s, and ensures that code rescheduling within a basic block does not trigger any code size changes. A detailed discussion of the TINKER encoding, and the related properties of *Rescheduling Size-Invariance* that it exhibits, is presented in Chapter 4.

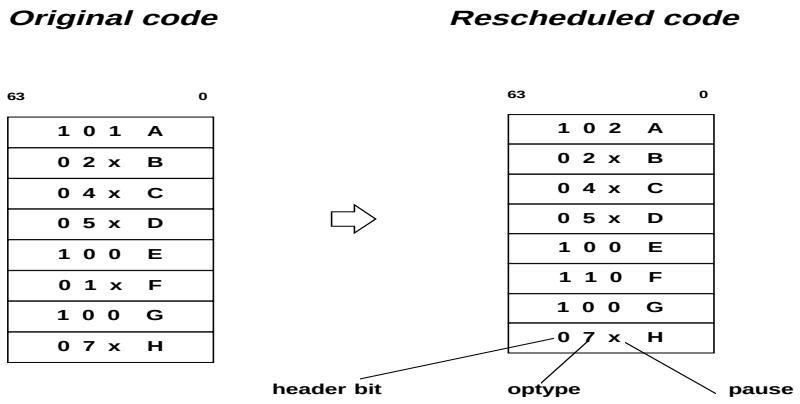


Figure 2.10: An example of TINKER Encoding scheme. Each Op is a fixed-format 64-bit word. The format includes the *Header Bit*, *optype*, and the *pause* fields, which together eliminate the need for *nop*s in code.

Figure 2.10 shows the rescheduled code previously shown in Figure 2.9, as it would be encoded in TINKER. It can be noted that the size of code has not changed, the *nop*s in each MultiOp have been squeezed out using the `optype` field, and empty cycles have been squeezed out using the `pause` field. The code size remains 64 bytes in total for both the machines.

2.4.2 Speculative code motion

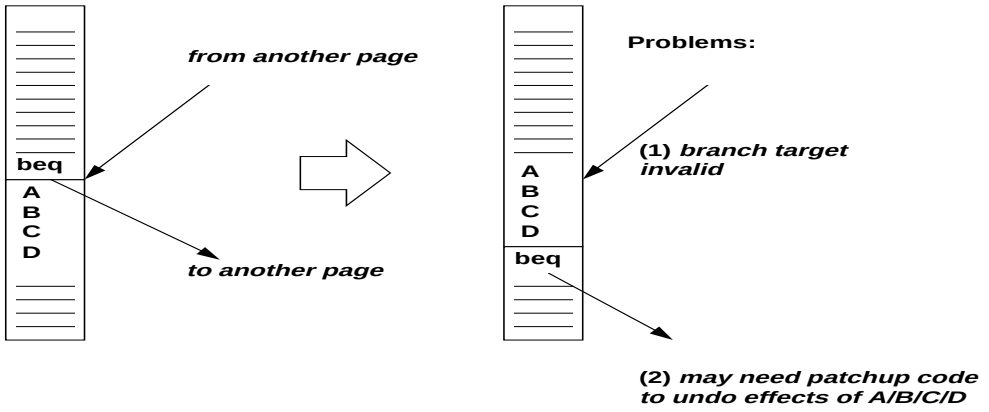


Figure 2.11: Problems introduced by speculative code motion.

Two problems are introduced by speculative code motion, if any, during rescheduling, and are illustrated with an example in Figure 2.11. The first problem is caused by incorrect execution of code due to target invalidation. In the example, Op A is the target of a branch from elsewhere in the code. If all Ops A, B, C, and D are speculatively moved above the branch (`beq`) which was originally before Op A, then this motion invalidates the target of the incoming branch. The second problem is caused by patch-up code which may be necessary to undo effects of code motion. For example, if the outgoing branch (`beq`) is taken, the effects of speculating Ops A, B, C, and D, may need to be undone by inserting patch-up/COPY code at the target of this branch. This target which may very well lie in another page, which may not be memory resident. Even if the target lies in the current page, code insertion would

lead to overflow at the page boundary. For these reasons, generation of patch-up code must be avoided during rescheduling.

To solve the first problem, the dynamic rescheduling algorithm takes advantage of the property of the single-entry acyclic regions (SEARs) such as Superblocks, Hyperblocks and Treegions. The SEARs, have a unique entry point at the top of the block, and no side-entrances (i.e. merge-points). Each SEAR in a page is rescheduled individually. The compiler is page-size cognizant when it forms the SEAR, guaranteeing that they do not span page boundaries. Since speculation is never performed across a region boundary, the possibility of branch target invalidation is eliminated.

The second problem could be solved by performing speculation only during the initial compilation and confining rescheduling to basic blocks (i.e. no speculation during rescheduling). But this would limit the opportunity to expose more ILP in a more parallel VLIW architecture. Instead, dynamic rescheduling performs limited speculation, but only if it requires no patch-up code. To support this, the compiler saves the live-out set info for each branch in the program in the object file (see Section 2.5). During rescheduling, if the rescheduler detects that a speculatable Op modifies a register in the live-out set of the Branch, it cancels the move. Any other Op not modifying a member of live-out set can be moved, since it does not require generation of patch-up code in the form of `COPY` operations.

2.4.3 Register file changes

Better hardware implementation techniques sometimes allow for more registers in an advanced generation, thus providing more opportunity to reduce register pressure. Although it is not traditional to allow a change in the register file size in the ISA, it is interesting to consider this possibility. From the perspective of dynamic rescheduling technique, such a change poses additional problems. When code is rescheduled for a machine with a different register file architecture, the rescheduler must perform register re-allocation. It is likely that spill code will be generated during this process, causing an increase in code size. This would violate the requirement that the page-size cannot change at run-time. The dynamic rescheduling scheme proposed in this thesis assumes that the register file architecture does not change across generations, although the algorithm can withstand the changes in register file architecture in a limited way. An *increase* in the number of registers with no change in compiler's subroutine calling conventions, can be handled by the algorithm without generation of spill-code. This is true only for the programs originally scheduled for the machine with a smaller register file, being rescheduled for the machine with a larger register file, and not *vice versa*.

2.5 Additional object-file information

The following is a brief review of the information included in the object file to support dynamic rescheduling. The version of the VLIW architecture for which the code is scheduled is encoded in the header of the object file, along with information about the architecture (i.e., the number and latency of each functional unit type). Alternatively, a machine architecture ID, which acts as a key to access the machine description database would suffice. Also, the block boundaries (for Hyperblocks and Superblocks) are marked using the `BlockBit` in Ops. The *live-out* register sets for each side-exit (branch) in the code are saved, and are used to guide the speculation decisions made during dynamic rescheduling.

These pieces of information about the object-code are saved as annotations in a non-loadable segment in the object-file. A comprehensive list of object-file annotations used in all phases of evolutionary compilation is presented in Chapter 7.

2.6 Operating system support

As was previously shown in Figure 1.1, the OS virtual memory management subsystem invokes the dynamic scheduling module in EC at first-time page faults. The OS program loader is modified to provide this support. Yet another part of the OS that plays crucial role is the file system buffer cache. The buffer cache routinely holds the pages (rescheduled or native) that were used in the recent past. This is a standard

mechanism available in modern operating systems, which directly helps amortize the cost of rescheduling over the first-time page accesses. The penalty of rescheduling is therefore not incurred for every page access made during the life-span of the program. A separate page-caching scheme called *Persistent Rescheduled-Page Cache*, which extends buffer-caching and saves the pages across multiple executions of the program is presented in Chapter 3.

Chapter 3

Performance of Dynamic Rescheduling and the PRC

3.1 Performance of Dynamic Rescheduling

The effectiveness of dynamic rescheduling was measured using programs from the SPECint92 benchmark suite [39], and several UNIX utility programs in [40] [39]. These are shown in Table 3.1. In order to construct an interactive load, the benchmarks were divided into two categories: tools (*cccp*, *026.compress*, *085.gcc*, *grep*, and *tbl*) and applications (*008.espresso*, *023.eqntott*, *022.li*, *lex*, *072.sc*, and *yacc*). It was assumed that tools would be invoked twice as often as applications, but that there would be no other pattern in the workload. Two sets of inputs were used for each benchmark, and were alternated between invocations of the benchmark. Using these

Table 3.1: Benchmarks used for evaluation of Dynamic Rescheduling.

Benchmark	Description
cccp	C pre-processor
tbl	Table formatter
grep	Pattern Search
lex	Scanner generator
yacc	Parser generator
008.espresso	PLA Optimization
022.li	LISP Interpreter
023.eqntott	Truth Table generator for logic circuits
026.compress	compression/decompression utility
085.gcc	GNU C Compiler
072.sc	Spreadsheet Calculator

assumptions, several workloads were created and used to measure the performance of the benchmarks with and without dynamic rescheduling.

3.1.1 Machine Models and Methodology

Three machine models were used for the evaluation of the performance of the benchmarks in situations that required dynamic rescheduling: Generations 1, 2, and 3 (their organizations are shown in Figure 3.1). The types of functional units are shown horizontally, while the execution latency assumptions are shown vertically. The unit `pred` is used to perform predicate computation in integer compare operations. Predicate computation in FP compare operations is done in the FPAdd unit.

A three part method was used to evaluate the dynamic rescheduling technique, as illustrated in Figure 3.2. In the first part, intermediate code for a benchmark was scheduled for a given machine model, using a VLIW scheduler; hyperblock scheduling

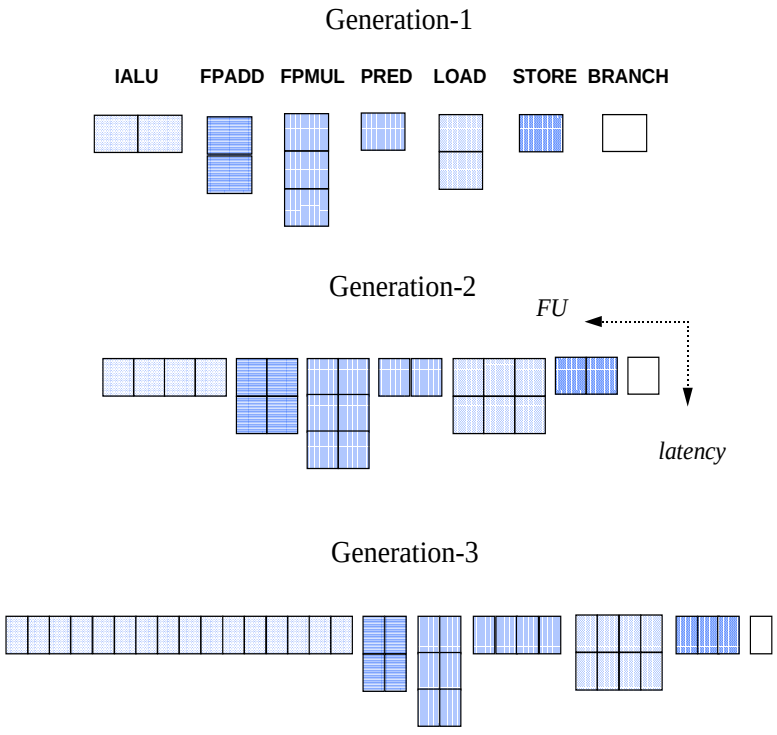


Figure 3.1: Machine models used to evaluate Dynamic Rescheduling and the Persistent Rescheduled-Page Cache (PRC) (See Section 3.2.1).

was used for the initial compilation [33]. The intermediate code was then profiled in order to find the worst–case estimate of execution time in terms of the number of cycles. The number of times each page of code is accessed is also recorded, which also indicates each unique code page that is accessed. This was called the *native* mode experiment. In the second part, the code scheduled for *native* mode execution was rescheduled for the other machine models. Execution time estimates for the rescheduled code were also generated as described before. This time estimate indicates the performance of the rescheduled code without taking into account the rescheduling

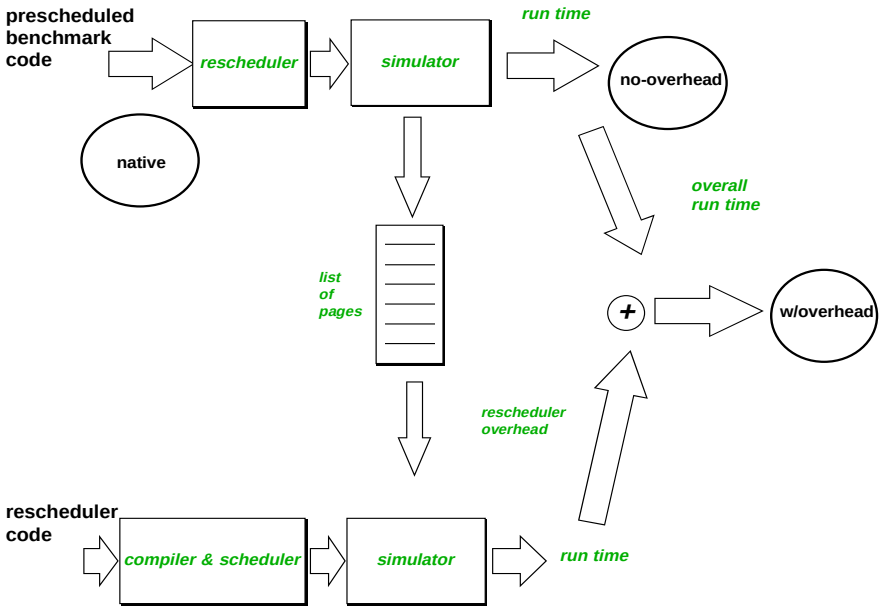


Figure 3.2: Experimental Method used to evaluate Dynamic Rescheduling.

overhead incurred by the rescheduler. Hence this part was termed the *no overhead* experiment.

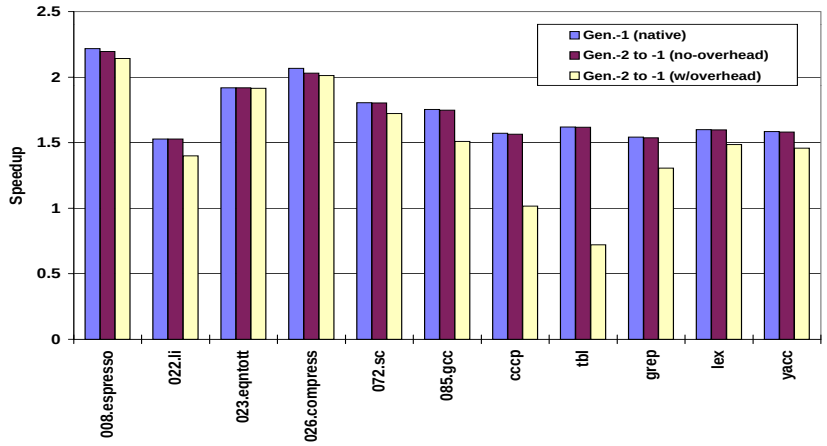
In the third part, the rescheduler itself was compiled, scheduled for the machine model used in the first part, and then used as a benchmark. The input to the the rescheduler benchmark was pages taken from each of the other benchmarks. The performance of the rescheduler benchmark was used to find the average time to reschedule a single 4K page for each of the three machine models. This was found to be 54,272 cycles for Generation–1, 51,200 cycles for Generation–2, and 48,108 cycles for Generation–3. This was then combined with the number of unique page accesses from the first part of the experiment to estimate the total number of execution cycles for the

rescheduling overhead. The rescheduling overhead was added to the execution times from the no–overhead experiment to derive execution times for the *w/overhead* experiment. Finally, to compare the performance achieved in the above three parts, the speedup with respect to a single-unit, single-issue processor model (the *base model*) was calculated. Speedup is: (*number of cycles of execution estimated in the experiment*)/(*number of cycles of execution estimated for the base model*). All three parts assumed a page size of 4K bytes, as is used in many contemporary operating systems [41] [42] and processors [43] [44].

Representative results for all the benchmarks, are shown in Figures 3.3, and 3.4. General trend is the same over all six combinations of rescheduling transformations (Gen.–1 to –2, and –3, Gen.–2 to –1, and –3, and Gen.–3 to –1, and –2). Harmonic means of the three speedup measurements over all the benchmarks are shown in Figure 3.5.

It can be seen that the *no overhead* speedup compares quite well with that of *Native*. The performance of rescheduled code when overhead is included (*w/overhead* is less than the *no overhead* results, as should be expected. This reflects the effectiveness of dynamic rescheduling as a run-time technique. The performance compares well, suggesting the technique is effective in most cases. The notable exceptions are *cccp*, *tbl*, and *grep*: in their case the penalty due to overhead is quite high. This phenomenon can be explained by observing the unique page access counts presented later in Table 3.2, which is identical to the number of first time page faults. *Cccp*, *tbl*, and

(a) Generation–2 to Generation–1



(b) Generation–3 to Generation–1

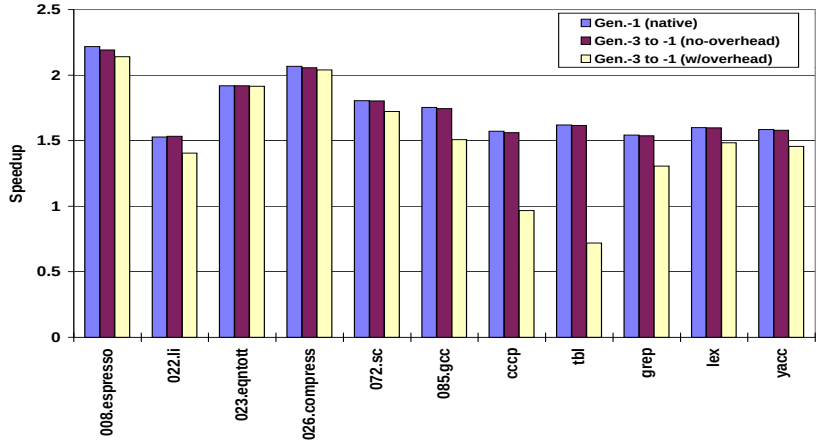
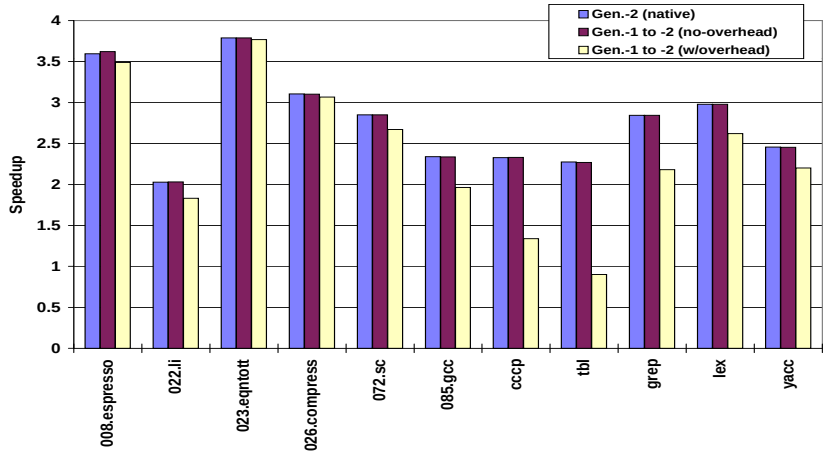


Figure 3.3: Results for Dynamic Rescheduling of (a) Generation–2 code to Generation–1 (b) Generation–3 code to Generation–1.

(a) Generation–1 to Generation–2



(b) Generation–1 to Generation–3

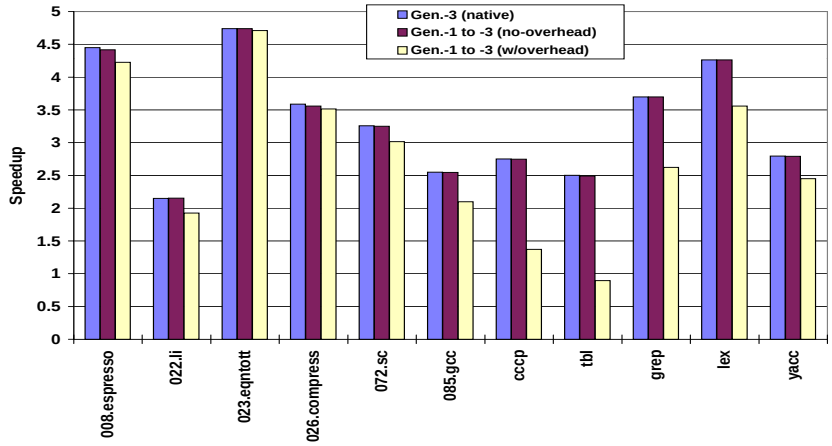


Figure 3.4: Results for Dynamic Rescheduling of (a) Generation–1 code to Generation–2 (b) Generation–1 code to Generation–3.

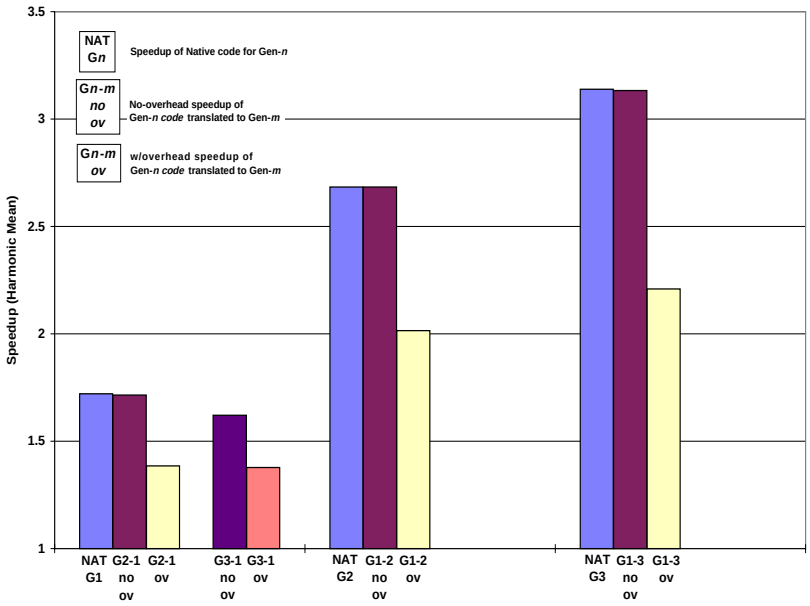


Figure 3.5: Speedups for programs under dynamic rescheduling framework. Each bar is the harmonic mean of speedups of benchmarks for a specific generation–to–generation rescheduling. Each set of bars is identified on the x-axis by a Gn-m label, which indicates that the code originally scheduled for Generation–n was rescheduled for Generation-m. The first bar in each set is native performance on Generation-m, the remaining bars show rescheduled performance, both with and without the overhead due to dynamic rescheduling.

grep are relatively short-running benchmarks, and the first time page faults are higher compared to their overall execution time than for the other benchmarks. This implies that dynamic rescheduling may not execute short-running programs efficiently. Techniques to reduce the overhead of dynamic rescheduling are further explored in Section 3.2.

Since dynamic rescheduling is performed when the OS handles a page-fault, some

analysis of the process of page-fault handling is now due. The dominant time in page-fault handling is the time spent reading the page from disk if the executable is stored on the local machine. If the page is being accessed over a LAN (as is common in a client-server environment), the network latency is the most dominant factor, and is at least an order of magnitude more than the local disk access. The time spent rescheduling a page adds to this page access latency. To estimate the temporal overhead dynamic rescheduling would add to page-fault handling in a local paging environment, an experiment was conducted to measure the page-fault service time when paging from a set of contiguous blocks on a local disk. The SAR performance analysis tool [45] was used to measure average page fault service time on two hardware platforms and was found to be about 2500 μs [1], when averaged over a total of 50 page-faults. Based on average execution times on a 100 MHz machine, dynamic rescheduling increases the page fault service time by about 20%, a significant increase (rescheduling a 4 kilobyte page, with 64-bit operations, from the Generation-2 code to Generation-1 code, for example, would take 54,272 cycles times 10 ns, which is approximately 543μs, more than 20% of the average page-fault service time measured here). It is apparent that when paging over a LAN, the relative overhead of dynamic rescheduling will be lower, but probably not small enough to completely neglect it.

There are two approaches that can be used to reduce the overhead: (1) improve the performance of the rescheduling algorithm, or (2) reduce the number of pages that

[1]SAR was run on a 133 Mhz Pentium–based Data General computer running DG–UX, and a 99 MHz Hewlett–Packard 9000/715 computer running HP–UX.

require rescheduling over multiple invocations of the program. The second component is the focus of the investigation in this chapter. The OS saves rescheduled pages in the text swap if a program is swapped out during execution. These rescheduled pages are effectively *cached* on disk during a program's current execution. Extending this concept, the OS can aid in caching rescheduled pages not only during a single program execution but across multiple program executions as well. The following section introduces an OS–supported caching scheme that achieves this.

3.2 Disk caching and the Persistent Rescheduled-Page Cache

The significant overhead introduced by dynamic rescheduling can be largely alleviated through the use of a caching scheme that employs a *persistent rescheduled-page cache (PRC)*. A PRC is an OS–managed disk cache that holds binary images of rescheduled pages; if rescheduled pages are cached, they will not need to be rescheduled when re–accessed across program runs. A PRC is defined by its behavior during a program execution and at program termination. During execution, rescheduled pages are stored in text swap space. When a program terminates, its rescheduled pages are written to the PRC. Page placement and replacement policies are implemented within the OS.

The concept behind the persistent rescheduled-page cache originated from software–

managed disk caching, a proven method for reducing I/O traffic [46], [47].

The idea is to cache the most recently accessed disk blocks in memory in the hope that they will be accessed again in the near future. Typical Unix systems implement a file system buffer cache to hold the most recently accessed disk blocks in memory using a LRU algorithm for replacement [46], [47]. The buffer cache effectively operates as a fragmented page cache also and reduces the amount of I/O during page faults [2]. Variants of this have been used in distributed file systems such as Sprite [48] and Andrew [49], for the purposes of remote file caching. Disk caching has also been used to reduce translation overhead in architectural interpreters [50], and can be used effectively with dynamic rescheduling.

3.2.1 Persistent Rescheduled-Page Caches

A PRC can be organized as a system–wide cache: a portion of the file system that holds only rescheduled pages and managed by the OS. Pages from different programs can displace each other in this case. The primary configuration parameters for the PRC are the size, placement and replacement policies, and are discussed below.

During initial program execution on a non–native host, text pages are rescheduled at first–time page faults, as described in Chapter 2. If a rescheduled page is displaced from physical memory during program execution, it is written to the text swap space on disk; this prevents rescheduling a page multiple times during program execution.

[2]The term "fragmented" is used because all of the disk blocks that comprise a page might not be in the buffer cache at the same time.

At the end of the initial execution, all of the rescheduled pages are written to the PRC (the page placement policy will be explained shortly). During subsequent executions, when a page fault occurs for a text page with a generation mismatch, the PRC is probed to check for the presence of a rescheduled version of the page. If a rescheduled version is present, it is retrieved and loaded into physical memory, and the overhead of rescheduling is not incurred. If a rescheduled version is not available, the page is retrieved from the binary image of the program, rescheduled, and then loaded into physical memory. If this page is replaced during program execution, it is written to the text swap space. At program termination, all rescheduled pages of a program are written to the PRC. A rescheduled page can be placed into any entry in the cache, which implies that the cache is effectively fully associative with a replacement policy such as LRU. An outline of the algorithm used for PRC management is shown in Figure 3.6.

Probing the PRC on disk for the presence of a page is an expensive operation. This can be eliminated by modifying a program's disk block pointers during execution and program exit. When a page is rescheduled and subsequently written to the PRC at program termination, a separate set of disk block pointers (*PRC pointers*) for the program are set to point to the rescheduled versions of pages in the cache. A PRC pointer is annotated to indicate which original page the rescheduled page replaces. This scheme implements a PRC probe as an examination of a program's disk block pointers in one central location rather than multiple locations in the PRC on disk.

Algorithm *PRC Management*
begin
  do
    begin
      • Load next new program;
      • If rescheduled pages for the program exist in the PRC,
        set page-table entries to disk addresses in PRC;
      • Begin program execution;
      • At program termination, write all rescheduled pages into
        the PRC; if needed, displace existing pages in
        the PRC using appropriate displacement algorithm
        (LRU, *overhead-based replacement*, etc.)
      • Update program PRC disk-block pointers to point to program
        file data structure in the OS kernel;
    end
end

Figure 3.6: Persistent Rescheduled-Page Cache (PRC) management algorithm.

Disk block pointers can be cached by the OS, which can further reduce the number of disk probes. The rescheduled version of a page is accessed without probing the PRC on disk and perhaps without any disk accesses at all, if the program's disk block pointers are in the in–memory cache. Rescheduled pages that are stored in swap during program execution are managed by the OS using known methods for managing text swap space [46].

The probing of multiple PRC pointers during page faults can also be eliminated. At program load time, the disk block and PRC pointers for the program are examined to determine if rescheduled versions of pages exist in the PRC. If a page has a rescheduled version, the loader modifies the page table entry (PTE) for that page to point to the rescheduled version. When a page fault occurs for a page that has a rescheduled

version, the rescheduled version is accessed directly, using the updated PTE. No disk accesses or in–memory searches are required to implement a PRC probe: if the PTE for a page points into the PRC, a rescheduled version exists. The re-mapping of the PTE entries can be done at program load time [46], [47]. As rescheduled pages are displaced from the use of a PRC due to replacement, the PRC pointers to the displaced pages must be nullified. This can be accomplished with OS support. A table can be maintained by the PRC manager that lists the locations of the PRC pointers associated with each page in the cache. When a page is replaced, its PRC pointer is set to null.

3.2.2 PRC performance

Figure 3.7 presents the performance of PRCs of different sizes for rescheduling across four generation–to–generation combinations. The metric used is speedup over a single universal–unit, single–issue processor. Each set of bars shows the harmonic mean of speedups for a rescheduling combination, for various PRC sizes. As indicated here, PRC-n means a PRC of size n pages. PRC-infinite indicates the performance when rescheduling of the unique page accesses was performed only at the initial invocations. This is essentially a PRC without an upper bound on its size. PRC-infinite speedups are the same as the no-overhead case speedups mentioned in Section 3.1. PRC-0 indicates the performance when no PRC is used, and all the pages uniquely accessed by the program are rescheduled at each invocation. This provides a measure of the

worst-case overhead of rescheduling. A page can displace any other page in the PRC, based on the LRU replacement policy. This organization is called a *unified PRC*.

Performance of all of the rescheduling combinations benefits from the use of a PRC. The trend is that a larger PRC provides greater speedup. Note that for PRC-1024, perfect speedup (equal to the PRC-infinite) case is achieved. This happens because the total number of pages in the workload is less than the size of the PRC and all the programs completely reside in a PRC of that size without any requirement to reschedule once the PRC is populated.

Table 3.2: Unique page counts of the benchmarks.

BENCHMARK	UNIQUE PAGE COUNT
cccp	34
tbl	50
grep	4
lex	45
yacc	56
008.espresso	137
022.li	47
023.eqntott	25
026.compress	8
072.sc	60
085.gcc	323

All of the benchmarks do not benefit equally from the presence of a PRC. To illustrate this, Figure 3.8 presents speedups for individual benchmarks for Generation–1 to Generation–3 rescheduling. Some benchmarks – such as *008.espresso*, *023.eqntott*, *026.compress* – show only a small improvement with even a large PRC. Others, such

Table 3.3: Overhead Ratio (O): Generation–1 to Generation–3 rescheduling.

BENCHMARK	OVERHEAD RATIO (percentage)	OVERHEAD CATEGORY
cccp	**50.09**	**high**
tbl	**64.15**	**high**
grep	**29.10**	**high**
lex	16.51	moderate
yacc	12.19	moderate
008.espresso	4.35	low
022.li	10.52	moderate
023.eqntott	0.64	low
026.compress	1.25	low
072.sc	6.29	moderate
085.gcc	17.50	moderate

as *cccp, tbl, grep* show substantial improvement with the increasing PRC size. Moderate improvement is shown by *022.li, 072.sc, 085.gcc, lex*, and *yacc*. The reason behind this behavior can be explained using the *overhead ratio* metric for a program. The overhead ratio is defined as:

$$O = R/(E + R)$$

where E is the execution time of the program, and R, the total rescheduling overhead = (*the Unique Page Count of the program* $*$ *avg. time required to reschedule a page*). The *unique page count* of a program is defined as the number of first–time page faults that occur during the execution of the program. Values of the unique page counts are shown in Table 3.2. Table 3.3 shows the percentage values of the overhead ratio for Generation-1 to Generation-3 rescheduling, and Figure 3.9 shows a plot of the

overhead ratios for ease of comparison. A high overhead ratio (20% and above in this case) indicates that the amount of time taken to reschedule is relatively high. Such programs benefit the most from the use of a PRC – *cccp, tbl*, and *grep* as shown here – and are termed *high-overhead* programs. For programs which showed the least performance improvement, the overhead ratio is relatively small (less than 5%). These programs are termed *low-overhead* programs (*008.espresso, 023.eqntott*, and *026.compress*). All the others (*022.li, 072.sc, 085.gcc, lex*, and *yacc* – they all have a value of O between 5% and 20%) are referred to as *moderate-overhead* programs.

In the unified PRC, the low- or moderate-overhead programs can evict the high-overhead programs completely if their unique page count is large. Thus, for example, even if the unique page count of a high-overhead program such as *cccp* is small (35 in this case), a moderate-overhead program such as *085.gcc* can replace all the pages allocated to *cccp* because it's unique page count is relatively high (323 in this case). If these two programs are run in an alternate fashion (which one would expect, because one is a C preprocessor, and the other is a C compiler), the number of pages for *cccp* that will have to be rescheduled could be sizable, thus increasing its overhead, especially for smaller PRC sizes. A better organization of the PRC which reduces this effect is presented next.

3.2.3 Split PRC

Because program behavior in the presence of a PRC varies directly with a program's overhead ratio, one approach is to *partition* the cache to hold pages for different classes of programs, based on the program overhead ratio. This should prevent *cache pollution*: a benchmark that benefits very little from a PRC displacing the pages of a program whose performance is substantially enhanced by a PRC. OS–gathered statistics can be dynamically used to compute the overhead ratios of the programs, and to determine which PRC partition a program should use. The OS needs to record the unique page counts, the program execution time, along with the average time take to reschedule a page, for this purpose.

A PRC with two partitions can be labeled a *2-way split PRC*: one partition to hold the low- and moderate-overhead programs, and the other for the exclusive use of the high-overhead programs. Figure 3.10 presents results for such a PRC across various generation–to–generation reschedulings and PRC sizes. (Here, both the partitions are of the same size, though this is not a requirement – the partition sizes can be varied depending on the program unique page counts.) Performance for practically all generation–to–generation rescheduling combinations improved using the 2-way split PRC over a unified PRC. In particular, for the Generation–1 to Generation–3 rescheduling using a PRC-512, the speedup was 91% of the speedup when using an infinite PRC (infinite PRC corresponds to the no–overhead experiment as described in Section 3.1). The Generation–1 to Generation–2 rescheduling also showed improve-

ment with a split cache, particularly for PRC-256 and PRC-512. The general trend was that larger PRCs performed better, as was expected. The speedups of individual benchmarks with a 2-way split PRC, for Generation-1 to Generation-3 translation is shown in Figure 3.11. All of the high-overhead benchmarks fared well, compared to their performance under a unified PRC (Figure 3.8). Their low performance for the smaller PRC sizes is because they compete with each others in the same partition. It can also be observed that the cache pollution effect still persists, but is substantially reduced (for example, observe the improvement in performance for *cccp* and *tbl* as compared to the unified PRC, for all PRC sizes). Based on these experiments, an N-way split cache with M pages per partition would do better than a unified cache with $M * N$ entries. In an actual implementation, a PRC can be partitioned with as much granularity as the OS allows.

3.2.4 Unified PRC with overhead-based replacement

Another technique to reduce the cache pollution effect observed in the unified PRC is the use of an *overhead-based* PRC page replacement policy instead of LRU. This works as follows. With each page in the PRC is associated the overhead ratio for the program to which the page belongs. A page is not allowed to replace another page from the PRC unless its overhead ratio is higher than the overhead ratio of that page. This replacement policy ensures that more of the high-overhead programs stay PRC-resident, once placed in the PRC. The priority of use of the PRC is governed by the

overhead ratio: the higher the overhead ratio, the higher the priority. Consequently, the low- and moderate-overhead programs incur higher rescheduling overhead, since a relatively less number of their pages get cached between invocations. Intuitively, this scheme is similar to the multiple dynamically re-sizable partitions within a PRC.

Figure 3.12 shows the performance of individual benchmarks for Generation–1 to Generation–3 translation for this scheme. It can be observed that the top three high-overhead programs – *cccp, tbl, grep* – perform much better for this replacement policy than for LRU (Figure 3.8). They show a perfect speedup for PRC sizes 128 and above. (The low speedups for small PRC sizes are due to a large number of capacity misses.) On the other hand, the rest of the programs show a decrease, albeit small, in the performance for smaller PRC sizes compared to LRU. For the larger PRCs, their performance has, however, improved [3]. This is based on the fact that for smaller PRC sizes, they too encounter a large number of capacity misses, effectively displacing each other from the PRC and incurring a larger rescheduling overhead.

In case of *008.espresso, 022.li, 023.eqntott, 026.compress, 072.sc*, and *lex*, however, there is little or no performance gain for any of the PRC sizes. This is explained by considering the case of *lex*. *Lex* has a relatively small unique page count of 45 (refer to Table 3.2), compared to the 323 of *085.gcc*. It's overhead ratio (16.51), however, is only slightly lower than that of *085.gcc* (17.50) (refer to Table 3.3). *Lex* gets displaced from the PRC after every invocation of *085.gcc*, and incurs the overhead of

[3] As mentioned earlier, all the programs have a perfect speedup for PRC-1024 because it is large enough to hold the entire workload throughout the experiment.

rescheduling all of it's pages at each subsequent invocation. It is speculated that the impact of this phenomenon may be reduced if *085.gcc* was allowed to displace these programs only if they were invoked in the relatively remote past. A combination of the LRU and the overhead-based schemes may prove effective in such cases, and is a topic of future research.

Nonetheless, all of these programs (*008.espresso, 022.li, 023.eqntott, 026.compress, 072.sc*, and *lex*) being low- or moderate-overhead, the overall performance across all programs is better than the previous two schemes. This is apparent from Figure 3.13. When compared with the performance of the unified PRC with LRU replacement (Figure 3.7), and the 2-way split PRC with LRU replacement (Figure 3.10), the overhead-based replacement performs better across all the PRC sizes. The general performance trend observed for the overhead-based replacement policy confirms the intuition that the priority of use of the PRC is dictated by the overhead ratio.

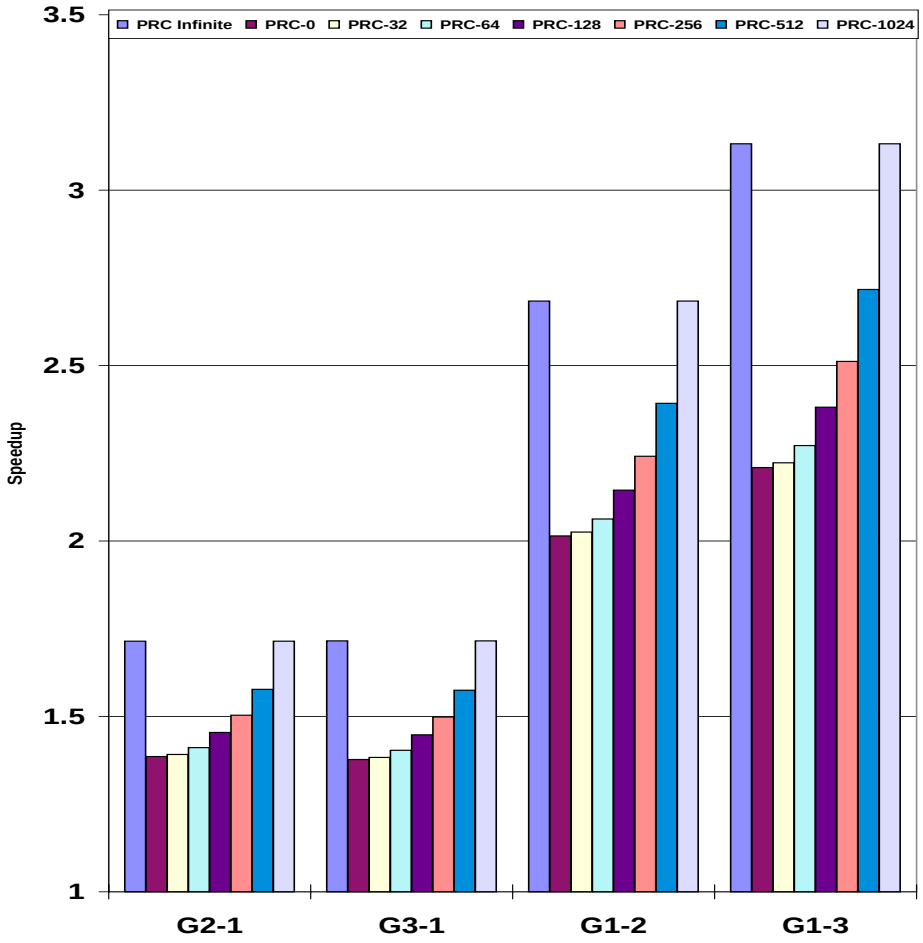


Figure 3.7: Speedups of benchmarks for PRC performance, unified PRC, LRU replacement. Each bar is a harmonic mean of speedups of the benchmarks for a particular generation–to–generation rescheduling and PRC size. Each set of bars is identified on the x-axis by a Gn-m label, which indicates that code originally scheduled for Generation–n was rescheduled for Generation-m. The first bar in each set of bars is the no–overhead performance, the second bar is the worst–case overhead (no PRC), and subsequent bars are performance with the indicated PRC size; PRC size is the maximum number of pages the cache holds.

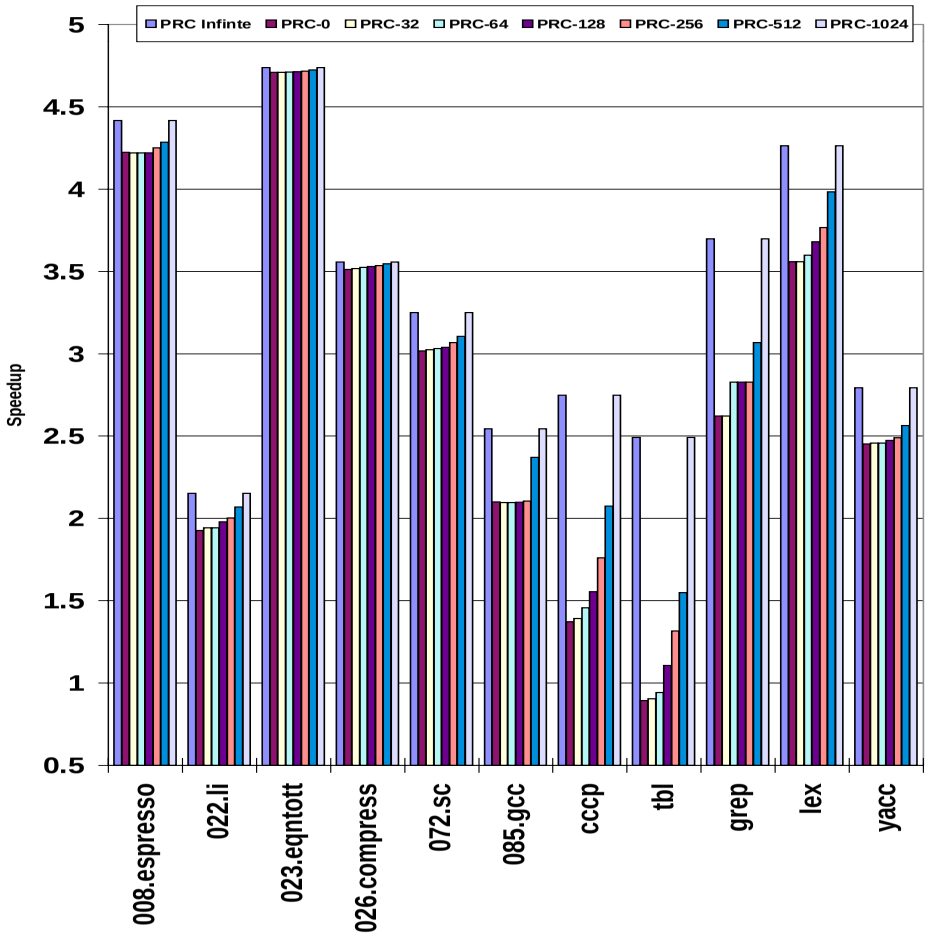


Figure 3.8: Generation–1 to Generation–3 rescheduling, unified PRC, LRU replacement. Each bar is the speedup for a Generation–1 to Generation–3 rescheduling for the specified PRC size. Each set of bars corresponds to an individual benchmark, as indicated by the labels on x-axis. The first bar in each set of bars is the no–overhead performance, the second bar is the worst–case overhead (no PRC), and subsequent bars are performance with the indicated PRC size; PRC size is the maximum number of pages the cache holds.

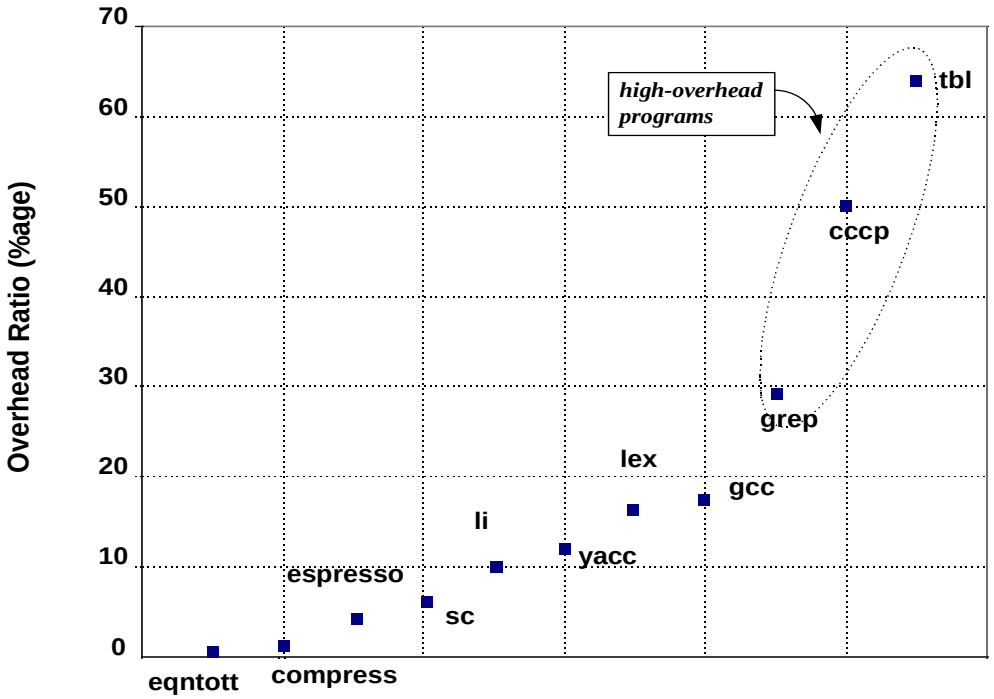


Figure 3.9: Overhead ratios of various programs in the experimental framework.

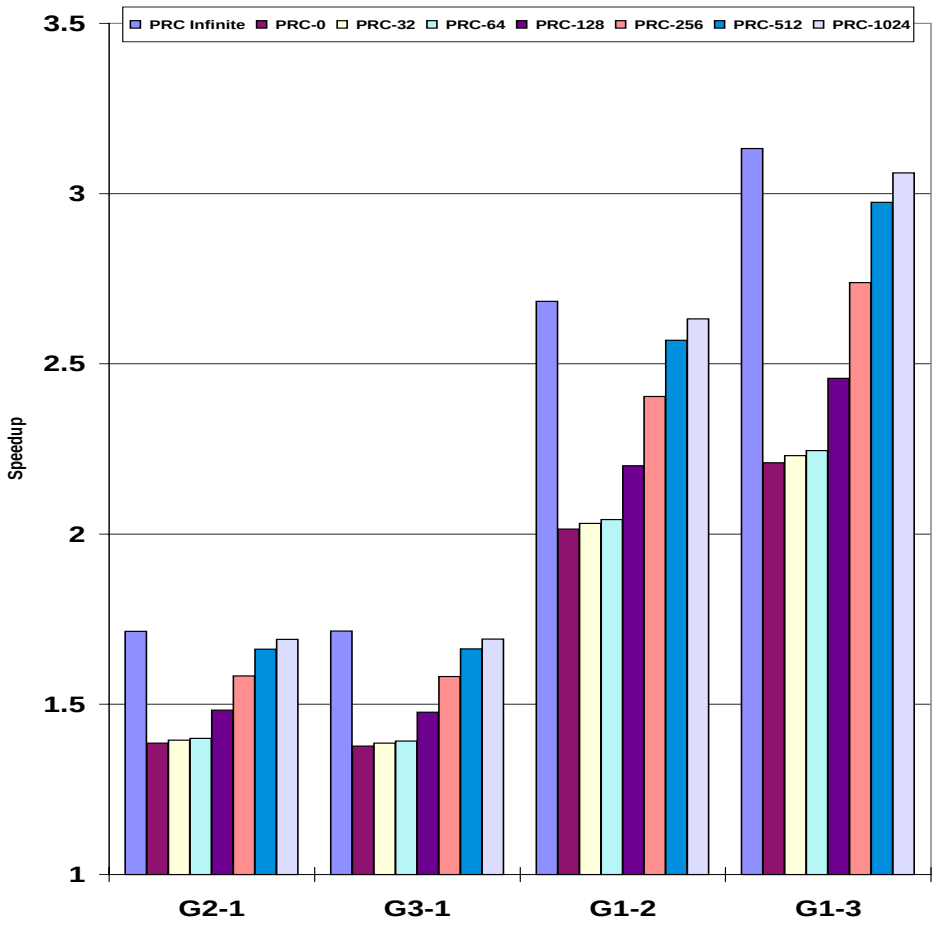


Figure 3.10: Speedups for PRC performance: 2-way split PRC, LRU replacement.

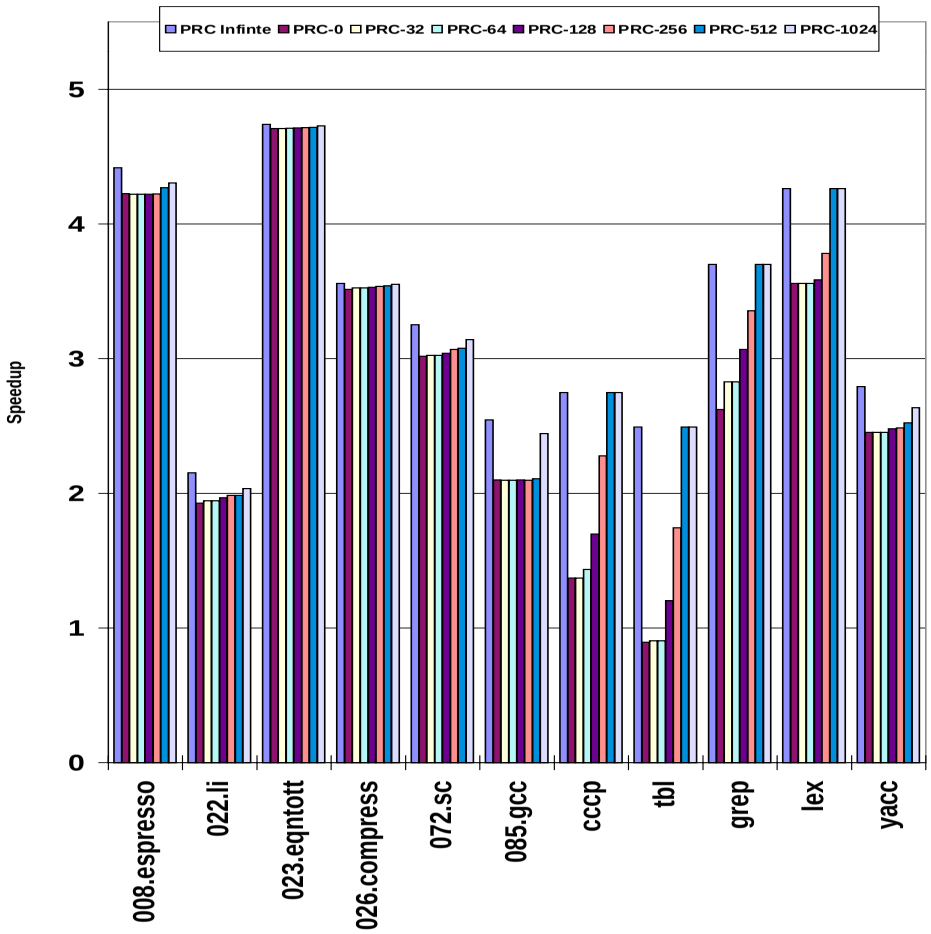


Figure 3.11: Generation–1 to Generation–3 rescheduling: 2-way split PRC, LRU replacement.

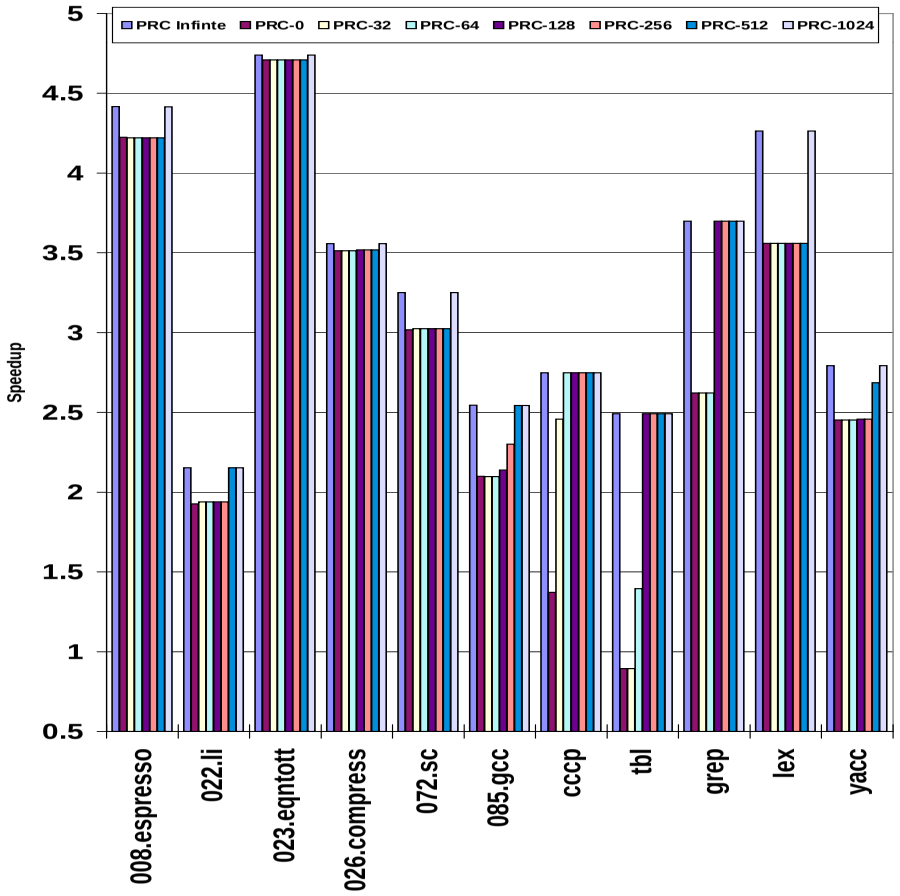


Figure 3.12: Generation–1 to Generation–3 rescheduling, Unified PRC, overhead-based replacement.

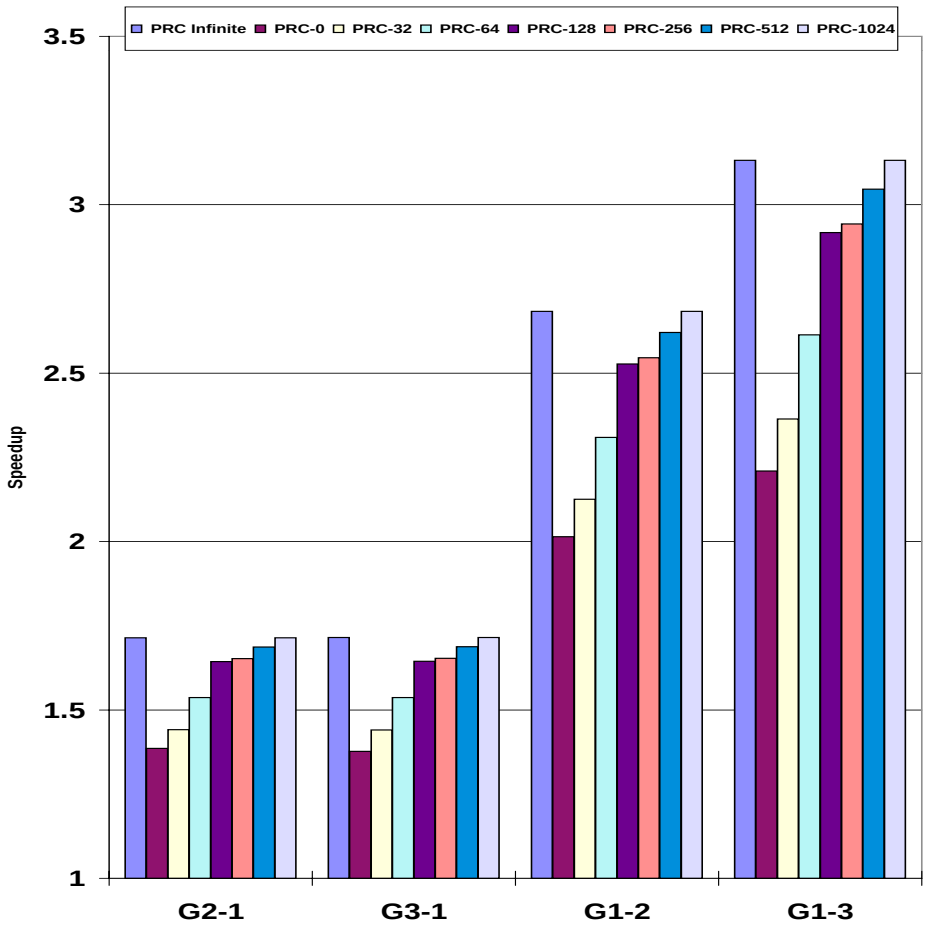


Figure 3.13: Speedups for PRC performance: Unified PRC, overhead-based replacement.

Chapter 4

TINKER List Encoding and Rescheduling Size Invariance

The *TINKER List Encoding* is an ISA encoding which requires no explicit representation of NOPs and empty MultiOps in the object-code, and hence it is a *zero-NOP* encoding. This property of list encoding is used to support evolutionary compilation in the TINKER architecture. This chapter presents a formal definition of the TINKER list encoding, followed by an introduction to the concept of *Rescheduling Size-Invariance (RSI)*. Also, the property that an arbitrary list-encoded piece of object-code is *rescheduling size-invariant* is studied for cyclic and acyclic object-code.

4.1 List encoding in TINKER

Definition 1 (Op) *A VLIW Operation (*Op*) is defined by a 6-tuple:* {H, p_n, s_{pred}, FUtype, opcode, operands}, *where,* $H \in \{0, 1\}$ *is a 1-bit field called* header-bit, p_n *is an n-bit field called* pause, s.t. $p_n \in \{0, \ldots, 2^n - 1\}$, s_{pred} *is a stage predicate (discussed further in Section 4.4),* FUtype *uniquely identifies the FU instance where the* Op *must execute,* opcode *uniquely identifies the task of the* Op, *and* operands *is the set of valid operands defined for* Op. *All* Ops *have a constant width.* □

Definition 2 (Header Op) *An* Op, O, *is a* Header Op *iff the value of the* header-bit *field in* O *is 1.* □

Definition 3 (VLIW MultiOp) *A VLIW MultiOp,* M, *is defined as an unordered sequence of* Ops $\{O_1, O_2, \ldots, O_m\}$, s.t. $0 < m \leq w$, *where* w *is the number of hardware resources to which the Ops are issued concurrently, and* O_1 *is a Header Op.* □

Definition 4 (VLIW Schedule) *A VLIW schedule,* S, *is defined as an ordered sequence of MultiOps* $\{M_1, M_2, \ldots\}$. □

A discussion of the list encoding is now in order. All Ops in this scheme of encoding are fixed-width. In a given VLIW schedule, a new MultiOp begins at a Header Op and ends exactly at the Op before the next Header Op; the MultiOp fetch hardware uses this rule to identify and fetch the next MultiOp. The value of the p_n field in an Op is referred to as the *pause*, because it is used by the fetch hardware to stop

MultiOp fetch for the number of machine cycles indicated by p_n. This is a mechanism devised to eliminate the explicit encoding of empty MultiOps in the schedule. The *FUtype* field indicates the functional unit where the Op will execute. The *FUtype* field allows the elimination of NOPs inserted by the compiler in an arbitrary MultiOp. Prior to the execution of a MultiOp, its member Ops are routed to their appropriate functional units based on the value of their *FUtype* field.

This scheme of encoding the components of a VLIW schedule is termed as *List encoding*. Since the size of every Op is the same, the size of a given list encoded schedule, S, can be expressed in terms of the number of Ops in it:

$$sizeof(S) = \sum_i \sum_j O_j$$

where i is the number of MultiOps in S, O_j is an Op, and j is the number of Ops in a given MultiOp.

4.2 Rescheduling Size Invariance

Definition 5 (VLIW Generation) *A VLIW* generation *G is defined by the set $\{R, L\}$, where R is a set of hardware resources in G, and $L \in \{1, 2, \ldots\}$ is the set of execution latencies of all the Ops in the operation set of G. R itself is a set consisting of pairs $\{r, n_r\}$, where r is a* resource type *and n_r is the number of instances of r.*

This definition of a VLIW generation does not model complex resource usage patterns for each Op, as used in [51], [52] and [53]. Instead, each member of the set of machine resources R, presents a higher-level abstraction of the "functional units" found in modern processors. Under this abstraction, the low-level machine resources such as the register-file ports and operand/result busses required for the execution of an Op on each functional unit are bundled with the resource itself. All the resources indicated in this manner are assumed to be busy through the period of time equal to the latency of the executing Op, indicated by the appropriate member of set L. This abstraction of machine resources is used to merely simplify the view of the machine, and is not a pre-constraint for the property of *Rescheduling Size-Invariance* (presented below) to hold. The property would hold equally well for a more complex machine abstraction.

Definition 6 (Rescheduling Size-Invariance (RSI)) *A VLIW schedule S is said to satisfy the RSI property iff* $\text{sizeof}(S_{G_n}) = \text{sizeof}(S_{G_m})$, *where S_{G_n}, S_{G_m} are the versions of the original schedule S prepared for execution on arbitrary machine generations G_n and G_m respectively. Further, schedule S is said to be* rescheduling size-invariant *iff it satisfies the RSI property.* □

The proof that list encoding is RSI will be presented in two parts. First, it will be shown that acyclic code in the program is RSI when list encoded, followed by the proof that the cyclic code is RSI when list encoded. Since all code is assumed to be either acyclic or cyclic, the result that list encoding makes it RSI will follow. In the

remainder of this section, algorithms to reschedule each of these types of codes are presented, followed by the proofs themselves.

4.3 Rescheduling Size-Invariant Acyclic Code

The algorithm to reschedule acyclic code from VLIW generation G_{old} to generation G_{new} is shown in Algorithm *Reschedule_Acyclic_Code*. It is assumed that both the old and new schedules are LTE schedules, and that both have the same register file architecture and compiler register usage convention.

Algorithm *Reschedule_Acyclic_Code*
input
S_{old}, the old schedule, (assumed no more than n_{old} cycles long);
$G_{old} = \{R_{old}, L_{old}\}$, the machine model description for the *old* VLIW;
$G_{new} = \{R_{new}, L_{new}\}$, the machine model description for the *new* VLIW;
output
S_{new}, the new schedule;
var
n_{old}, the length of S_{old};
n_{new}, the length of S_{new};
Scoreboard[number of registers], to flag the registers "in–use" in S_{old}.
$RU[n_{new}][\sum_r n_r]$, the resource usage matrix, where:
r represents all the resource types in G_{new}, and,
n_r is the number of instances of each resource type r in G_{new};
UseInfo$[n_{new}]$[number of registers], to mark the register usage in S_{new};
T_δ, the cycle in which an Op can be scheduled while
satisfying the data dependence constraints;
$T_{rc+\delta}$, the cycle in which an Op can be scheduled while satisfying
the data dependence and resource constraints;
functions
RU_loopkup $(O\,(T_\delta))$ returns the earliest cycle, later than cycle T_δ, in which Op O can be scheduled after satisfying the data dependence and resource constraints;
RU_update (σ, O) marks the resources used by Op O in cycle σ of S_{new};

$dest\_register\,(O)$ returns the destination register of Op O;
$source\_register\,(O)$ returns a list of all source registers of Op O;
$latest\_use\_time\,(\phi)$ returns the latest cycle in S_{new} that register ϕ was used in;
$most\_recent\_writer\,(\rho)$ returns the id of the Op which modified register ρ latest in S_{old};

begin
for each MultiOp $M_{old}[c] \in S_{old}$, $0 \leq c \leq n_{old}$ **do**
begin
— *resource constraint check:*
for each Op $O_w \in S_{old}$ that completes in cycle c **do**
begin
$O_w\,(T_{rc+\delta}) \leftarrow RU\_loopkup\,(O_w\,(T_\delta))$;
$M_{new}[new\_cycle] \leftarrow M_{new}[new\_cycle] \mid O_w$;
$RU\_update\,(T_{rc+\delta}, O_w)$;
end
— *update the scoreboard:*
for each Op $O_i \in S_{old}$ which is unfinished in cycle c **do**
begin
$Scoreboard[dest\_register\,(O_r)] \leftarrow reserved$;
end
— *do data dependence checks:*
for each Op $O_r \in M_{old}[c]$ **do**
begin
$O_r\,(T_\delta) \leftarrow 0$;
— *anti-dependence:*
for each $\phi \in dest\_register\,(O_r)$ **do**
$O\,(T_\delta) \leftarrow MAX(O\,(T_\delta), latest\_use\_time\,(\phi))$;
— *pure dependence:*
for each $\phi \in source\_register\,(O_r)$ **do**
if $(Scoreboard[\phi] = reserved)$
$O_r\,(T_\delta) \leftarrow MAX\,(O_r(T_\delta),\ completion\_time\,(reserving\_Op))$;
— *output dependence:*
for each $\phi \in dest\_register\,(O_r)$ **do**
if $(Scoreboard[\phi] = reserved)$
$O_r\,(T_\delta) \leftarrow MAX\,(O_r(T_\delta),\ completion\_time\,(reserving\_Op))$;
end
end
end

The RSI property for list encoded acyclic code schedule will now be proved.

Theorem 1 *An arbitrary list encoded schedule of acyclic code is RSI.*

Proof: The proof will be presented using induction over the number of Ops in an arbitrary list encoded schedule. Let L_i be an arbitrary, ordered sequence of i Ops ($i \geq 1$) that occur in a piece of acyclic code. Let F_i denote a directed dependence graph for the Ops in L_i, *i.e.* each Op in L_i is a node in F_i, and the data- and control-dependences between the Ops are indicated by directed arcs in F_i. Let S_{G_n} be the list encoded schedule for L_i generated using the dependence graph F and designed to execute on a certain VLIW generation G_n. Also, let G_m denote another VLIW generation which is the target of rescheduling under dynamic rescheduling.

Induction Basis. L_1 is an Op sequence of length 1. In this case, $sizeof(S_{G_n}) = 1$, and the dependence graph has a single node. It is trivial in this case that S_{G_n} is RSI, because after rescheduling to generation G_m, the number of Ops in the schedule will remain 1, or,

$$sizeof(S_{G_n}) = sizeof(S_{G_m}) = 1 \tag{4.1}$$

Induction Step. L_p is an Op sequence of length p, where $p > 1$. Assume that S_{G_n} is RSI. In other words,

$$sizeof(S_{G_n}) = sizeof(S_{G_m}) = p \tag{4.2}$$

Now consider the Op sequence L_{p+1}, which is of length $p + 1$, such that to L_{p+1} was obtained from L_p by adding one Op from the original program fragment. Let this

additional Op be denoted by z. Op z can be thought of as borrowed from the original program, such that the correctness of the computation is not compromised. L_p is an ordered sequence of Ops, and Op z must then be either a prefix of L_p, or a suffix to it. Also, let T_{G_n} denote the list encoded schedule for sequence L_{p+1}, which means *sizeof* $(T_{G_n}) = p + 1$. In order to prove the current theorem, it must now be proved that T_{G_n} is RSI if S_{G_n} is RSI.

The addition of Op z to L_p may change the structure of the dependence graph F_p in two ways: (1) if the Op z adds one or more data dependence arcs to F_p, or (2) the Op z does not add any data dependence arcs to F_p.

- **Op z adds dependence(s)**:

  In this case, Op z is control- and/or data-dependent on one or more of the Ops in L_p, or *vice versa*. Under the assumptions of dynamic rescheduling, during the initial scheduling pass or during the invocation of the *Reschedule_Acyclic_Code* algorithm, only NOPs and empty MultiOps are added to/deleted from the schedules T_{G_n} and T_{G_m} to honor the added dependences or possible machine resource constraints. When the schedules T_{G_n} and T_{G_m} are list encoded, the empty MultiOps are made implicit using the *pause* field in the Header Op of the previous MultiOp, and the NOPs in a MultiOp will be made implicit via the *FUtype* field in the Ops. Thus, the only addition to both schedules T_{G_n} and T_{G_m} is the newly added Op z.

- **Op z does not add any dependences**:

In this case, only the resource constraints, if any, would warrant the insertion of empty MultiOps. By an argument similar to that in the previous case, it is trivial to see that the only source of size increase in schedules T_{G_n} and T_{G_m} is the newly added Op z.

Thus, in both the cases, $sizeof(T_{G_n}) = sizeof(S_{G_n}) + 1$. From this and from Equation 4.2, it follows that:

$$sizeof(T_{G_n}) = p + 1 \tag{4.3}$$

Similarly, for both the cases, $sizeof(T_{G_m}) = sizeof(S_{G_m}) + 1$, which leads to:

$$sizeof(T_{G_m}) = p + 1 \tag{4.4}$$

From Equations 4.3 and 4.4, and by induction, it follows that an arbitrary list encoded schedule of acyclic code is RSI. ■

An example of this transition of code was previously demonstrated in Figures 2.9 and 2.10. It can be observed that the size of the original code (on the left) in Figure 2.10 is the same as that of the rescheduled code (on the right). The NOPs and the empty MultiOps have been eliminated in the *list encoded* schedules; the rescheduling algorithm merely re-arranged the Ops, and adjusted the values of the H and the p_n (pause) fields within the Ops to ensure the correctness of execution on G_{new}.

4.4 Rescheduling Size-Invariant Cyclic Code

Most programs spend a great deal of time executing the inner loops, and hence the study of scheduling strategies for inner loops has attracted great attention in literature [54], [55], [56], [19], [57], [58], [59], [60]. Inner loops typically have small bodies (relatively fewer Ops) which makes it hard to find ILP within these loop-bodies. Software pipelining is a well-understood scheduling strategy used to expose the ILP across multiple iterations of the loop [61], [56]. There are two ways to perform software pipelining. The first one uses *loop unrolling*, in which the loop body is unrolled a fixed number of times before scheduling. Loop bodies scheduled via unrolling can be subjected to rescheduling via the *Reschedule_Acyclic_Code* algorithm described in Section 4.3. The code expansion introduced due to unrolling, however, is often unacceptable, and hence the second technique, *Modulo Scheduling* [61], is employed. Modulo-scheduled loops have very little code expansion which makes it very attractive. In this thesis, only modulo-scheduled loops are examined for the RSI property; unrolled-and-scheduled loops are covered by the acyclic RSI techniques presented previously. First, some discussion of the structure of modulo-scheduled loops in presented, followed by an algorithm to reschedule modulo scheduled code. The section ends with a formal treatment to show the list-encoded modulo-scheduled cyclic code is RSI. Concepts from Rau [60] are used as a vehicle for the discussion in this section.

Assumptions about the hardware support for execution of modulo scheduled loops

are as follows. In some loops, a datum generated in one iteration of the loop is consumed in one of the successive iterations (an inter-iteration data dependence). Also, if there is any conditional code in the loop body, multiple, data-dependent paths of execution exist. Modulo-scheduling such loops is non-trivial[1]. This thesis assumes three forms of hardware support to circumvent these problems. First, register renaming via *rotating registers* [60] in order to handle the inter-iteration data dependencies in loops is assumed. Second, to convert the control dependencies within a loop body to data dependencies, support for *predicated execution* [64] is assumed. Third, support for *sentinel scheduling* [65] to ensure correct handling of exceptions in speculative execution is assumed. Also, the *pre-conditioning* [60] of counted-DO loops is presumed to have been performed by the modulo scheduler when necessary.

A modulo scheduled loop, Ω_{G_n}, consists of three parts: a *prologue* (π_{G_n}), a *kernel* (κ_{G_n}), and an *epilogue* (ε_{G_n}), where G_n is the machine generation for which the loop was scheduled. The prologue initiates a new iteration every II cycles, where II is known as the *initiation interval*. Each slice of II cycles during the execution of the loop is called a *stage*. In the last stage of the first iteration, execution of the kernel begins. More iterations are in various stages of their execution at this point in time. Once inside the kernel, the loop executes in a steady state (so called because the kernel code branches back to itself). In the kernel, multiple iterations are simultaneously in progress, each in a different stage of execution. A single iteration completes at

[1]See [62] and [63] for some of the work in this area.

the end of each stage. The branch Ops used to support the modulo scheduling of loops have special semantics, by which the branch updates the loop counts and enables/disables the execution of further iterations. When the loop condition becomes false, the kernel falls through to the epilogue, which allows for the completion of the stages of the unfinished iterations. Figure 4.1 shows an example modulo schedule for a loop and identifies the prologue, kernel, and the epilogue. Each row in the schedule describes a cycle of execution. Each box represents a set of Ops that execute in a same resource (e.g. functional unit) in one stage. The height of the box is the *II* of the loop. All stages belonging to a given iteration are marked with a unique alphabet $\in \{A, B, C, D, E, F\}$.

Figure 4.1 also shows the loop in a different form: the *kernel-only (KO)* loop [57] [60]. In a kernel-only loop, the prologue and the epilogue of the loop "collapse" into the kernel, without changing the semantics of execution of the loop. This is achieved by predicating the execution of each distinct stage in a modulo scheduled loop on a distinct predicate called a *stage predicate*. A new stage predicate is asserted by the loop-back branch. Execution of the stage predicated on the newly asserted predicate is enabled in the future executions of the kernel. When the loop execution begins, stages are incrementally enabled, accounting for the loop prologue. When all the stages are enabled, the loop kernel is in execution and the loop is in the steady state. When the loop condition becomes false, the predicates for the stages are reset, thus disabling the stages one by one. This accounts for the iteration of the epilogue of

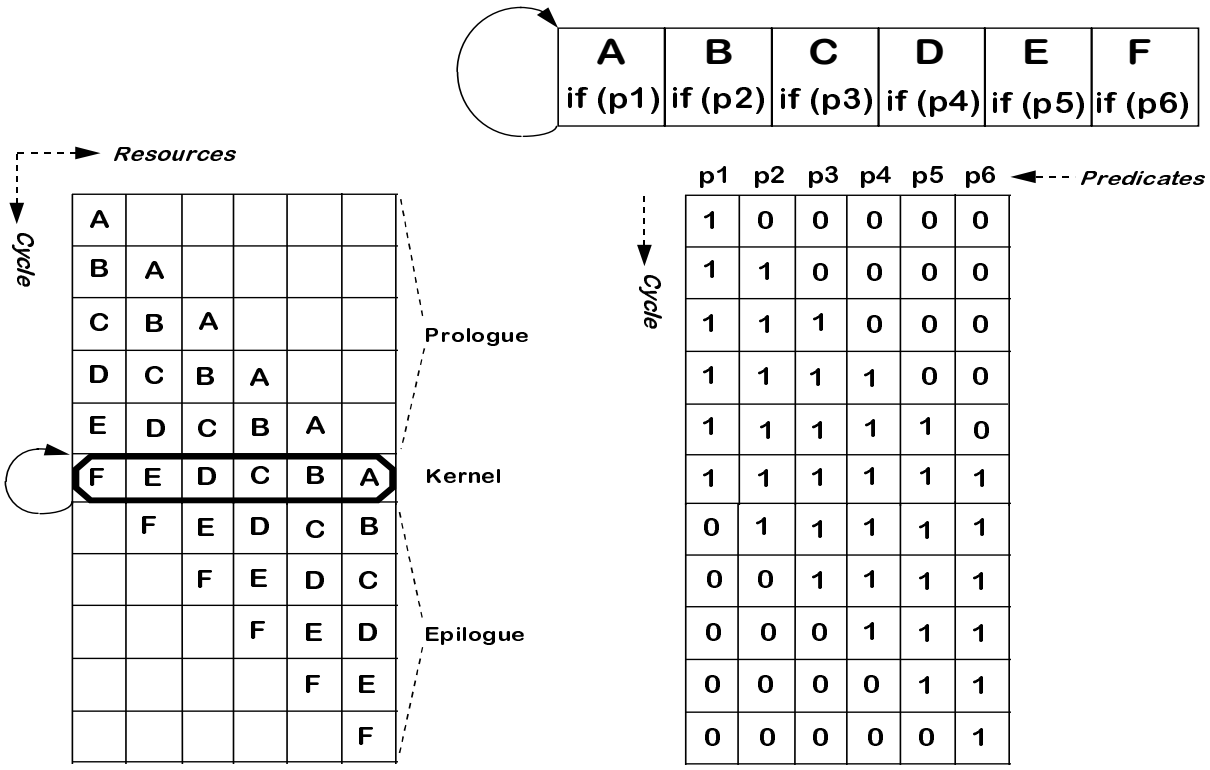


Figure 4.1: A Modulo scheduled loop: on the left, a modulo scheduled loop (with the *prologue*, *kernel*, and the *epilogue* marked) is shown. The same schedule is shown on the right, but in the "collapsed" *kernel-only (KO)* form. *Stage predicates* are used to turn the execution of the Ops ON or OFF in a given stage. The table shows values that the stage predicates $\{p1, p2, p3, p4, p5, p6\}$ would take for this loop.

the loop. A modulo scheduled loop can be represented in the KO form, if adequate hardware (predicated execution) and software (a modulo scheduler to predicate the stages of the loop) support is assumed. Further discussion of KO loop schedules can be found in [60]. All modulo-scheduled loops can be represented in the KO form. The KO form thus has the potential to encode modulo schedules for all classes of loops, a property which is useful in the study of dynamic rescheduling of loops, as will be shown shortly.

The size of a modulo scheduled loop is larger than the original size of the loop, if the modulo schedule has an explicit prologue, a kernel, and an epilogue. In contrast, a KO loop schedule has exactly one copy of each stage in the original loop body, and hence has the same size as the original loop body, provided the original loop was completely if-converted[2]. This property of the KO loops is useful in performing dynamic rescheduling of modulo scheduled loops. Algorithm *Reschedule_KO_Loop* details the steps. The input to the algorithm is the modulo scheduled KO loop, and the machine models for the old and the new generations G_{old} and G_{new}. Briefly, the algorithm works as follows: identification of the predicates that enable individual stages is performed first. An order is imposed on them, which then allows for the derivation of the order of execution of stages in a single iteration. The ordering on the predicates may be implicit in the predicate-id used for a given stage (increasing order of predicate-ids). Alternatively, the order information could be stored in the ob-

[2]For any *pre-conditioned* counted DO-loops, the size is the same as the size of loop body after pre-conditioning.

ject file and made available at the time DR is invoked, without substantial overhead. Once the order of execution of the stages of the loop is obtained, the reconstruction of the loop in its original, unscheduled form is performed. At this time, the modulo scheduler is invoked on it to arrive at the new KO schedule for the new generation.

Algorithm *Reschedule_KO_Loop*
input
Ω_{old} =the KO (kernel-only) modulo schedule such that:
number of stages = n_{old}; initiation interval = II_{old};
$G_{old} = \{R_{old}, L_{old}\}$, the machine model description for the *old* VLIW;
$G_{new} = \{R_{new}, L_{new}\}$, the machine model description for the *new* VLIW;
output
Ω_{new}: KO (kernel-only) modulo schedule for G_{new};
var
$B\,[n_{old}]$, the table of n_{old} buckets each holding the Ops from a unique stage, such that the relative ordering of Ops in the bucket is retained;
functions
FindStagePred (O) returns the stage predicate on which Op O is enabled or disabled;
BucketOP $(O,\, B\,[p])$ puts the Op O into the bucket $B\,[p]$;
OrderBuckets $(B,\, func)$ sorts the table of buckets B according to the ordering function *func*;
StagePredOrdering () describes the statically imposed order on the stage predicates;
begin
— *unscramble the old modulo schedule:*
for all MultiOps $M \in \Omega_{old}$ **do**
for each Op $O \in M$
begin
p = *FindStagePred* (O);
BucketOP $(O,\, B\,[p])$;
end
— *order the buckets:*
OrderBuckets $(B,$ *StagePredOrdering* $())$;
— *perform modulo scheduling:*
Perform modulo scheduling on the sorted table of buckets B, using the algorithm described by Rau in [61] to generate KO schedule Ω_{new};
end

The RSI nature of List encoded modulo scheduled KO loop will now be proved.

Theorem 2 *An arbitrary List encoded Kernel-Only modulo schedule of a loop is RSI.*

Proof: Let L_i be an arbitrary, ordered sequence of i Ops ($i \geq 1$) which represents the loop body. Let F_i denote a directed dependence graph for the Ops in L_i, *i.e.* each Op in L_i is a node in F_i, and the data- and control-dependences between the Ops are indicated by directed arcs in F_i. Note that the inter-iteration data dependences are also indicated in F_i. Let Ω_{G_n} denote a list encoded KO modulo schedule for generation G_n. Also, let G_m denote the VLIW generation for which rescheduling is performed.

Induction Basis. L_1 is a loop body of length 1. In this case, $sizeof(\Omega_{G_n}) = 1$, and the dependence graph has a single node. It is trivial in this case that Ω_{G_n} is RSI, because after rescheduling to generation G_m, the number of Ops in the schedule will remain 1, or,

$$sizeof(\Omega_{G_n}) = sizeof(\Omega_{G_m}) = 1 \tag{4.5}$$

(Note that a loop where $sizeof(\Omega_{G_n}) = 1$ is the degenerate case).

Induction Step. L_p is a loop body of length p, where $p > 1$. Assume that Ω_{G_n} is RSI. In other words,

$$sizeof(\Omega_{G_n}) = sizeof(\Omega_{G_m}) = p \tag{4.6}$$

Now consider another loop body L_{p+1}, which is of length $p+1$. Let the $p+1^{th}$ Op be denoted by z. Also, let Θ_{G_n} denote the list encoded KO modulo schedule for L_{p+1}, which means $sizeof\ (\Theta_{G_n}) = p+1$. In order to prove the theorem at hand, it must now be proved that Θ_{G_n} is RSI if Ω_{G_n} is RSI.

It is possible that due to addition of Op z in L_{p+1}, the nature of the graph F_p could change in two ways: (1) Op z is data dependent on one or more Ops in L_{p+1} or *vice versa*, or (2) the Op z is independent of all the Ops in L_{p+1}. In both of these cases, the data dependences and the resource constraints are honored by the modulo scheduling algorithm via appropriate use of NOPs and/or empty MultiOps within the schedule. When this schedule is list encoded, the NOPs and the empty MultiOps are made implicit via the use of *pause* and the *FUtype* fields within the Ops. Thus, the only net addition to the space requirements of schedule Θ_{G_n} is that of Op z. Hence,

$$sizeof(\Theta_{G_n}) - sizeof(\Omega_{G_n}) = sizeof(z) = 1 \tag{4.7}$$

In other words,

$$sizeof(\Theta_{G_n}) = sizeof(\Omega_{G_n}) + 1 \tag{4.8}$$

From this result and from Equation 4.6, it follows that:

$$sizeof(\Theta_{G_n}) = p + 1 \tag{4.9}$$

Similarly, for both the cases, $sizeof(\Theta_{G_m}) = sizeof(\Omega_{G_m}) + 1$, which leads to:

$$sizeof(\Theta_{G_m}) = p + 1 \tag{4.10}$$

From Equations 4.9 and 4.10, and by induction, it follows that an arbitrary list encoded KO modulo schedule is RSI. ■

Corollary 1 *A List encoded schedule is RSI.*

Proof: All program codes can be divided into the two categories: the acyclic code and cyclic code. Hence, It follows from Theorem 1 and Theorem 2 that a list encoded schedule is RSI. ■

Chapter 5

Performance Shift due to Profile Variations

Profile-based optimizations utilize information gathered from program runs with a subset of inputs. Due to this input-dependent nature, the optimizations may lead to performance degradation if the nature of the input differs substantially. This problem is termed as *profile shift*. Alternatively, if the input used to gather program profile does not exercise all portions of the program call-graph and control-flow graph, the profile information may be inadequate to perform optimizations on all parts of the program. This phenomenon is termed as the *lack of representativeness* of profile information. In order to reduce the effect of this phenomenon, the program developer may collect a large number of profiles using multiple inputs and combine them (e.g. by simply summing them) in the hope that the combined profile will be representative

of the typical workload that the program is expected to execute. This solution, while effective in some cases, is not guaranteed to be so on all inputs.

The objective of this chapter is to demonstrate the phenomena of profile variation and the lack of representativeness. Experimental results on the industry-standard benchmark suite SPECInt95 are presented for this purpose. Two types of program codes are considered in this set of experiments: (1) programs which are compiled using *Superblocks* as the basic structure for compilation, and (2) programs which use *Treegions* [34] as the basic structure. Detailed analysis of the results is presented with the hope that it will provide an insight into the decision-making process for evolutionary compilation. Related and previous work done by other researchers is briefly discussed in Section 5.3.

5.1 Performance shift: a case study using Superblock code

5.1.1 Metrics

The following metric is used as a measure of performance shift for a program on an arbitrary input a:

$$\text{Change in runtime on input } a,\ \mathcal{C}^a = \frac{\mathcal{T}_b^a - \mathcal{T}_a^a}{\mathcal{T}_a^a} \tag{5.1}$$

where $\mathcal{T}_b^a$ indicates the runtime of the program code profiled, optimized and scheduled using input b, and running on input a. In the case when $a = b = i$, $\mathcal{T}_i^i$ is the runtime of the code optimized and scheduled for input i, while executing on the same input i. It is expected that $\mathcal{T}_i^i$ will be the shortest runtime for the program running on input i. Thus, in Equation 5.1, the quantity $\mathcal{T}_a^a$ in the numerator indicates the shortest expected runtime for the program on input a. Since the program code used to measure the other quantity ($\mathcal{T}_b^a$) was profiled using input b, it is expected that it would perform at most as good as or worse than $\mathcal{T}_a^a$, (*i.e.* $\mathcal{T}_b^a \geq \mathcal{T}_a^a$). This metric attempts to capture the relative difference in runtime of the two program codes on the same input. It caters well to the aim of this section, which is to show that input and machine model variations impact performance. Other researchers have used slightly different metrics to measure the impact of input variations; these will be described later in Section 5.3.

5.1.2 Machine models

Three classes of machine models were used in these experiments, with each class having a 4-issue (*"T4"*), an 8-issue (*"T8"*), and a 16-issue (*"T16"*) machine. The classes of machines are shown in Figure 5.1. The *SU-BR1* class has machines with special-purpose functional units (*"SU"*), and each machine includes a single *BRANCH* unit (*"BR1"*). Similarly, the *SU-BR2* class includes machines with special-purpose functional units (*"SU"*), and with each machine having two *BRANCH* units (*"BR2"*).

Four types of special-purposed functional units are considered: *INT, FP, MEM, and BRANCH*. INT executes all the integer operations, as well as the predicate computation and checking operations (where necessary). The FP units perform floating-point operations. MEM units perform LOADs and STOREs. BRANCH operations exclusively handle the branches. The third class is termed *UU*. All machines in this class are composed of *universal* functional units: any functional unit in a UU-class machine can execute any type of operation in the ISA. As examples of the nomenclature used in this report, *SU-BR2-T16* indicates a 16-issue machine in the class SU-BR2, and *UU-T4* indicates a 4-issue machine in the class UU. All the machines are assumed to be statically scheduled. Also, the machines are assumed to have 64 general-purpose, 64 floating-point, and 32 double-precision architected registers. The double-precision registers are overlapped with the floating-point registers. The execution latencies of operations vary with the operation type; the latency assumptions for this study are shown in Table 5.1. Both the instruction and the data caches are assumed to be perfect.

This choice of machine models was based on the following observations about the benchmarks used. As will be presented in Section 5.1.3, all the benchmarks used in this study are control-intensive applications with integer computations. Class SU-BR1 abstracts machines which have limited capability to perform branching in a single cycle (1 branch per cycle). The effect of deviations in the inputs on the performance is expected to be the most apparent for machines in this class. The class SU-BR2 has

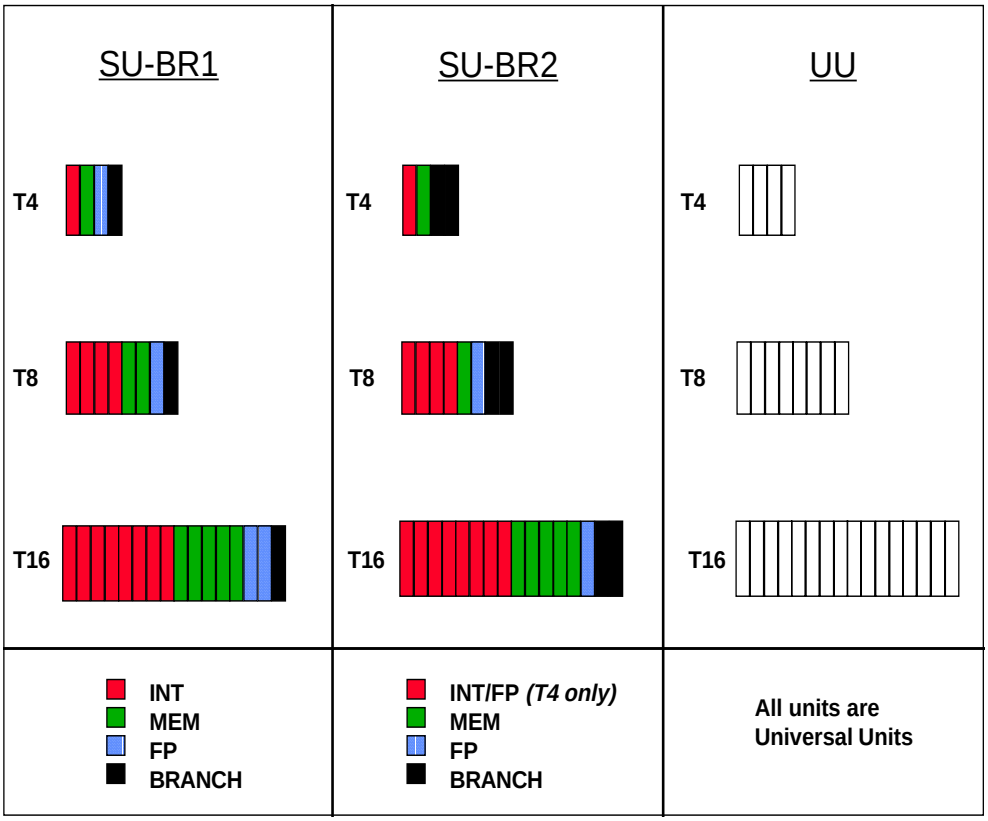


Figure 5.1: Machine models used to study performance shift: three classes of machines are used in this study. Each rectangle represents a functional unit. Execution latencies of the operations vary with the operation type.

Table 5.1: Operation execution latencies.

Operation type	Execution latency (cycles)
Integer arithmetic	1
Memory loads	2
Memory stores	1
Predicate computation	1
Predicate compare	1
Floating-point addition	2
Floating-point multiply	3
Floating-point divide	9
Branch	1

machines with two BRANCH units, which means the machines have more branching freedom, and hence are expected to show reduced deviations. Class UU incorporates machines which allow the most freedom for the issue-widths considered because potentially all the functional units can be used to perform branching in a given cycle. Besides, the universal nature of the functional units reduces resource contention. It is expected that this class of machines will show the least effect of deviations in program inputs.

5.1.3 Benchmarks

All eight programs from the SPECInt95 benchmark suite were used (shown in Table 5.2) to study performance shift. Program inputs used are also indicated. Where possible, the reference inputs provided by SPEC are used. In the case where multiple reference inputs were unavailable, either a test input (also provided by SPEC) was used or a new input was created by hand.

Table 5.2: Benchmarks used to study performance shift due to profile variations.

Benchmark	Description	Input–1	Input–2
099.go	AI: the game of *Go*	`5stone21`	`9stone21`
124.m88ksim	Motorola 88K chip simulator	`dhry`	`dcrand`
126.gcc	GNU C compiler	`2toplev.i`	`2stmt.i`
129.compress	File compression/decompression utility	`bigtest`	`bigtest2`
130.li	LISP interpreter	`8queens`	`reflsp`
132.ijpeg	Graphic compression and decompression	`penguin`	`specmun`
134.perl	Pattern Extraction and Report Language	`primes`	`scrabbl`
147.vortex	OO-DBMS	`test`	`ref`

The procedure used to measure shift in performance is illustrated in Figure 5.2. For each program the program code was function-inlined using call-graph profile of the program for a training input run. The code was then converted into the intermediate format of the compiler, and was then subjected to classical optimizations. Two profiling runs were then performed on this optimized code: one each on one of the two inputs a or b. Using the profile gathered on these runs, the code was subjected to *superblock formation* and *superblock optimization* [66]. As the end result of these transformations, two program codes, one profiled and optimized for input a and the other for input b were generated. Finally, the code was scheduled for each of the machine models and profiled with inputs a and b. These runs yielded the final values of the program run times, *viz.* $\mathcal{T}_b^a$, $\mathcal{T}_a^a$, $\mathcal{T}_a^b$, and $\mathcal{T}_b^b$. These measurements were made using the Impact compiler [67].

All the benchmarks considered in this study are control-intensive, integer computations. No floating-point code was considered for this study. This decision was based

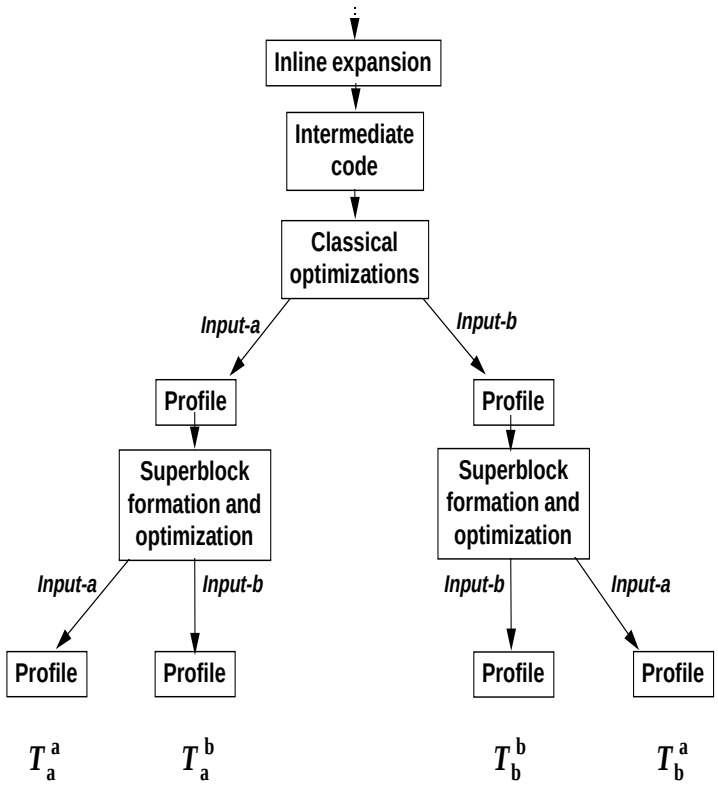


Figure 5.2: Method used for measurement of the change in performance: Inputs a & b are used to profile the code and the profile information is used to perform superblock formation and optimization. Superblock code thus generated is re-profiled using each input to estimate the runtime in each of the four cases.

on the observation that the floating-point applications have a largely predictable control, and hence are not suitable subjects for this study. Another important thing to note is that the results will be used to make a case for *dynamic* object-code rescheduling. In this light, benchmarks which are of on-line, interactive nature are better candidates for this study. It is worth noting here that benchmarks such as *130.li* (a LISP interpreter), *134.perl* (an interpreter commonly used in interactive tasks and), *147.vortex* (an object-oriented DBMS), and *099.go* (non-interactive version of an interactive game) are the most suitable from this point of view.

5.1.4 Results and analysis

Figures 5.3, 5.4, and 5.5 show the results of the performance shift measurements for machine classes SU-BR1, SU-BR2, and UU respectively. Each figure has one plot per benchmark. Each plot shows the performance shift that the benchmark exhibited on each of its inputs. The machine models in the class are lined along the X-axis.

In all the cases, all the benchmarks show a shift in performance (sans one exception: *129.compress* on SU-BR2-T16 (Figure 5.4)). A positive shift indicates a degradation in the performance. For example, notice the value of $\mathcal{C}^{primes} = 9.90\%$ for *134.perl* on machine SU-BR2-T16 (Figure 5.4). This means that for that machine, the benchmark when profiled and scheduled using the other input *(primes)* performs 9.90% worse on the input *primes* than when it is profiled and scheduled using *primes*. This positive shift in the performance is expected (as mentioned in Section 5.1.1), and

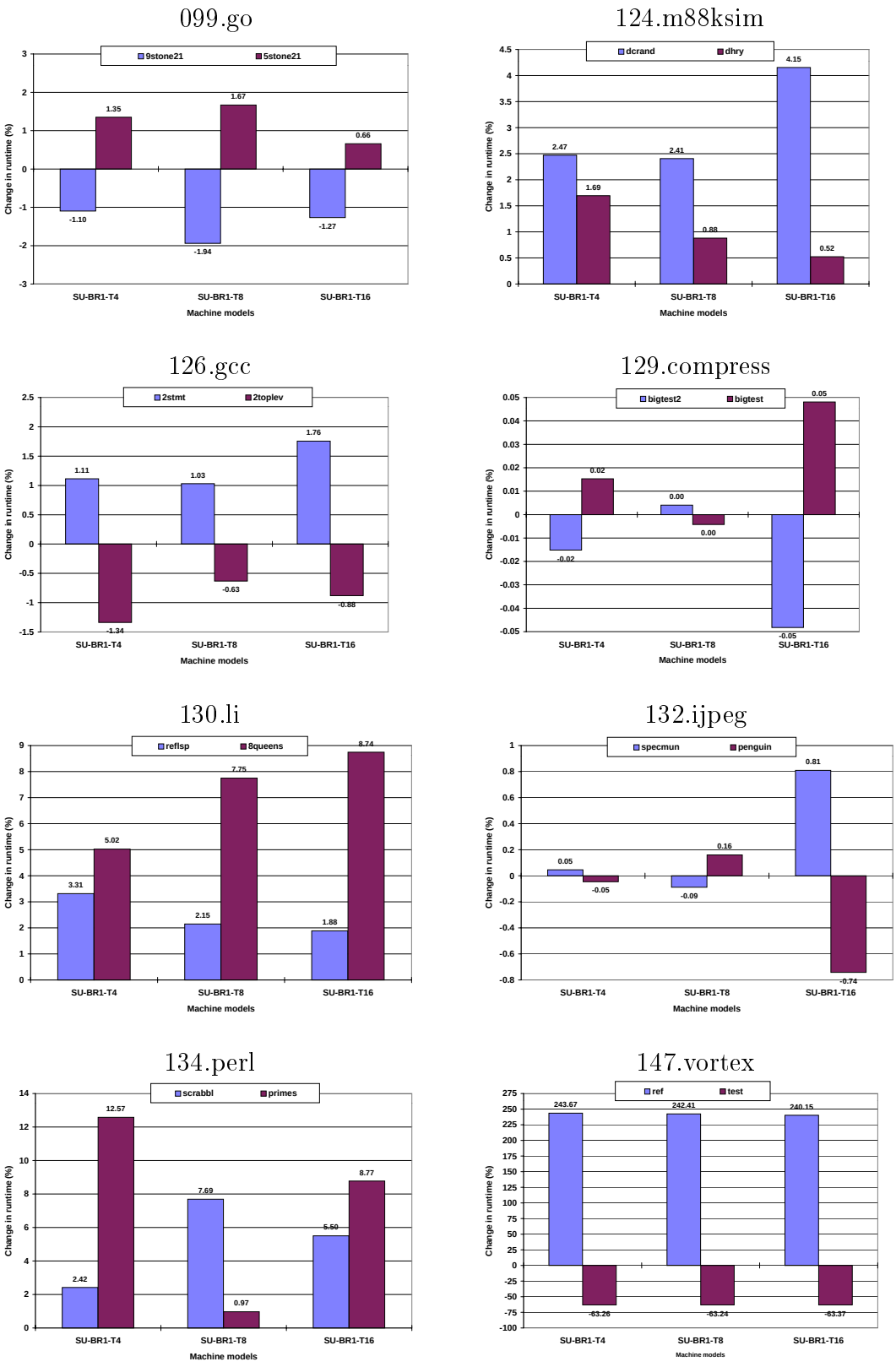


Figure 5.3: Results for machine class SU-BR1: performance shift (as %age change in runtime).

indicates that the code could be improved via re-optimization and rescheduling for the input *primes* with substantial gains (best-case reduction of 9.90% in the runtime). Similar behavior is seen in benchmarks *124.m88ksim* and *130.li* as well.

There are three main reasons why this degradation in performance occurs: (1) change in profile of the program across inputs is such that the priority assumptions used by the instruction scheduler are partially inaccurate (which could lead to an increase in the *effective* schedule height), (2) the assumptions made by the superblock formation algorithm (which also uses the profile information) were partially incorrect, thus leading to "unfriendly" superblock formation decisions, and (3) the scheduler sometimes *over-speculates* the operations so much so that they end up taking away valuable machine resources from critical operations, again increasing the effective schedule height. The first two cases are related to profile variation, whereas the third relates more to the machine model variation. Examples of improper superblock formation (from the ultimate performance point-of-view) have been in shown elsewhere in the literature (see [68]). To reverse these decisions, superblock *dismantling* and *reformation* may have to be performed. These global transformations are potentially expensive and are beyond the scope of this thesis. Another manner in which this problem could be resolved is via the use of *treegions* [35]. Treegions will be discussed in greater depth later in this section. Examples where aggressive speculation decisions made without the knowledge of the machine model (Case 3 above) can degrade the performance are documented in [16].

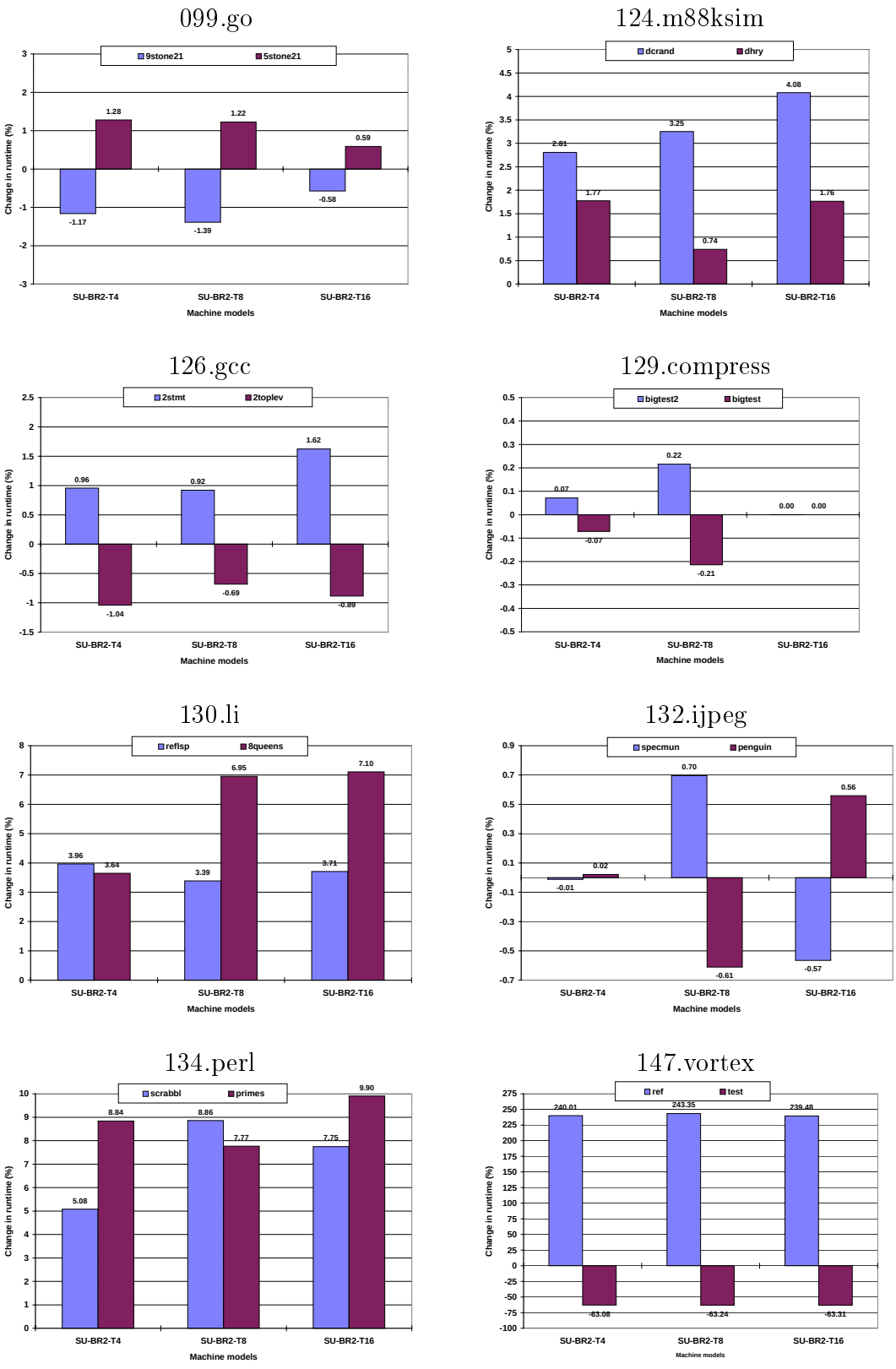


Figure 5.4: Results for machine class SU-BR2: performance shift (as %age change in runtime).

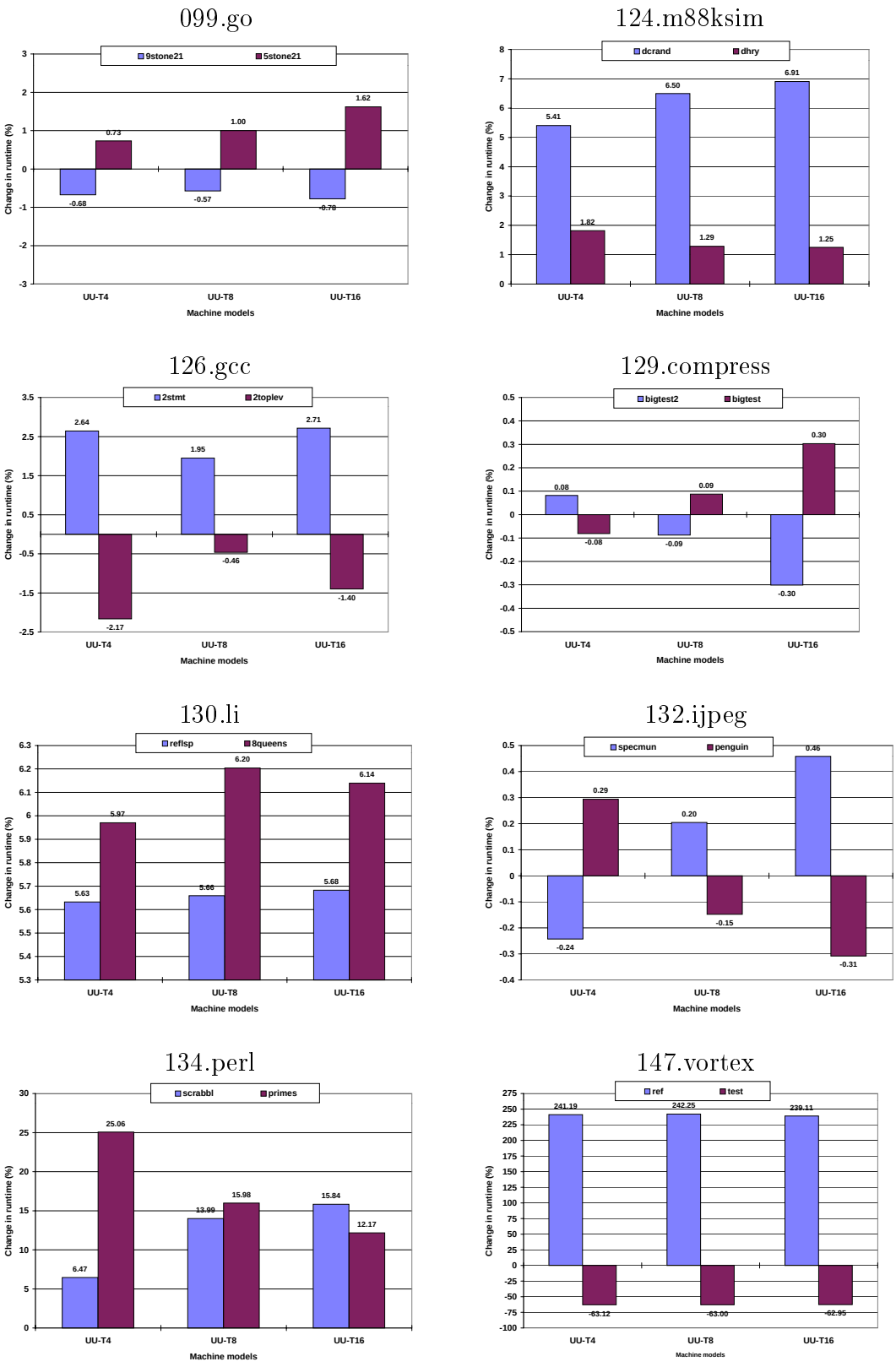
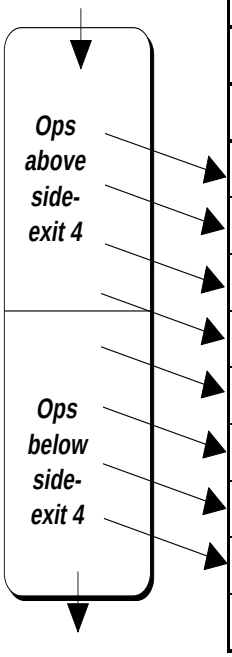


Figure 5.5: Results for machine class UU: performance shift (as %age change in runtime).

As an example of Case 1 above, execution frequency measurements for a superblock (cb 4) from function `xlygetvalue` in the benchmark *130.li* is presented in Figure 5.6[1]. Cb 4 has a total of eight side-exits (destinations of which are cb*s*: 8, 6, 14, 6, 15, 6, 16, and 4 respectively), and a *fallthru* (where the control "falls-through" to the next block of code in order). The superblock was entered 383194974 times on input *reflsp* and 4357946 times on input *8queens*. Of these, the number of times the side-exits were *taken* is shown in columns (B) and (D), along with the %age taken frequency shown in columns (C) and (E), respectively.

The side-exit taken-frequencies are important because they are often the basis for priority assignment during instruction scheduling. If a side-exit which has a high taken-frequency, and is dependent on an earlier operation, then it is imperative that the operation be scheduled as early as possible so that the side-exit can be retired quickly. The usefulness of speculation (i.e. *upward code percolation*) is also indicated by taken-frequencies. An operation should be speculated above a side-exit only if the side-exit is less likely to be taken. In that case, the computation performed by the operation is more likely to be useful. Also, when there are several candidate operations for speculation, the one that produces most useful work should have a higher priority over the others. Extending this argument to account for the machine model, an operation which is least likely to hold a valuable machine resource needed

[1] Of the total runtime of the benchmark, it was found that the largest fraction of time was spent in function `xlygetvalue`. It was also found that the superblock (cb 4) makes a substantial contribution to the time spent in the function. An insight into the execution of this superblock is expected to be valuable in understanding the degradation in the performance of *130.li*, and hence the choice for an example.



(A)	(B)	(C)	(D)	(E)	(F)
Inputs ▶	*reflsp*		*8queens*		
entry count	383194974		4357946		
destinations ▼	taken count	freq (%)	taken count	freq (%)	change (%)
cb 8 (side-exit 1)	0	0.00	0	0.00	0.00
cb 6 (side-exit 2)	195108738	50.92	976542	22.41	28.51
cb 14 (side-exit 3)	0	0.00	0	0.00	0.00
cb 6 (side-exit 4)	135080066	35.25	0	0.00	35.25
cb 15 (side-exit 5)	0	0.00	0	0.00	0.00
cb 6 (side-exit 6)	52899692	13.80	1843532	42.30	-28.50
cb 16 (side-exit 7)	0	0.00	0	0.00	0.00
cb 4 (side exit 8)	0	0.00	0	0.00	0.00
cb 6 (fallthru)	106478	0.03	1537872	35.29	-35.26

Figure 5.6: *130.li*: Side-exit frequency details of function `xlygetvalue`, superblock-4.

by a frequently taken side-exit, should have a higher priority over the others. Such heuristics are commonly used during scheduling, and underline the importance of the side-exit taken-frequencies.

Returning to the example at hand, notice the change in taken-frequencies of the three side-exits across the two inputs. In particular, the taken-frequencies for side-exits–2, –4, and –6 have changed substantially. The fallthru frequency has changed as well. If the code was scheduled using any one of the two taken-frequency measurements, it is clear that the scheduling decision that would be made would be substantially different from the other. On input *reflsp* side-exit–4 is more likely taken (about 35.25% *more frequently* than on input *8queens*). The scheduler would discourage speculation from below the side-exit–4, because it would amount to useless computation 35.25% of the time. The code scheduled using the measurements for the input *8queens*, would have, on the contrary, a large degree of speculation w.r.t. the side-exit–4, because the side-exit is not likely to be taken. If the latter code was run on input *reflsp*, a degradation in performance could occur primarily because of the useless speculation and an increased resource contention induced by it.

A number of negative shifts in the performance are evident from Figures 5.3, 5.4 and 5.5. A negative shift indicates an improvement in performance. For example, notice that $\mathcal{C}^{9stone21} = -1.27\%$ for the benchmark *099.go* on SU-BR1-T16 (Figure 5.3). The code that was profiled and scheduled using input *5stone21* actually performed *better* on the other input *9stone21* compared to the code profiled and

scheduled using *9stone21* (*i.e.* $\mathcal{T}_{5stone21}^{9stone21} < \mathcal{T}_{9stone21}^{9stone21}$). Similar behavior is observed in benchmarks *099.go*, *126.gcc*, *129.compress*, *132.ijpeg*, and *147.vortex*. This result is counter-intuitive, and hence a special attempt was made to explain the event(s) that could lead to this result.

One such event is explained using Superblock cb 20 from function `mrglist` in the benchmark *099.go*. The runtime estimates of the superblock indicated that it aids the counter-intuitive result. Two schedules for this superblock, one generated using profile of the input *5stone21* and the other using that of the input *9stone21* are presented in Figures 5.7 and 5.8 respectively. The target machine is SU-BR1-T16. The height of the schedule on input *5stone21* is 37 cycles, whereas the height is 42 cycles for the other schedule. In Figure 5.7, the side-exit taken frequencies corresponding to the two inputs are shown in columns (B) and (D) respectively. Column (E) shows the variations in the taken-frequencies across these inputs. From this information, it can be seen that the side-exit op 287 is being taken more often on input *9stone21* than on input *5stone21* (by about 3.35%). Similar changes are evident for side-exit ops 288, 301, and 313 as well. From the discussion explaining the positive shift in performance, one would expect that the schedules would perform worse on the *other* input. An extraneous event, however, has offset this degradation: register allocation for the schedule on *5stone21* is better than that for the schedule on *9stone21*. In particular, observe the number of cycles between side-exit ops 279 and 287 in the two: it is 2 cycles *Vs.* 4 cycles respectively. This has happened because the register

allocation for the schedule on *9stone21* (Figure 5.8) was poor (probably based on the global profile of that input), and the spill-fill ops 2369 (cycle 23) and 2371 (cycle 26) stretched the height between the side-exits by 2 cycles. Similarly, the spill op 2372 stretched the height between side-exit ops 313 and 319 by two cycles. The total height increase was 5 cycles (all due to the register allocation decisions), which is the same as the difference in the heights of schedules in Figures 5.7 and 5.8. From the superblock entry counts on input *9stone21*, it then follows that the runtime of the schedule in Figure 5.7 would be $8923993 * 37 = 330187741$ cycles, while that on the scheduled in Figure 5.8 would be $8923993 * 42 = 374807706$ cycles. This confirms the observation made from the experiments that for the machine SU-BR1-T16, $\mathcal{T}_{5stone21}^{9stone21} < \mathcal{T}_{9stone21}^{9stone21}$. Whether the spill-fills inserted by the register allocated were based on profile information or not remains to be seen.

Benchmark *147.vortex* (an object-oriented DBMS) deserves a detailed analysis because the speedups and slowdowns (-ve and +ve performance shifts) that it exhibits are huge: the average speedup is approx. 60%, while the average slowdown is almost 240%! This behavior can be understood by examining the difference in specifications for the two inputs *ref* and *test*. These are shown in Figure 5.9. On both the inputs, the same databases (`parts.db`, `draw.db`, `emp.db`, and `persons.1k`) are used. However, the parameters for the database query generation module are substantially different. For example, notice the semantics and the values of the parameters `PART_COUNT` (number of parts records to be created), `OUTER_LOOP` (num-

Issue slot	0	1	2	3	4	5	6	7	8	9	10	11	15
Cycle 0	234	237	284	247	274	240	243	281					
Cycle 1	285	248	275	266	241	231	263	297	235	3022			
Cycle 2	282	232	264	298	302	270	277	254	3023	3024	3026		
Cycle 3	267	309	233	303	290	321	271	278	236	238	3025	3031	
Cycle 4	310	291	322	255	258	261	314	251	3028	3035			
Cycle 5	239	305	259	317	3027								
Cycle 6	306	293	318	242	3032								
Cycle 7	294	315	3029										
Cycle 8	268	244	245										
Cycle 9	3030												
Cycle 10	246												
Cycle 11	249	3036											
Cycle 12	250	252	260										
Cycle 13													
Cycle 14	253	269											
Cycle 15	256	272											
Cycle 16	257	262	3033	3034									
Cycle 17													265
Cycle 18	273	286											
Cycle 19	276												
Cycle 20													
Cycle 21	280												279
Cycle 22	283	292											
Cycle 23													
Cycle 24	296												287
Cycle 25	299	304											288
Cycle 26	289	3037											
Cycle 27	308												295
Cycle 28	311	316											300
Cycle 29													301
Cycle 30	320	326											307
Cycle 31	323												312
Cycle 32													313
Cycle 33													319
Cycle 34													324
Cycle 35													325
Cycle 36													327

Branch Operations

Speculated Operations

Superblock entry count on *9stone21* = 8923993

Superblock entry count on *5stone21* = 8371637

	(A)	(B)	(C)	(D)	(E)
	5stone21 taken count	5stone21 taken freq. (%)	9stone21 taken count	9stone21 taken freq. (%)	diff (%)
265	*509436*	*6.09*	*512766*	*5.75*	*0.34*
279	*0*	*0.00*	*0*	*0.00*	*0.00*
287	*3912439*	*46.73*	*3871737*	*43.39*	*3.35*
288	*2033058*	*24.29*	*2313940*	*25.93*	*-1.64*
295	*339657*	*4.06*	*468235*	*5.25*	*-1.19*
300	*917269*	*10.96*	*955379*	*10.71*	*0.25*
301	*366603*	*4.38*	*550422*	*6.17*	*-1.79*
307	*45319*	*0.54*	*46247*	*0.52*	*0.02*
312	*127292*	*1.52*	*68886*	*0.77*	*0.75*
313	*64024*	*0.76*	*89010*	*1.00*	*-0.23*
319	*9339*	*0.11*	*8686*	*0.10*	*0.01*
324	*17106*	*0.20*	*7551*	*0.08*	*0.12*
325	*17754*	*0.21*	*19309*	*0.22*	*0.00*
327	*12341*	*0.15*	*11825*	*0.13*	*0.01*

Figure 5.7: *099.go*: Schedule for function `mrglist`, superblock-20 for machine SU-BR1-T16 generated using the input *5stone21*. Ops are represented by their respective op *id*s, and the *BRANCH* ops are shown in thick boxes. Speculated ops are shaded. *Empty cycles* are shown as dark horizontal bars. The table next to the branches is shows taken-frequency statistics on the two inputs. Notice the substantial change (column E) in branch taken-frequencies for branch ops *287, 288, 295* and *301*. The change indicates profile variation.

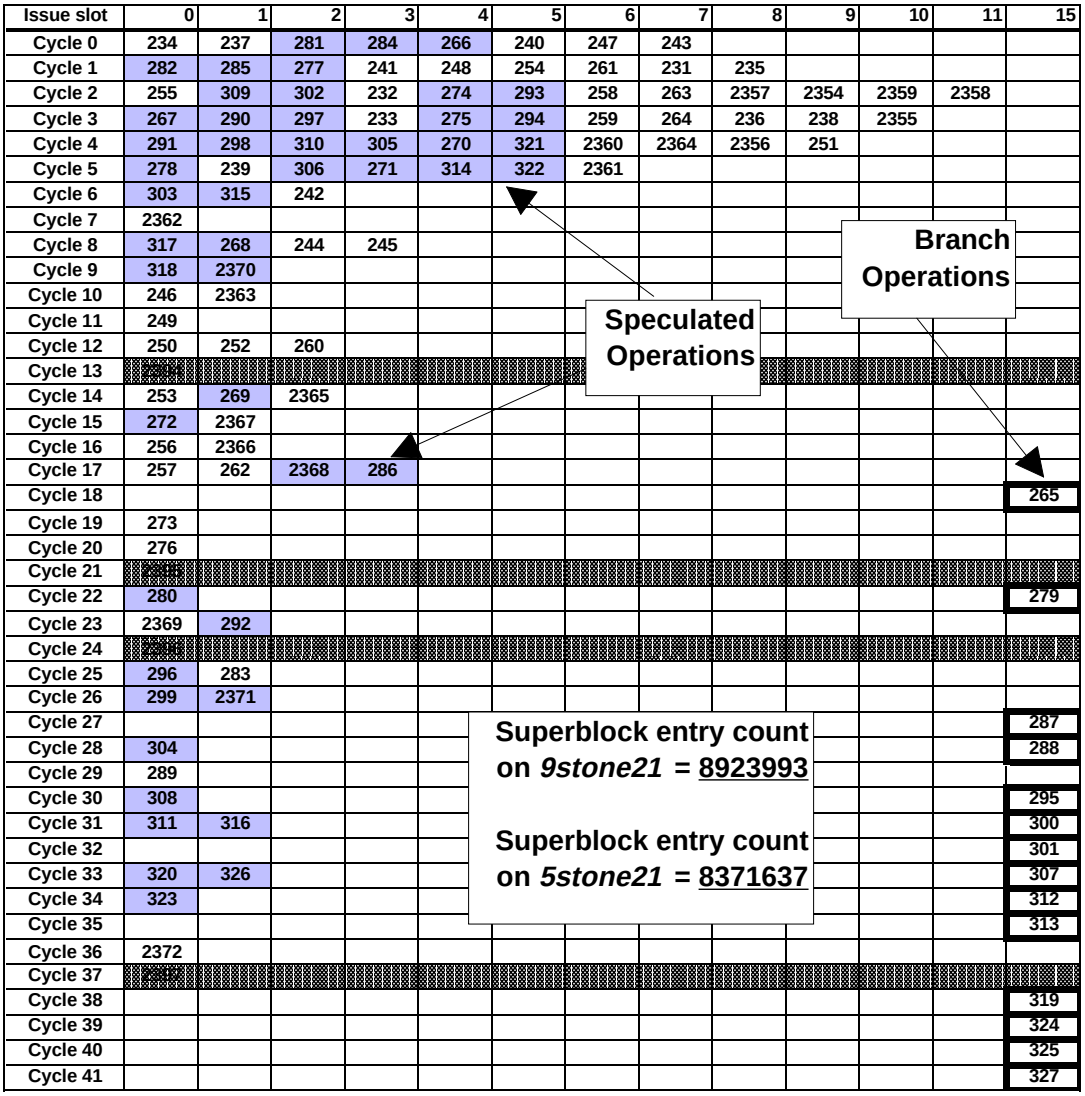


Figure 5.8: *099.go*: Schedule for function `mrglist`, superblock-20 for machine SU-BR1-T16 generated using the input *9stone21*.

(input *ref* specifications)			(input *test* specifications)	
MESSAGE_FILE	vortex.msg		MESSAGE_FILE	vortex.msg
OUTPUT_FILE	vortex.out		OUTPUT_FILE	vortex.out
DISK_CACHE	bmt.dsk		DISK_CACHE	bmt.dsk
RENV_FILE	bendian.rnv		RENV_FILE	bendian.rnv
WENV_FILE	bendian.wnv		WENV_FILE	bendian.wnv
PRIMAL_FILE	vortex.pml		PRIMAL_FILE	vortex.pml
PARTS_DB_FILE	parts.db		PARTS_DB_FILE	parts.db
DRAW_DB_FILE	draw.db		DRAW_DB_FILE	draw.db
EMP_DB_FILE	emp.db		EMP_DB_FILE	emp.db
PERSONS_FILE	persons.1k		PERSONS_FILE	persons.1k
PART_COUNT	16000	<>	PART_COUNT	10000
OUTER_LOOP	1	<>	OUTER_LOOP	2
INNER_LOOP	14	<>	INNER_LOOP	4
LOOKUPS	250		LOOKUPS	250
DELETES	8000	<>	DELETES	500
STUFF_PARTS	8000	<>	STUFF_PARTS	500
PCT_NEWPARTS	0	<>	PCT_NEWPARTS	50
PCT_LOOKUPS	0	<>	PCT_LOOKUPS	25
PCT_DELETES	0	<>	PCT_DELETES	50
PCT_STUFFPARTS	0	<>	PCT_STUFFPARTS	100
TRAVERSE_DEPTH	3	<>	TRAVERSE_DEPTH	5
FREEZE_GRP	1		FREEZE_GRP	1
ALLOC_CHUNKS	10000		ALLOC_CHUNKS	10000
EXTEND_CHUNKS	5000		EXTEND_CHUNKS	5000
DELETE_DRAWS	1		DELETE_DRAWS	1
DELETE_PARTS	0		DELETE_PARTS	0
QUE_BUG	1000		QUE_BUG	1000
VOID_BOUNDARY	67108864		VOID_BOUNDARY	67108864
VOID_RESERVE	1048576		VOID_RESERVE	1048576

Figure 5.9: Differences between the specifications for inputs *ref* and *test* used with *147.vortex*. The differences are shown using a '<>' marker.

ber of outer loops in the query generation module: 1 *vs.* 2), `INNER_LOOP` (number of inner loops: 14 *vs.* 4), `DELETES` (number of parts records deleted per inner loop: 8000 *vs.* 500), `STUFF_PARTS` (number of parts records to be added per inner loop: 8000 *vs.* 500), and `TRAVERSE_DEPTH` (number of "traversals" / "reversals": 3 *vs.* 5). The values are larger/more rigorous on the *ref* specification probably because the input (as per the benchmarking process required by SPEC) is used in the comparison of various hardware platforms. On the other hand, input *test* is not as rigorous, but is used merely to test the correctness of code controlled by all the parameters[2]. These observations lead to two conclusions: (1) input *test* miserably fails to represent the input *ref* (leading to a slowdown of approx. 240% seen on *ref*), and (2) input *ref* exercises various section of *147.vortex* code such that its profile results in a better runtime (by about 60%) for input *test* than obtained using the profile of *test* itself.

Benchmarks *129.compress* and *132.ijpeg* show a minor or no change in performance across the various cases considered. This can be explained as follows. *132.ijpeg* is a program for compression and decompression of graphics (picture files in *PPM* format). The core of the benchmark consists of the following tasks: *forward DCT* and *reverse DCT* used for compression and decompression respectively[3]. These algorithms treat portions of the image as a matrix, and using a pre-computed matrix of co-efficients perform transform the image into a (lossy) compressed representation, or

[2]Note that the `PCT_....` parameters all have a 0 value for input *ref*, but a non-zero value for input *test*. This means that the `PCT_....` parameters are unused in *ref* and are tested for correctness in *test*.

[3]This implementation is based on [69].

back into a PPM (uncompressed) representation. Both the forward and reverse transform have a very regular control structure (similar to that of matrix multiplication). As an example, vital elements of routine `jpeg_fdct_...` are shown in Figure 5.10. The SPEC-supplied wrapper for the benchmark which controls the computations, constitutes most of the control structure in the benchmark. The parameters that determine the nature of control are: (1) `compression_quality` (varies from 90 to 10 in steps of 10), (2) `optimize_coding` (either 0 or 1), and (3) `smoothing_factor` (varies from 90 to 10 in steps of 10). These values are the same across both the inputs used in this study. This outermost control structure is shown in Figure 5.11. With the parameters shown above, the benchmark executes compression and decompression of the same image exactly 128 times. Some parts of the control logic determine if the requirements such as `image quality` and `smoothing factor` have been met. Other even smaller parts consist of parameter checking, configuration tests. In conclusion, the benchmark does not include much control-intensive computation. All the parameters and the configuration are the same across both the inputs, leaving little scope for profile variations across the inputs.

Similar observations have been made about *129.compress*, which performs Lempel-Ziv [70] [71]–style character compression/decompression. The core of the program is a loop which consumes pseudo-randomly generated data (by the input-generator wrapper supplied by SPEC). The execution profile of the program within the loop does not change with various sets of data, leading to little or no profile variations.

```
void jpeg_fdct_.... (image data)
{
  ........ /* variable declarations */
  /* Pass 1: process rows. */
  dataptr = data;
  for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
    /* even part ... */
    .....
    /* odd part */
    .....
  }
  /* Pass 2: process columns.... */
  for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
    /* even part ... */
    .....
    /* odd part */
    .....
  }
}
```

Figure 5.10: Routine `jpeg_fdct_....` in *132.ijpeg*.

```
for (compression_quality=90; compression_quality>10; compression_quality -= 10)
  for (smoothing_factor=90; smoothing_factor>10; smoothing_factor -= 10) {
     optimize_coding=0;
     execute_compression_and_decompression ();
     optimize_coding = 1;
     execute_compression_and_decompression ();
  }
}
```

Figure 5.11: SPEC-supplied control structure in *132.ijpeg* (approximate).

Detailed nature of the benchmark is not discussed here due to space limitations.

Section 5.1.2 expressed the intuition that as machine models become wider and/or more general, there will be a reduction in the magnitude of performance shift. Careful observation of Figures 5.3, 5.4, and 5.5 indicates that the actual behavior of the scheduled code does not always match the intuition (there is no consistency in the cases where the intuition is holds partially). This can be attributed to the instruction scheduler used in these experiments (described by Bringmann in [15]). The scheduler implementation is incapable of controlling the amount of speculation based on the nature/availability of machine resources. A scheduling heuristic such as the *speculative hedge* [16] can be useful in achieving the required level of control, and should show reduced performance shifts as machines become wider/more general.

The conclusions drawn in this section are based solely on the observations made using the set of inputs used in this study. The benchmarks could behave in a vastly different manner with a different set of inputs, thus rendering these conclusions void. It is expected however, that the behavior observed would not be uncommon; only its degree would vary. Based on the current results, one can opine that there exists a case for object-code rescheduling to improve the performance of SPECInt95 benchmarks *134.perl*, *130.li*, *124.m88ksim*, and *147.vortex*.

5.1.5 Discussion: Superblocks and Treegions

The experiments have shown that the optimized superblock code exhibits a potential for object-code evolution across changing inputs. The following discussion contrasts and compares suitability of superblocks to *treegions* [35] from the point of view of object-code evolution.

Superblocks [11] are examples of linear code regions with a single entry in the code (at the beginning of the superblock) and (possible) multiple side-exits from the body of the superblock. The superblock formation process is based on profile-guided decisions of *tail duplication*. This process is shown in Figure 5.12. By very nature of superblocks, all speculation can be performed within the linear body of the superblock. While this speculation is useful as long as the pattern of execution confines itself to the superblock. Any side-exits taken potentially harm the performance in two ways: (1) because speculation w.r.t. the side-exit is useless, and (2) because the superblock is linear, no speculation can be performed from the side-exit *taken path* (*i.e.* the destination of the side-exit) even if machine resources are available to execute code from the taken path speculatively. The evolution of superblock code to adapt to changes in the inputs can take place in two directions: (1) rescheduling the code for the changes in side-exit taken frequencies, and (2) superblock dismantling and re-formation to include the erstwhile side-exit taken path into the new superblock. The second alternative is potentially an expensive transformation than the first, and is not likely the best way to gain performance via object-code evolution.

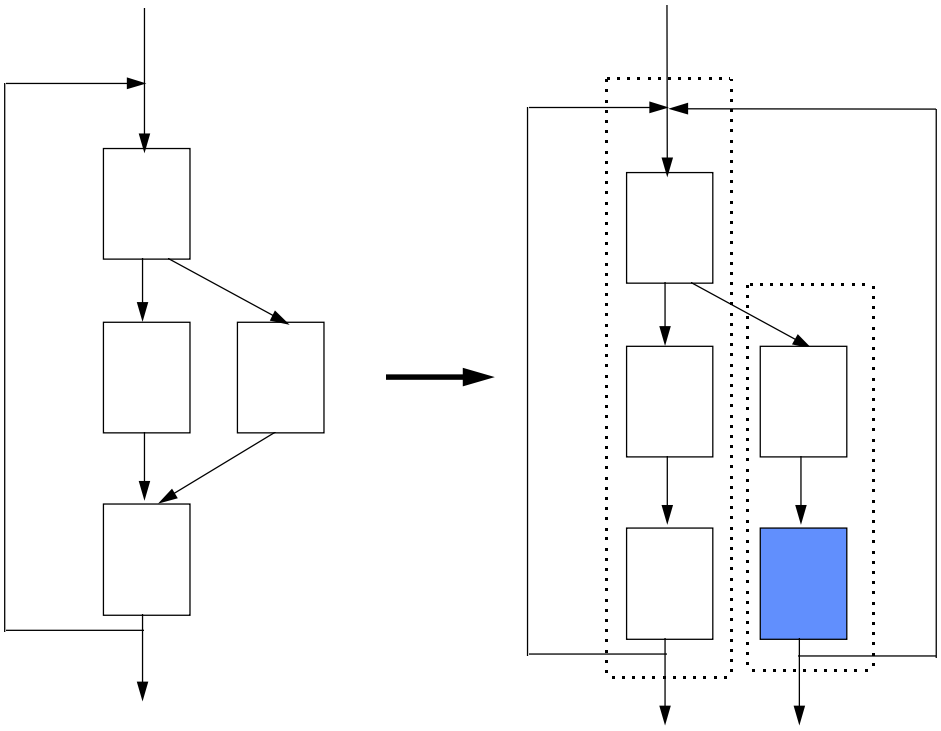


Figure 5.12: Superblock formation: the shaded block of code is the duplicated tail. Boundaries of the two superblocks are shown by dotted boxes.

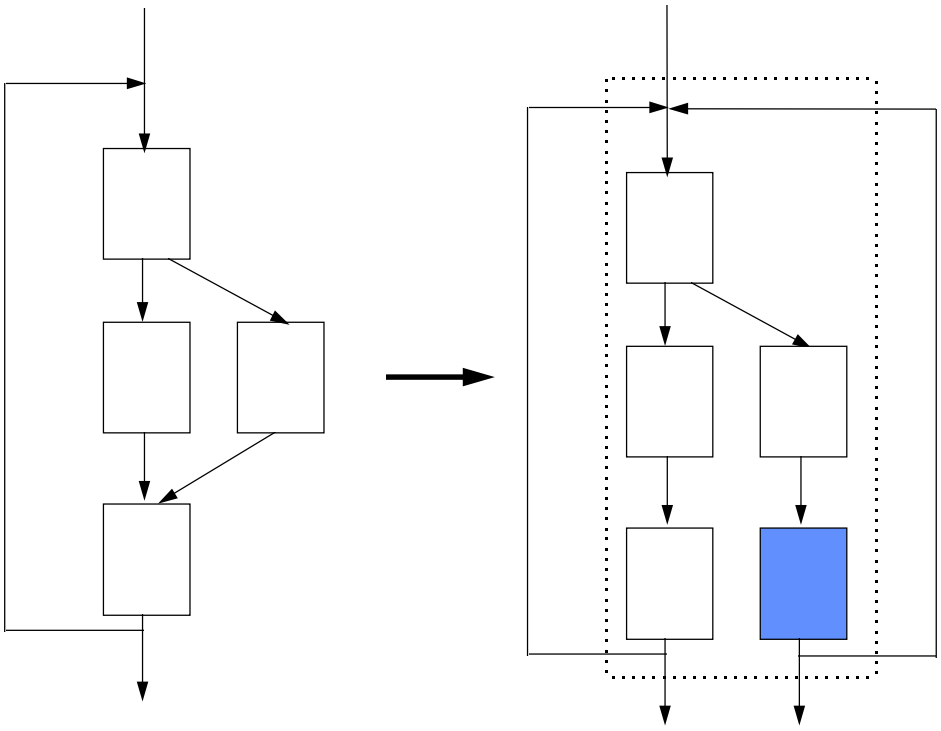


Figure 5.13: Treegion formation: the shaded block of code is the duplicated tail. Treegion boundary is shown by the dotted box.

These problems can be effectively addressed using a *treegion*. The treegion formation process is also profile-based, and is shown in Figure 5.13. Treegions are a type of non-linear regions. One can speculate operations from both the *fallthru* path and the *taken* path of an exit, using a treegion. This means that the treegions can take better advantage of the available machine resources. Treegions are better than superblocks because treegion dismantling and re-formation is not necessary to optimize along the fallthru *or* the taken path: both the fallthru and the taken paths are already available as parts of a treegion and can be effectively used in the rescheduling transformation.

5.2 Performance shift: a case study using the LEGO ILP framework

Study of performance shift in treegionized code was performed using the *LEGO* compiler for the TINKER [72] framework. The LEGO compiler is currently under construction and has several passes for ILP optimizations built into it. It expects the input to be in the REBEL format, an intermediate representation language designed at the Hewlett-Packard Laboratories. Some early work which uses the LEGO compiler can be found in [35] [34] [73].

For this study, a subset of the benchmarks presented in Table 5.1.3 was used. Also, two of the three machine classes used in Section 5.1.2, namely `SU-BR1` and `UU` were used. One additional machine model of issue-width 32 was also considered.

The method to obtain performance measurements was similar to that depicted in Figure 5.2, only performed in the LEGO domain. The LEGO compiler currently does not have a way to incorporate profile information from different inputs into the intermediate code, hence the results presented here primarily measure the effect of the Treegion formation algorithm in LEGO [73]. The "global weight" scheduling heuristic was used in the scheduling passes (see [73] for more information on this heuristic).

5.2.1 Results and Analysis

The results of performance shifts for treegionized code for machine class SU-BR1 are shown in Figure 5.14 and those for machine class UU are shown in Figure 5.15. For benchmarks *134.perl*, *126.gcc*, and *132.ijpeg*, on machine class SU-BR1 the performance shift is miniscule. For *099.go*, however, the shift is sizable for narrow machine model SU-BR1-T4. The reason for this is as follows: the benchmark has treegions which are large in size, and the machine issue-width is narrow. In this case, the treegion scheduler operating under the *global weight* heuristic makes certain speculation decisions, which are harmful because the speculated ops take away valuable execution resources from more critical ops in the region. It can be observed that the effect reduces drastically for wider-issue machines. For 16- and 32-wide machines, *099.go* exhibits no performance shift.

The trend of low performance shift continues for the machine class UU, except that now for *099.go*, the situation has improved substantially. The machine model is

lot less restricted than SU-BR1 (all the units are universal units, and can execute any operation– thus creating a large scope for speculation and eliminating competition for resources). *132.ijpeg* exhibit no performance shift. Shift for *134.perl* is noticeable, but low. Based on these results, one may conclude that Treegionized code is resilient to the phenomenon of profile shifts in these programs.

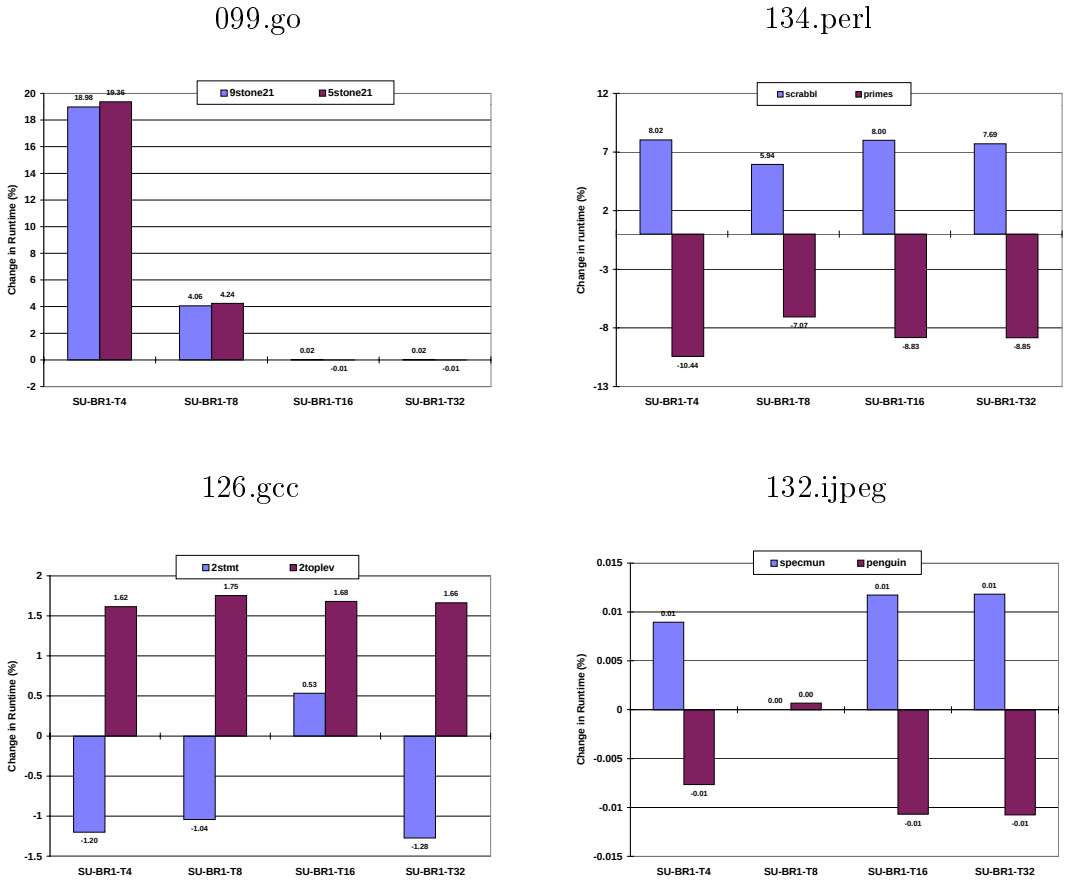


Figure 5.14: Results for machine class SU-BR1: performance shift in Treegions (as %age change in runtime).

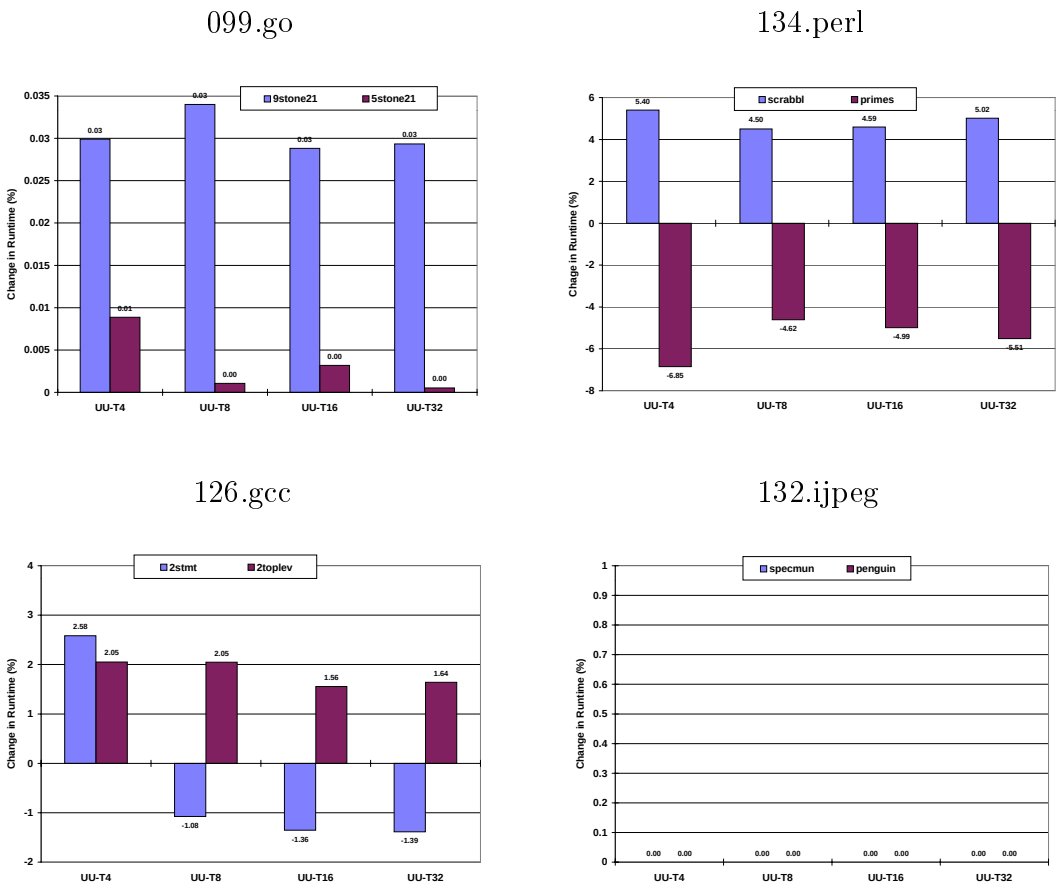


Figure 5.15: Results for machine class UU: performance shift in Treegions (as %age change in runtime).

5.3 Synopsis of previous and related Work

Wall [74] conducted a study which compared the change in the profile information across multiple inputs and the profile estimates using various static estimation techniques. The profile information did not include branch profiles. Wall found that the variations in profile information such as the basic block execution counts was acceptably lower. Fisher and Freudenberger [75] conducted a study of SPECInt92 and other integer branch-intensive programs to measure the effectiveness of profile-based branch prediction. Their conclusion was that the branch directions are largely predictable via the use of profile information. Conte and Hwu [76] measured the validity of profile-driven optimizations in the wake of input variations. The optimizations considered were *trace selection* and *trace layout*. This study found that, for the benchmarks considered, these optimizations are effective within acceptable range. None of the studies attempted to measure the impact of profile variations on the final performance of scheduled code. As this thesis has found out, there is at least a subset of programs which exhibit significant performance shifts due to profile variations. In this regard, the current study complements the work performed by previous researchers.

Chapter 6

Evolutionary Compilation for Performance

6.1 Object-code evolution for performance

Figure 1.1 (Chapter 1) illustrates the process of evolutionary compilation for improvement of performance. Each program object-file ("application") compiled for execution in the EC framework consists of two parts: (1) the program code and initialized data, and (2) a non-loadable object-code annotation segment. When the application is executed, the OS interprets a part of the annotations such as the machine generation-id, the region structure of the code, and nature of branches in the code. If the compiler has provided specific hints about branches, pairs of `LOAD` and `STORE` instructions which could be potentially monitored during run-time etc., they

could be used to program the monitoring hardware.

At the end of each run of an application, the gathered profile information is downloaded into a database. This database is compared with the database stored in the object-code annotation segment by the initial compiler, and the information is used to make any object-code evolution decisions. Alternatively, for long-running programs, assessment of the variation in program profile may be made periodically, and the program may be subjected to transformation at events such as page-faults. An approach to making a rescheduling decision will be described in detail later in this chapter.

The rescheduling decision is made for individual regions in the program. A region of code may be any Single-entry Acyclic Region (SEAR) of code. The region is the scope over which the rescheduling algorithm is applied. At the time of rescheduling, information specific to the region is accessed from the object-code annotation segment. Examples of this information include the *live-out* sets for the side-exits in the region, the operation priorities (pre-computed), or the data- and control-flow graph (DDG) for the region. If the DDG is not available (not saved by the initial compiler), it may be constructed on-the-fly at extra expense, but may be stored for future use.

The decision-making oracle uses a *history* of N previous decisions and the corresponding profiles made in the past. This database can be further refined with annotations and hints about the nature of the code and the control-flow observed at run-time. The emphasis of this chapter is on the rescheduling decision-making oracle,

and the algorithms used in evolution. The rest of this chapter elaborates these two critical aspects of EC in more detail.

6.2 Rescheduling decision

The problem statement for rescheduling decision-making is as follows. Given a history database of profiles for a program, and a sample profile from a recent run of the program, can the following be decided about the program:

1. Can a list of regions of code be generated, such that when the regions are subjected to evolutionary compilation, an improvement in performance can be achieved?

2. If yes, can a cost index be associated with the evolutionary transformations?

The straight-forward way to determine if code evolution would improve the performance of the program is by simply subjecting it to the rescheduling transformation each time a new profile becomes available. If any improvement is seen over the next execution, then the rescheduled code could be cached for further use, else thrown out (deeming the transformations useless). The drawback of this approach is that the overhead of code evolution may be substantial, and may offset its advantages. Instead, a static heuristic-based *oracle*, which returns a figure of merit in favor of/or against rescheduling would be a better solution. Such an oracle was developed as a part of this thesis. Experience gained from the performance shift measurements in

Chapter 5 proved useful in the development of this oracle. Evaluation of the effectiveness of this oracle is presented in Section 6.2.1. Algorithm *EC_Oracle* shown in Figure 6.1 outlines the oracle.

Explanation of various steps in *EC_Oracle* is as follows. After the program reaches the rescheduling decision point, select procedures (`procs`) in the decreasing order of their contribution to the total runtime of the program. These procedures will be considered for possible evolutionary transformations, based on the intuition that improving code quality in areas where the control spends most time would fetch the largest benefits. The ceiling on the number of functions to examine is controlled via a pre-specified limit, and the algorithm is terminated when this limit is reached. In each procedure P, select the regions in which the most runtime was spent. For each selected region R, consider the set of side-exits from the region.

For each side-exit s, F_s denotes the taken-frequency of s in the latest run (obtained by dividing the number of times the side-exit*s* was *taken*, by the number of times control entered region R. Remember that only the regions which are single-entry, multiple-exit are considered suitable for EC). Also, $opcount\_u$ represents the number of operations in the region that are control-dominated by side-exit s, and $opcount\_o$ is the number of ops that control-dominate the side-exit. The ratio $opcount\_ratio$ is computed as $(opount\_u)/(opcount\_u + opcount\_o)$. $opcount\_ratio$ is used to gauge the *positional* importance of side-exit s in region R. Intuitively, if there is a large number of operations that are control-dominated by side-exit s, then a substantial change

```
Algorithm EC_Oracle
begin
   for (next proc P where most runtime was spent) do
   begin
      if (execution frequency of P = 0) then
          quit;
      for (next region R in P where most runtime was spent) do
         begin
            if (execution freq. of R = 0) then
                quit;
            for (each side-exit s in R) do
               begin
                  opcount_u = number of ops in R that are
                     control-dominated by s;
                  opcount_o = number of ops in R that control-dominate s;
                  opcount_ratio = (opcount_u)/(opcount_u + opcount_o);
                  F_s = taken-frequency of s in the last execution of the code.
                  mean = arithmetic mean of taken-frequencies of side-exit s
                     in the last N profiles;
                  sd = Standard Deviation of the taken-frequency of s
                     in the last N profiles;
                  if ((mean - sd * SD_MULTIPLIER <= F_s)
                     and (F_s >= mean + sd * SD_MULTIPLIER)
                     and (opcount_ratio >= OPRATIO_THRESHOLD)
                     and (F_s >= WT_THRESHOLD)) then
                         mark R for rescheduling;
               end
            if (region R is marked for rescheduling) then
               cost_index = cost_index+ number of ops in R;
            if (ceiling on number of regions to examine reached) then
                quit;
         end
      if (ceiling on number of procs to examine reached) then
          quit;
   end
end
```

Figure 6.1: An oracle for making rescheduling decisions.

in F_s would imply that the speculative yield [12] of these operations (a widely used scheduling heuristic) would change substantially. Hence, such a change should lead to a positive rescheduling decision if the value of *opcount_ratio* is sufficiently large. Ceiling on this value is controlled by a parameter called $OPRATIO\_THRESHOLD$.

The other measure used is the taken-frequency F_s itself. If the magnitude of F_s is not significant, any change in it across multiple runs should not lead to a positive rescheduling decision for region R. This is controlled by a parameter called $WT\_THRESHOLD$.

Statistical techniques for measurement of dispersion [77] [78] based on the concept of *standard deviation* are used to identify any side-exits which demonstrate "substantial" change in their taken-frequencies. According to statistical theory, standard deviation of a data-set indicates the "spread" of data from the *arithmetic mean* of the data set. In case of a side-exit of a region, the number of occurrences of the event in which the side-exit is *taken* contribute to the taken-frequency of the side-exit. The taken-frequency values for the side-exit in the profile history database are used to form the basis of statistical measure of dispersion. Arithmetic mean of taken-frequencies of a side-exit over the entire history is taken, which is then used to compute the standard deviation of the latest taken-frequency F_s w.r.t. the mean of the data-set. The mean and the standard deviation are indicated by variables *mean* and *sd* respectively in Figure 6.1.

Whether a given data-point is an *outlier* in the data-set, *i. e.*, differs substantially from the rest of the data about the past behavior of the side-exit, is detected using the following test:

$$(mean - sd * SD\_MULTIPLIER) \leq F_s \quad (6.1)$$

$$(mean + sd * SD\_MULTIPLIER) \geq F_s \quad (6.2)$$

The parameter $SD\_MULTIPLIER$ defines the spread around the mean of the data-set, and acts as a boundary to determine the outliers. If the current value of F_s does not satisfy this test, then it is deemed an outlier, and the region of which the side-exit s is a part, is marked for rescheduling. If all the side-exits in a region pass all the above tests, then it is decided that rescheduling the region is unnecessary, based on the profile information available.

Values of the three parameters ($OPRATIO\_THRESHOLD, WT\_THRESHOLD$, $SD\_MULTIPLIER$) are key to the accuracy of the prediction of this oracle. There are several ways the values can be determined. A set of seed values could be supplied by the initial compiler, based on the actual profile differences observed during initial compile-profile-compile sequences. Alternatively, the EC framework could use a set of seeds for different classes of programs, based on user experiences. Third, if EC is invoked explicitly by a user to optimize his/her runtime environment, then the values made available from such invocations can be used to make further decisions.

The cost index (variable *cost_index*) for code evolution is directly related to the number of operations to be rescheduled in a region. The time– or space–complexity of a code evolution transformation can be expressed in terms of the number of operations (RISC instructions) that must be considered during the transformation. Hence this was considered to be a good metric of the cost of code evolution. The EC oracle keeps track of the number of operations in candidate region marked for rescheduling and the total number of ops is used as a cost index of code evolution. If the number of operations to be rescheduled is large, then the EC framework may elect to perform rescheduling of only a fraction of the set of regions which are candidates for rescheduling.

6.2.1 Evaluation of the EC Oracle

The following benchmarks from the SPECInt95 suite were used for the evaluation of the oracle: (1) *130.li*, a LISP compiler, (2) *134.perl*, the Pattern Extraction and Report Language, commonly used to simplify system administration and other text handling tasks, (3) the game of *099.go*, and (4) *126.gcc*, the GNU C Compiler. These input sets used in the evaluation are described in Table 6.1. The experiments to evaluate the oracle were conducted in the following manner. Benchmark code used in experiments for measurement of performance shift (described in Section 5.1) was used. For each benchmark, the effectiveness of the oracle in selecting regions which reduce the *slowdown* across inputs was the key indicator of merit. Machine

Table 6.1: Benchmarks and inputs used to evaluate the EC Oracle.

Benchmark	Primary Inputs	Description	Secondary Input(s)	Description
130.li	`8queens`	SPEC test	`takr`	`reflsp` component
	`reflsp`	SPEC ref	`boyer`	`reflsp` component
			`triang`	`reflsp` component
134.perl	`primes`	SPEC test	`numberlines`	Utility [79]
	`scrabbl`	SPEC test	`capitalize`	Utility [79]
			`linelength`	Utility [79]
099.go	`5stone21`	SPEC ref	`2stone9`	SPEC test
	`9stone21`	SPEC ref		
126.gcc	`2stmt`	SPEC ref	`2cccp`	SPEC ref
	`2toplev`	SPEC ref	`2expr`	SPEC ref
			`2recog`	SPEC ref

classes SU-BR1, SU-BR2, and UU, as described in Figure 5.1.2 were used. The *primary* inputs in Table 6.1 were used to evaluate the oracle. Profile information from each of the *secondary* inputs (3)–(5) was used only as a component of profile history database, hence these inputs are called *secondary* inputs. As described in Section 6.2, the profile history database forms the statistical basis for the comparison of new profile information. Several experiments were run, for each benchmark, to observe the improvement in the runtime of each primary input on the code optimized for the other primary input, and when each of the secondary inputs was used as statistical basis for the oracle. Results for benchmarks *134.perl* and *130.li* are shown in Tables 6.2–6.8.

In the leftmost column of Table 6.2, the 3-tuples show the values of parameters ($WT\_THRESHOLD$, $OPRATIO\_THRESHOLD$, and $SD\_MULTIPLIER$). In the second column, the result of rescheduling regions selected by the oracle is shown,

Table 6.2: *134.perl*:EC Oracle outcome on input `primes` on machine SU-BR1-T4. The base code was compiled and optimized for input `scrabbl`.

primary: `primes`, secondary: `capitalize`		old slowdown(%)=12.57	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1, 0.25, 2.0	11.63	2409	55
0.1, 0.25, 1.0	11.21	2722	67
0.1, 0.25, 0.5	11.21	3975	110
primary: `primes`, secondary: `linelength`			
0.1, 0.25, 2.0	11.63	2447	64
0.1, 0.25, 1.0	11.21	2704	71
0.1, 0.25, 0.5	11.21	3932	102
primary: `primes`, secondary: `numberlines`			
0.1, 0.25, 2.0	11.63	2952	71
0.1, 0.25, 1.0	11.21	3197	79
0.1, 0.25, 0.5	11.21	3982	103

Table 6.3: *134.perl*: EC Oracle outcome on input `primes` on machine UU-T4. The base code was compiled and optimized for input `scrabbl`.

primary: `primes`, secondary: `capitalize`		old slowdown(%)=25.06	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1, 0.25, 2.0	19.23	2648	55
0.1, 0.25, 1.0	19.23	2974	67
0.1, 0.25, 0.5	19.62	4259	110
primary: `primes`, secondary: `linelength`			
0.1, 0.25, 2.0	19.23	2676	64
0.1, 0.25, 1.0	19.23	2944	71
0.1, 0.25, 0.5	19.62	4216	102
primary: `primes`, secondary: `numberlines`			
0.1, 0.25, 2.0	19.62	3201	71
0.1, 0.25, 1.0	19.62	3455	79
0.1, 0.25, 0.5	19.62	4268	103

Table 6.4: *134.perl*: EC Oracle outcome on input `scrabbl` on machine UU-T4. The base code was compiled and optimized for input `primes`.

primary: `scrabbl`, secondary: `capitalize`		old slowdown(%)=6.47	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1, 0.25, 2.0	6.08	3390	95
0.1, 0.25, 1.0	6.08	3570	105
0.1, 0.25, 0.5	6.08	3776	117
primary: `scrabbl`, secondary: `linelength`			
0.1, 0.25, 2.0	6.20	3113	92
0.1, 0.25, 1.0	6.20	3362	105
0.1, 0.25, 0.5	6.22	3893	119
primary: `scrabbl`, secondary: `numberlines`			
0.1, 0.25, 2.0	6.20	3287	97
0.1, 0.25, 1.0	6.20	3328	102
0.1, 0.25, 0.5	6.22	3888	120

and is indicated as "new slowdown". This is the figure of merit, which determines the effectiveness of the oracle. Notice that the field "old slowdown" in the first row of the table indicates the slowdown observed (12.57%), when *134.perl* code compiled and scheduled for input `scrabbl` was run on input `primes`. The new slowdown is 11.21% in most cases, or 11.63% in some cases. A reduction in slowdown means that the regions chosen to be rescheduled yielded performance improvement. Column 3 in Table 6.2 (titled `numops`) indicates the value of $cost\_index$, which is the total number of Ops that must be rescheduled to achieve this performance improvement. In the last column of the table, the number of regions picked by the oracle is shown.

In this experiment, the value of the $SD\_MULTIPLIER$ parameter was varied at values 2.0, 1.0, and 0.5. The number of regions picked under each of the values is shown. The number of regions for $SD\_MULTIPLIER = 0.5$ is the largest (as

expected), but it does not much help improve the performance. This means that a higher value of the multiplier would be fine, which would lead to less cost, but not a substantial drop in performance improvement. It was observed that the greed of the standard deviation heuristic in picking the regions was effectively controlled by the other two parameters. After extensive experimentation with the two benchmarks under consideration, it was found that the values of $WT\_THRESHOLD = 0.1$, and $OPRATIO\_THRESHOLD = 0.25$ were best in controlling the greed. It was also found that the value of 2.0 for the standard deviation multiplier made the best choices under the most circumstances, while also keeping the number of ops to be rescheduled, relatively low. Hence, the recommended set of values of the three parameters in this oracle, for the benchmarks considered here, is: (0.1, 0.25, and 2.0).

Another important point to notice in Table 6.2 is that varying the secondary input did not make substantial difference in the results in performance improvement. But this made a large difference in the cost of rescheduling: observe that for secondary input `capitalize`, the number of ops to be rescheduled is the smallest (2409), yet the performance improvement is the same as that for inputs `linelength` and `numberlines`. Thus the cost overhead of the code evolution transformation also largely depends on the nature of the secondary input(s) used to form the statistical basis. No concrete conclusion can be drawn for a general case, however, based on these results, because the sequence and the types of inputs used in real-world situations can be vastly different.

Similar observations about the effectiveness of the oracle can be made based on the results shown for for primary input `primes` for other machine models, and for primary input `scrabbl`.

Table 6.3 shows very strong performance improvement on primary input `primes`: the slowdown decreased by more than 5% for all of the secondary inputs. The number of ops, and hence the cost of rescheduling was at a minimum for standard deviation multiplier of 2.0, same as observed before.

Results for primary input `scrabbl` on machine UU-T4 are shown in Table 6.4.

Table 6.5: *130.li*: EC Oracle outcome on input `reflsp` on machine SU-BR2-T16. The base code was compiled and optimized for input `8queens`.

primary: `reflsp`, secondary: `boyer`		old slowdown(%)=3.70	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1, 0.25, 2.0	3.72	1062	39
0.1, 0.25, 1.0	3.72	1145	42
0.1, 0.25, 0.5	3.72	1490	59
primary: `primes`, secondary: `takr`			
0.1, 0.25, 2.0	3.72	1124	41
0.1, 0.25, 1.0	3.72	1376	45
0.1, 0.25, 0.5	3.74	1587	61
primary: `primes`, secondary: `triang`			
0.1, 0.25, 2.0	3.71	1124	38
0.1, 0.25, 1.0	3.71	1230	44
0.1, 0.25, 0.5	3.72	1607	59

Results generally similar to those shown above for *134.perl* were obtained for benchmark *130.li*, and are shown in Tables 6.6–6.8.

The oracle does not yield an improvement in slowdown in all cases, however. One such case in point is the performance of primary input `reflsp` on machine SU-BR1-

Table 6.6: *130.li*: EC Oracle outcome on input `reflsp` on machine SU-BR1-T4. The base code was compiled and optimized for input `8queens`.

primary: `reflsp`, secondary: `boyer`		old slowdown(%)=3.31	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1,0.25,2.0	1.81	938	39
0.1,0.25,1.0	1.81	1009	42
0.1,0.25,0.5	1.94	1319	59
primary: `reflsp`, secondary: `takr`			
0.1,0.25,2.0	1.72	1001	41
0.1,0.25,1.0	1.75	1220	45
0.1,0.25,0.5	1.85	1406	61
primary: `reflsp`, secondary: `triang`			
0.1,0.25,2.0	1.94	971	38
0.1,0.25,1.0	1.94	1068	44
0.1,0.25,0.5	1.93	1389	59

T16 shown in Table 6.5. The slowdown increases by about 0.1–0.3%. The machine model is very wide– a feature that can withstand pressures of over-speculation by the scheduler. In that light, the $OPRATIO\_THRESHOLD$ heuristic, which is designed to keep over-speculation under control, suppresses aggressive rescheduling decisions which are necessary in this case. Relaxing the OPRATIO constraint is expected to reduce the effect.

To verify this, extensive experiments were run varying both the parameters: $OPRATIO\_THRESHOLD$ (from 0.05 to 0.25 in steps of 0.05) and $WT\_THRESHOLD$ (from 0.02 to 1.0 in steps of 0.02). The results of these experiments are shown in Table 6.9. Based on this data, the hypothesis that the parameter $OPRATIO\_THRESHOLD$ is too strong in controlling the decision, is incorrect for this case. In all the variations of parameters, the slowdown was found to be higher than the original slow-

Table 6.7: *130.li*: EC Oracle outcome on input `reflsp` on machine SU-BR1-T8. The base code was compiled and optimized for input `8queens`.

primary: `reflsp`, secondary: `boyer`		old slowdown(%)=2.15	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1,0.25,2.0	1.67	994	39
0.1,0.25,1.0	1.67	1072	42
0.1,0.25,0.5	1.64	1401	59
primary: `reflsp`, secondary: `takr`			
0.1,0.25,2.0	1.63	1044	41
0.1,0.25,1.0	1.63	1281	45
0.1,0.25,0.5	1.65	1478	61
primary: `reflsp`, secondary: `triang`			
0.1,0.25,2.0	1.64	1046	38
0.1,0.25,1.0	1.64	1147	44
0.1,0.25,0.5	1.64	1491	59

down (3.70%).

Further examination revealed that the increase in slowdown occurs because of the differences in *representativeness* of profiles of execution on inputs `8queens` and `reflsp`. In particular, there is a number of procedures in *130.li* code that are not executed on input `8queens`, yet considerable number of cycles are spent in them when input `reflsp` is used. A sample of this set of procedures is shown in Table 6.10. Due to these behavioral differences, when code was optimized for input `8queens`, none of these procedures was optimized – for lack of profile information. When the EC oracle was invoked, the profile information on the base code indicated the same lack of information, and hence the oracle chose not to make a decision on rescheduling any regions in these procedures.

This may seem like a drawback of the EC oracle, but it is necessary that the

Table 6.8: *130.li*: EC Oracle outcome on input `reflsp` on machine UU-T4. The base code was compiled and optimized for input `8queens`.

primary: `reflsp`, secondary: `boyer`		old slowdown(%)=5.63	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1,0.25,2.0	5.22	990	39
0.1,0.25,1.0	5.22	1067	42
0.1,0.25,0.5	5.22	1406	59
primary: `reflsp`, secondary: `takr`			
0.1,0.25,2.0	5.22	1058	41
0.1,0.25,1.0	5.22	1300	45
0.1,0.25,0.5	5.21	1502	61
primary: `reflsp`, secondary: `triang`			
0.1,0.25,2.0	5.22	1036	38
0.1,0.25,1.0	5.22	1138	44
0.1,0.25,0.5	5.22	1492	59

oracle decide to defer making the decision for these procedures. The alternative, at this point, is to subject them all to rescheduling, based on the latest profile, which certainly is not an educated decision. If the oracle chooses to subject *all* such procedures to code evolution, the overhead of the transformation could be excessive. After a few executions of the program, when some more profile history becomes available, a more educated decision may be made about rescheduling them.

Results for the benchmarks *126.gcc* and *099.go* are shown in Tables 6.11–6.16. The results for *126.gcc* are very encouraging: the oracle was able to pick regions so that substantial improvements were observed. For example, for machine SU-BR1-T4, the slowdown decreased from 1.11% to 0.91% (best improvement). In only one case, when the secondary input was `2cccp` and the *SD_MULTIPLIER* value of 2.0, the slowdown increased by about 0.2%. This is due to the overly restrictive nature

Table 6.9: *130.li*: EC Oracle outcome on input `reflsp` on machine SU-BR2-T16: large number of experiments were run, varying the $OPRATIO\_THRESHOLD$ and $WT\_THRESHOLD$ parameters. The base code was compiled and optimized for input `8queens`.

primary: `reflsp`, secondary: `boyer`		old slowdown(%)=3.70	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.02, 0.05, 2.0	3.72	1298	49
0.02, 0.1, 2.0	3.72	1298	49
0.02, 0.15, 2.0	3.72	1197	45
0.02, 0.2, 2.0	3.72	1129	44
0.04, 0.05, 2.0	3.72	1230	48
0.04, 0.1, 2.0	3.72	1230	48
0.04, 0.15, 2.0	3.72	1129	44
0.04, 0.2, 2.0	3.72	1129	44
0.06, 0.05, 2.0	3.72	1230	48
0.06, 0.1, 2.0	3.72	1230	48
0.06, 0.15, 2.0	3.72	1129	44
0.06, 0.2, 2.0	3.72	1129	44
0.08, 0.05, 2.0	3.72	1230	48
0.08, 0.1, 2.0	3.72	1230	48
0.08, 0.15, 2.0	3.72	1129	44
0.08, 0.2, 2.0	3.72	1129	44
primary: `reflsp`, secondary: `takr`			
0.02, 0.05, 2.0	3.72	1522	54
0.02, 0.1, 2.0	3.72	1493	53
0.02, 0.15, 2.0	3.72	1402	51
0.02, 0.2, 2.0	3.72	1234	49
0.04, 0.05, 2.0	3.72	1399	51
0.04, 0.1, 2.0	3.72	1370	50
0.04, 0.15, 2.0	3.72	1279	48
0.04, 0.2, 2.0	3.72	1179	47
0.06, 0.05, 2.0	3.72	1390	49
0.06, 0.1, 2.0	3.72	1361	48
0.06, 0.15, 2.0	3.72	1270	46
0.06, 0.2, 2.0	3.72	1170	45
0.08, 0.05, 2.0	3.72	1363	48
0.08, 0.1, 2.0	3.72	1334	47
0.08, 0.15, 2.0	3.72	1243	45
0.08, 0.2, 2.0	3.72	1143	44
primary: `reflsp`, secondary: `triang`			
0.02, 0.05, 2.0	3.72	1579	47
0.02, 0.1, 2.0	3.72	1550	46
0.02, 0.15, 2.0	3.71	1471	43
0.02, 0.2, 2.0	3.71	1403	42
0.04, 0.05, 2.0	3.72	1511	46
0.04, 0.1, 2.0	3.72	1482	45
0.04, 0.15, 2.0	3.71	1403	42
0.04, 0.2, 2.0	3.71	1403	42
0.06, 0.05, 2.0	3.72	1471	45
0.06, 0.1, 2.0	3.72	1442	44
0.06, 0.15, 2.0	3.71	1363	41
0.06, 0.2, 2.0	3.71	1363	41
0.08, 0.05, 2.0	3.72	1419	44
0.08, 0.1, 2.0	3.72	1313	42
0.08, 0.15, 2.0	3.71	1234	39
0.08, 0.2, 2.0	3.71	1234	39

Table 6.10: *130.li*: Large number of cycles of execution are spent in these procedures on `reflsp`, but not on input `8queens`. The base code was compiled and optimized for input `8queens`.

procedure	cycles on `reflsp`	cycles on `8queens`
`xltest`	41960	0
`xequ`	2472632	0
`xatom`	3150561	0
`xlevmatch`	1849952	0
`xif`	1720872	0
`xself`	2134560	0
`xnconc`	2249040	0
`xrplca`	8116800	0

of the *SD_MULTIPLIER* constant. Once this constraint was relaxed, performance improvement was evident (notice the new slowdown values for *SD_MULTIPLIER* values of 1.0% and 0.5%).

The oracle was not much successful in picking the right regions for 099.go. The reduction in slowdown was not observed, except only in one case (see Table 6.16). One major reason for this failure is the lack of statistical basis to make rescheduling decisions. Unlike the other three benchmarks used as test cases in this evaluation, *099.go* has only one secondary input available. (The benchmark is a canned piece of code from the popular game of 'go'. The game being a commercial product, use of the code for purposes other than benchmarking is prohibited). The input, `2stone9`, is a short input used only for benchmark test purposes by SPEC, and hence is not suitable to form the necessary statistical basis for the EC oracle. Also, from the performance

shift results presented in Chapter 5, the performance of *099.go* does not change substantially across the two primary inputs `5stone21` and `9stone21`. In this light, the benchmark normally would not be subjected to evolutionary transformations. The performance of the oracle on *099.go* presented here should hence be interpreted with these peculiarities of the benchmark and the related input sets.

Table 6.11: *126.gcc*: EC Oracle outcome on input `2stmt` on machine SU-BR1-T4. The base code was compiled and optimized for input `2toplev`.

primary: `2stmt`, secondary: `2expr`		old slowdown(%)=1.11	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1,0.25,2.0	0.91	43968	917
0.1,0.25,1.0	0.95	71933	1513
0.1,0.25,0.5	1.04	106150	2262
primary: `2stmt`, secondary: `2cccp`			
0.1,0.25,2.0	1.31	52223	1082
0.1,0.25,1.0	0.99	76710	1682
0.1,0.25,0.5	1.09	106327	2338
primary: `2stmt`, secondary: `2recog`			
0.1,0.25,2.0	1.07	48341	1014
0.1,0.25,1.0	0.96	72666	1532
0.1,0.25,0.5	1.07	99880	2201

Figures 6.2 and 6.3 show the effect of the overhead of evolution on performance. For benchmarks *134.perl* and *130.li*, the overhead was measured as a per-operation estimate of number of cycles. This was found to be, on the average, 257 cycles. The total overhead estimate for rescheduling transformation was obtained by multiplying this number by the number of operations picked by the EC Oracle ("numops"), which was 2648 in case of *134.perl* (see Table 6.3) and 1001 in case of *130.li* (Table 6.6). The performance of the rescheduled code both with and and without overhead was

Table 6.12: *126.gcc*: EC Oracle outcome on input `2stmt` on machine SU-BR1-T16. The base code was compiled and optimized for input `2toplev`.

primary: `2stmt`, secondary: `2expr`		old slowdown(%)=1.76	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1,0.25,2.0	1.40	50547	917
0.1,0.25,1.0	1.47	82947	1513
0.1,0.25,0.5	1.53	121874	2262
primary: `2stmt`, secondary: `2cccp`			
0.1,0.25,2.0	1.70	60539	1082
0.1,0.25,1.0	1.45	88440	1682
0.1,0.25,0.5	1.50	121548	2338
primary: `2stmt`, secondary: `2recog`			
0.1,0.25,2.0	1.37	55785	1014
0.1,0.25,1.0	1.41	83654	1532
0.1,0.25,0.5	1.49	114503	2201

Table 6.13: *099.go*: EC Oracle outcome on input `5stone21` on machine SU-BR1-T4. The base code was compiled and optimized for input `9stone21`.

primary: `5stone21`, secondary: `2stone9`		old slowdown(%)=1.35	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1,0.25,2.0	1.35	84401	1476
0.1,0.25,1.0	1.39	118569	2143
0.1,0.25,0.5	1.39	154626	2896

plotted as a function of "cost budget". 100% cost equals the estimated number of cycles of overhead per operation (= 257 cycles), whereas 200% budget equals twice the estimate (= 514 cycles). It can be seen that for the two benchmarks, the improvements in performance were substantial, and the overhead of rescheduling, even with the 200% cost is insignificant. The EC oracle made effective choices in the case of the two benchmarks such that the performance improved without incurring large rescheduling overhead.

Table 6.14: *099.go*: EC Oracle outcome on input `5stone21` on machine SU-BR1-T8. The base code was compiled and optimized for input `9stone21`.

primary: `5stone21`, secondary: `2stone9`		old slowdown(%)=1.67	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1,0.25,2.0	2.20	97527	1476
0.1,0.25,1.0	2.20	137437	2143
0.1,0.25,0.5	2.20	179166	2896

Table 6.15: *099.go*: EC Oracle outcome on input `5stone21` on machine SU-BR1-T16. The base code was compiled and optimized for input `9stone21`.

primary: `5stone21`, secondary: `2stone9`		old slowdown(%)=0.66	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1,0.25,2.0	0.88	100476	1476
0.1,0.25,1.0	0.88	141274	2143
0.1,0.25,0.5	0.88	183597	2896

6.3 Rescheduling Algorithms

A variety of rescheduling algorithms can be used for object-code evolution. Some of these algorithms make aggressive use of annotations stored along with the code during initial compilation (object-file annotations are discussed in detail in Chapter 7). Using this information, the algorithms generate high-quality rescheduled code. In the rest of

Table 6.16: *099.go*: EC Oracle outcome on input `5stone21` on machine SU-BR2-T16. The base code was compiled and optimized for input `9stone21`.

primary: `5stone21`, secondary: `2stone9`		old slowdown(%)=0.59	
(WTT,OPT,SDM)	new slowdown(%)	numops	numregions
0.1,0.25,2.0	0.51	100493	1476
0.1,0.25,1.0	0.51	141421	2143
0.1,0.25,0.5	0.51	183633	2896

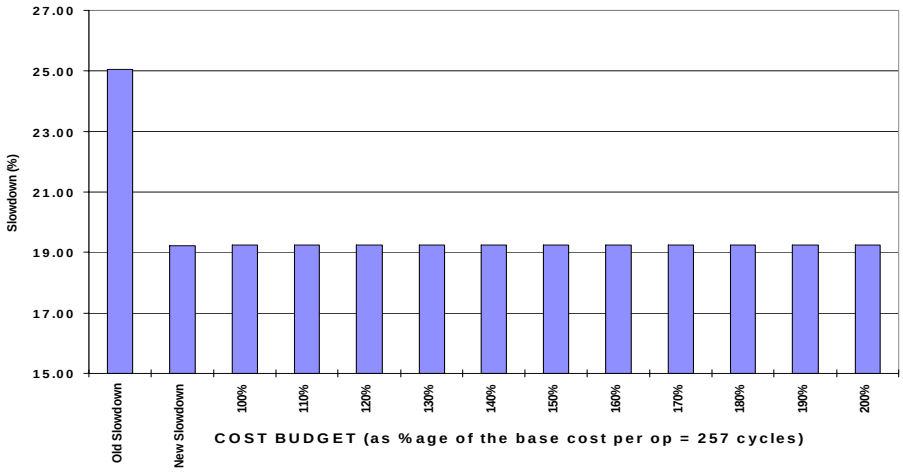


Figure 6.2: Performance of *134.perl* including the overhead of evolution. Primary input = `primes`, secondary input = `capitalize`, number of operations = 2648.

this chapter, two types of rescheduling algorithms are presented: (1) those which use the data dependence graph (DDG) in an implicit manner (*implicit-DDG* algorithms), and (2) those which use the DDG explicitly (*explicit-DDG* algorithms). First the implicit-DDG algorithms are described, followed by the explicit-DDG algorithms. In Chapter 4, an implicit-DDG algorithm was presented as the rescheduling algorithm for Acyclic code. The algorithm and its variations will be revisited in Section 6.3.2

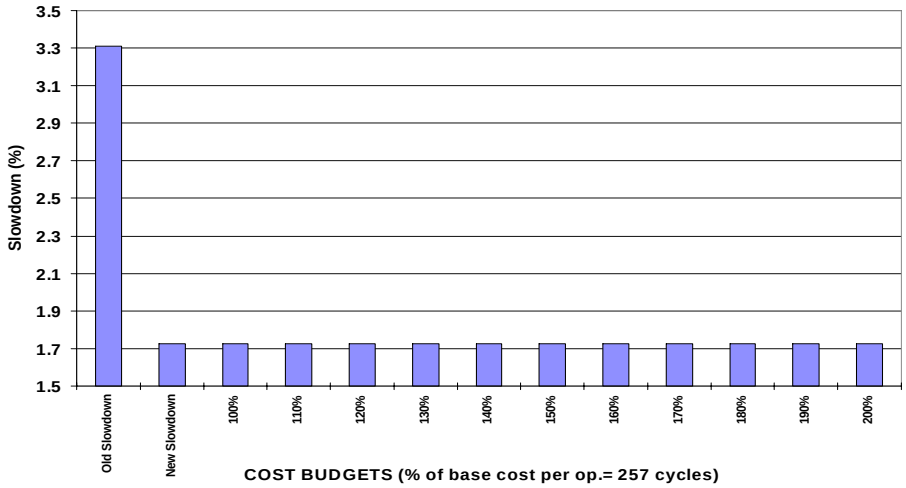


Figure 6.3: Performance of *134.li* including the overhead of evolution. Primary input = `reflsp`, secondary input = `takr`, number of operations = 1001.

of this Chapter.

6.3.1 Terminology

An *operation* is a single RISC instruction. The scope of the code over which rescheduling is performed is called a *region*. A region is a single-entry acyclic region (SEAR), as described in Chapter 2. A *Current schedule* is the existing schedule for the region.

A *New schedule* is the schedule generated as a result of rescheduling. Similarly a *Current machine* is the machine model for which current schedule was constructed, and, a *New machine* is the machine model to which the output of the rescheduler is being targeted.

Each of the algorithms outlined here takes four inputs: (1) the current schedule, (2) the current machine model, (3) the new machine model, and (4) set of priority functions that imposes an ordering over the operations for the purposes of rescheduling. The output of each algorithm is the new schedule.

6.3.2 Implicit DDG Algorithms

The Mercury Algorithm

This algorithm is similar to the well-known instruction scheduling algorithm called Operation Scheduling [80][1]. The main idea of the mercury algorithm is as follows: operations within a region of code are scanned one-at-a-time in a left-to-right, top-down fashion, and each operation is placed in the final schedule in such a way that the data/control dependences and the resource constraints are honored. This algorithm is an implicit DDG algorithm because it builds a partial, in-core DDG structure while it places the operations in the final schedule. The operation placement in the Mercury algorithm is greedy; an operation is hoisted as high up as possible so that it issues early, and no attempt is made to control the greed of the algorithm. The mercury

[1] Conte [81] described a processor simulation algorithm called Mercury, which is very similar to the algorithm described here, and from where the name "mercury" was borrowed.

algorithm is presented below.

Algorithm *Mercury*
input
S_{cur}, the *current* schedule, (assumed not more than n_{cur} cycles long);
$G_{cur} = \{R_{cur}, L_{cur}\}$, the machine model description for the *current* machine;
$G_{new} = \{R_{new}, L_{new}\}$, the machine model description for the *new* machine;
$PrioritySet$, a set of priority functions which will be used to compute
the scheduling priority of operations (Since mercury does not
perform explicit operation ordering, this set is empty);
output
S_{new}, the new schedule;
var
n_{cur}, the length of S_{cur};
n_{new}, the length of S_{new};
$Scoreboard$[number of registers], to flag the registers "in–use" in S_{cur}.
$RU\,[n_{new}][\sum_r n_r]$, the resource usage matrix, where:
r represents all the resource types in G_{new}, and,
n_r is the number of instances of each resource type r in G_{new};
$UseInfo\,[n_{new}]$[number of registers], to mark the register usage in S_{new};
T_δ, the cycle in which an Op can be scheduled while
satisfying the data dependence constraints;
$T_{rc+\delta}$, the cycle in which an Op can be scheduled while satisfying
the data dependence and resource constraints;
functions
$RU\_loopkup\,(O\,(T_\delta))$ returns the earliest cycle, later than cycle T_δ, in which Op O
can be scheduled after satisfying the data dependence and resource constraints;
$RU\_update\,(\sigma,\, O)$ marks the resources used by Op O in cycle σ of S_{new};
$dest\_register\,(O)$ returns the destination register of Op O;
$source\_register\,(O)$ returns a list of all source registers of Op O;
$latest\_use\_time\,(\phi)$ returns the latest cycle in S_{new} that register ϕ was used in;
$most\_recent\_writer\,(\rho)$ returns the id of the Op
which modified register ρ latest in S_{cur};
begin
for each MultiOp $M_{cur}[c] \in S_{cur}$, $0 \le c \le n_{cur}$ **do**
begin
— *resource constraint check:*
for each Op $O_w \in S_{cur}$ that completes in cycle c **do**
begin
$O_w\,(T_{rc+\delta}) \leftarrow RU\_loopkup\,(O_w\,(T_\delta))$;
$M_{new}[new\_cycle] \leftarrow M_{new}[new\_cycle] \mid O_w$;

```
            RU_update (T_{rc+δ}, O_w);
        end
    —  update the scoreboard:
    for each OP O_i ∈ S_cur which is unfinished in cycle c do
        begin
            Scoreboard[dest_register (O_r)] ← reserved;
        end
    —  do data dependence checks:
    for each Op O_r ∈ M_cur[c] do
        begin
            O_r (T_δ) ← 0;
            —  anti-dependence:
            for each φ ∈ dest_register (O_r) do
                    O (T_δ) = MAX(O (T_δ), latest_use_time (φ));
            —  pure dependence:
            for each φ ∈ source_register (O_r) do
                if (Scoreboard[φ] == reserved)
                    O_r (T_δ) ← MAX (O_r, (T_δ, completion_time (reserving_Op)));
            —  output dependence:
            for each φ ∈ dest_register (O_r) do
                if (Scoreboard[φ] == reserved)
                    O_r (T_δ) ← MAX (O_r, (T_δ, completion_time (reserving_Op)));
            end
    end
end
```

The worst-case time complexity of the mercury algorithm is $O(n^2)$, where n is the number of operations in the region. In the first step, the algorithm performs picks the "next" operation to be scheduled, which contributes a factor of n to the complexity. The next step decides the earliest cycle in which the operation can be placed so as to honor the data and control dependencies for the operation. This is a constant time step. The third step decides the cycle in which the operation can be placed so as to honor the resource constraints; this is a linear time step and in the worst case, could take n lookups of the resource table.

The mercury algorithm is attractive for the task of rescheduling because it performs a single pass over the region. While the asymptotic worst-case complexity of this algorithm is $O(n^2)$ (which is the same as that of most list scheduling algorithms), in practice, the overhead of rescheduling using mercury is expected to be less than the standard cycle scheduling techniques. This is because mercury does not build an explicit DDG. Since the overhead of rescheduling is the objective function to minimize in this study, the mercury algorithm is a suitable candidate.

The Mercury algorithm has two disadvantages: (1) there is no way to control the greed of operation placement, and (2) the order in which the operations appear in the current schedule is the order in which they will be rescheduled; there is no way to prioritize the operations based on a particular heuristic, as a common list scheduling algorithm would do. Item (2) here is closely related to item (1), because an ability to prioritize the operations results in an ability to control the greed of placement of the low-priority operations over that of the high-priority operations. Hence the focus of the next two algorithms, which are variations of the Mercury algorithm, is to overcome the second limitation.

The Top-Down Mercury algorithm

This variation of the mercury algorithm performs rescheduling in two steps. In the first step, it scans the current schedule in a left-to-right, top-down fashion just as the Mercury algorithm does and constructs an intermediate schedule for an infinite-

resource machine having a unit latency of execution for all operations. Essentially, this intermediate schedule honors only the data and control dependences inherent to the region, without any regard to the resource constraints. In the second step, the intermediate schedule is converted into the new schedule using the Mercury algorithm outlined earlier. Motivation behind this two-step process is that the first step should expose the "heads" of the data dependence chains in the region as early as possible so that they are considered for rescheduling by the Mercury algorithm early on in the process. It is clear that this algorithm makes use of two passes of the Mercury algorithm one after the other, and has $O(n^2)$ time complexity (worst-case).

This algorithm considers all the data dependence chains in the region with equal priority, while constructing the final schedule. This can sometimes be detrimental to the performance of the schedule. To understand this phenomenon, assume a target machine model which is much resource constrained. Using the top-down mercury algorithm, short dependence chains have the same priority as the longer ones. The algorithm could then assign operations in the shorter chains to valuable resources early on in the schedule, thus starving the longer chains. Since the longer chains usually decide the effective height of the schedule, this degrades the performance. The next variation attempts to offset this phenomenon by ordering the dependence chains implicitly.

The Bottom-Up Mercury algorithm

This algorithm also operates in two passes over the region and also has $O(n^2)$ time complexity. In the first pass, the region is scanned one operation at a time, in a right-to-left, bottom-up fashion. Each operation scanned is placed in an intermediate schedule for an infinite resource machine with unit operation latencies. In short, the intermediate schedule is built bottom-up so that the tail operations in each dependence chain have the same priorities, and the priorities of the heads of each chain are now dependent on the height of that chain. It is expected that this variation would result in the best performance of all the three. When the intermediate schedule is used to construct the final schedule, the operations in the longest chain have will the highest priority than the others. This should reduce effective "lengthening" of the final schedule.

6.3.3 Explicit DDG Algorithms

In an explicit DDG-based approach, a DAG of operations in the region encodes the dependence relationships between the operations and the values of fine-grain heuristics for each operation. The DDG could be stored (by the compiler) as a part of the program executable, or could be derived from the set of operations in the region.

Either the cycle scheduling algorithm[2] [82] or the operation scheduling algorithm[3] could then drive the process of scheduling using this DDG. A brief outline of cycle scheduling and operation scheduling is presented in this section. A discussion of the storage requirements for the DDG is presented in Chapter 7.

The cycle scheduling algorithm

Algorithm *Cycle_Schedule*

input

S_{cur}, the *current* schedule;
$G_{cur} = \{R_{cur}, L_{cur}\}$, the machine model description for the *current* machine;
$G_{new} = \{R_{new}, L_{new}\}$, the machine model description for the *new* machine;
$PrioritySet$, a set of priority functions which will be used to compute the scheduling priority of operations;

output

S_{new}, the new schedule;

var

L, the list of operations used for scheduling;
cycle, temporary variable used to keep track of the current cycle in the new schedule;

functions

$Build\_DDG\,(S_{cur}, G_{cur})$ returns the data dependence graph for the operations in S_{cur}, generated using the current machine model G_{cur};
$Build\_List\,(PrioritySet, DDG)$ returns an ordered list of operations in DDG, the ordering criterion is indicated using $PrioritySet$;
$Recompute\_Ready\_Ops\,(L, PrioritySet, DDG)$ recomputes the ready Ops in the list L using the $PrioritySet$ and the data dependence graph DDG;
$Schedule\_Op\,(O, cycle)$ schedule the operation O in cycle *cycle*;
$Delete\_Op\,(O, L)$ deletes the operation O from list L;
$Resource\_Check\,(O, cycle, G_{new})$ returns OK if there is no resource conflict

---

[2]Cycle scheduling, as described here, is the same as *list scheduling* as referred to by many researchers. Need for this variation in the nomenclature arises because both the cycle and operation scheduling algorithms are list scheduling algorithm, which calls for a finer distinction.

[3]There are many references to the operation scheduling algorithm; see[80] and [24], for example. Description of the algorithm given by Ellis in [80] is more detailed than the others. In [24], Rau mentions operation scheduling in the context of scheduling a loop body. The mercury algorithm discussed in Section 6.3.2 is a version of operation scheduling which operates on DDG implicitly.

```
    for machine G_new in cycle cycle for operation O, else returns NOTOK;
  Next_Ready_Op(L, cycle) returns the next ready op from list L,
    not yet visited in cycle cycle;
begin
  DDG ← Build_DDG(S_cur, G_cur);
  L ← Build_List(PrioritySet, DDG);
  cycle ← 0;
  while (!List_Empty(L))
  begin
    no_op_ready ← FALSE;
    while (!no_op_ready)
    begin
      O ← Next_Ready_Op(L, cycle);
      if (O = NULL)
        begin
          no_op_ready ← TRUE;
           break;
        end;
      if (Visited(O))
         continue;
      if (Resource_Check(O, cycle, G_new) = OK)
        begin
          Schedule_op(O, cycle);
          Delete_Op(O, L);
          Recompute_Ready_Ops(L, PrioritySet, DDG);
        end
      else
        Mark_Visited(O);
    end
    cycle ← cycle + 1;
  end
end
```

The operation scheduling algorithm

Algorithm *Operation_Schedule*

input

S_{cur}, the *current* schedule;

$G_{cur} = \{R_{cur}, L_{cur}\}$, the machine model description for the *current* machine;

$G_{new} = \{R_{new}, L_{new}\}$, the machine model description for the *new* machine;
$PrioritySet$, a set of priority functions which will be used to compute
the scheduling priority of operations;

output

S_{new}, the new schedule;

var

L, the list of operations used for scheduling;
cycle, temporary variable used to keep track of machine cycle in the new schedule;

functions

$Build\_DDG\,(S_{cur}, G_{cur})$ returns the data dependence graph for the operations in S_{cur},
generated using the current machine model G_{cur};
$Build\_List\,(PrioritySet, DDG)$ returns an ordered list of operations in DDG,
the ordering criterion is indicated using $PrioritySet$;
$Recompute\_Ready\_Ops\,(L, PrioritySet, DDG)$ recomputes the ready Ops in the
list L using the $PrioritySet$ and the data dependence graph DDG;
$Schedule\_Op\,(O, cycle)$ schedule the operation O in cycle *cycle*;
$Delete\_Op\,(O, L)$ deletes the operation O from list L;
$Resource\_Check\,(O, cycle, G_{new})$ returns OK if there is no resource conflict
for machine G_{new} in cycle *cycle* for operation O, else returns $NOTOK$;
$Next\_Ready\_Op\,(L)$ returns the next *ready* op from list L;

begin

$DDG \leftarrow Build\_DDG\,(S_{cur}, G_{cur})$;
$L \leftarrow Build\_List\,(PrioritySet, DDG)$;
while $(!List\_Empty\,(L))$
begin
$cycle \leftarrow 0$;
while $(TRUE)$
begin
$O \leftarrow Next\_Ready\_Op\,(L)$;
if $(Resource\_Check\,(O, cycle, G_{new}) = OK)$
begin
$Schedule\_op\,(O, cycle)$;
$Delete\_Op\,(O, L)$;
$Recompute\_Ready\_Ops\,(L, PrioritySet, DDG)$;
break;
end
else
$cycle \leftarrow cycle + 1$;
end
end
end

Chapter 7

Object-code Annotations

Various pieces of information about the object-code must be available to the rescheduler. Some of this information may be derived from the object-code; other pieces must be stored in the object-file at the initial compile time. In the following section, an exhaustive list of all the information and the purpose for which it can be used is presented. In Section 7.2, an example is used to elaborate the techniques for storage of the data dependence graph (DDG). Section 7.3 presents results of measurements of space requirements used to store the object-file annotations for SPECInt95 code.

7.1 Set of Object-code annotations

1. **The dependence graph**:

   The dependence graph stores data (*flow*, *anti* and *output*) and control dependence between various operations in the region. The DDG may be derived at

run-time, or may be stored in the object-file. The cost of storing it an an annotation is the disk-space required to store it and the time to retrieve it from the annotation tables at runtime. The DDG is probably the most vital piece of information about object-code, since it drives the rescheduling algorithm and ensures correctness. In the absence of the DDG, the rescheduling algorithms must take a conservative approach to scheduling operations, which could lead to longer schedules.

2. **Register liveness information**:

   The liveness information is stored in the form of a list of register id's that are live at each region exit and region fall-throughs. This information allows the rescheduler to make speculation decisions correctly (both upward and downward code percolation). The liveness at side-exits internal to a region of code is also useful because it helps make speculation decisions from multiple paths of code within the region. Liveness information may not be necessary in all rescheduling scenarios. Without this information, rescheduling can be performed conservatively *i.e.,* without any code speculation, possibly yielding poor performance.

3. **Speculation information**:

   A list of speculated operations along with their parent basic blocks (basic blocks where the op was positioned before it was scheduled) would help perform de-speculation. A de-speculation pass before rescheduling the code ensures that the speculation decisions made by the initial scheduling pass do not *bias* the

speculation decisions that the rescheduler would make. Speculation information must be stored by the compiler during the initial scheduling pass; it cannot be accurately derived at run-time. Alternatively, if the information is not available the rescheduler may attempt to demote all the operations, immaterial of whether they were speculated initially or not. This would be the equivalent of performing aggressive de-speculation.

4. **Side-exit probabilities**:

   This info allows the rescheduler to make the *region rescheduling* decisions, and *region re-formation* decisions (if applicable). Side-exit probabilities can be monitored dynamically, using special-purpose profiling hardware [1]. The original side-exit probability values used at the time of initial compile (for doing region formation or during code scheduling) should also be made available in the object-file.

5. **Function call frequencies**:

   The function call frequencies at each call-site that were used in making the original function-inlining decisions should be stored in the object-file by the compiler. This information can be used by the rescheduler, along with the dynamic function call frequencies that it monitors, to make decisions about more aggressive inlining (if the call frequencies have increased) or de-inlining (if the call frequencies have decreased sizably) of function call sites.

Also, a measurement of the runtime spent in a function is desirable, because it helps make narrow the domain of code over which region rescheduling decisions are made.

6. **Machine model**:

   A relevant subset of the machine model, or a machine model id, or a reduced machine model description can be stored along with the object-file, and be used while rescheduling.

7. **Region boundaries**:

   Region boundary markers, along with region-type identifier for each region be made available in the object-code.

8. **Register spill information**:

   This information is useful to the rescheduler when it performs a register re-allocation pass. Register spill info may be derived dynamically, and does not have to be stored in the object-file. Or, as an alternative to the spill-info, the compiler may store the *virtual register numbers (VRNs)* that were used in the intermediate representation.

9. **Memory disambiguation information**:

   This allows the rescheduler to make correct memory operation re-ordering decisions. It may be extremely hard to derive this information dynamically via analysis in software.

Hardware-based profiling techniques could be used to monitor the memory operations for interference. The data generated using the hardware-based techniques cannot be precise, but would only provide the approximation of the level of interference (e.g., operations X and Y interfere $p\%$ of the total number of times they are executed together.) Hence, a technique like Chen's Memory Conflict Buffer (MCB) [83], or *speculative disambiguation* [84] proposed by Huang, Slavenberg, and Shen, must be employed to handle and correct the inappropriate re-ordering decisions.

10. **Meld information**:

    This is useful if the rescheduler operates on more than one regions at a time, and take advantage of a possible overlap at the end of region schedules. Dynamic techniques based on the concept of *meld scheduling* [85] can be devised and applied if meld information is available.

7.2 Space requirements for storing DDGs

The storage requirements for DDGs corresponding to each region in code can be overwhelming if a good representation is not chosen. This section explores the possibilities for storing the DDG using an following example. Assume that a region suitable for rescheduling may have multiple component DDGs, one each for its component subregions. In what follows, a list of items of the DDG is presented, along with an estimate

of space requirements to store each field is shown.

- Number of DDG Components (1 byte)
  (max. 2566 components)
- For each DDG component:
  - Number of nodes in the component (1 byte)
    (max 256 nodes)
  - For each node:
    - Id of the operation that the node indicates (1 byte)
      (The Id is the serial number of the operation in the region.
      This assumes that there are not more than 256 operations in a region).
    - Number of dependences to/from the node (1/2 byte)
      (max 16 dependences)
    - For each dependence:
      - Id of the other node (1 byte)
      - Type of the dependence (1/2 byte) (flow, out, anti, branch, dependences; dependence direction is also indicated).

This representation of the DDG is straightforward, and can be used "as is" by the rescheduler, i.e. no extra computational effort is required in order to extract the DDG before rescheduling. But the amount of space required to store this DDG in an executable could be overwhelming. For example, for a region with 256 nodes, 128 dependences, all part of a single DDG, the space requirement to store this DDG is: $1 + 1 + (1 + 1/2) * 256 + (1 + 1/2) * 128 = 578$ bytes.

Assuming that each operation is encoded in 8 bytes (as in the TINKER encoding), the amount of space occupied by the region itself is: $256 * 8 = 2048$ bytes. In this case, the percentage increase in the executable size is approx. 28%. Obviously, this is is a large increase in the size of the executable, and may not be generally be acceptable.

There are two ways to deal with this problem. First, the DDG may be stored for a few selected regions that the compiler decides would potentially benefit the most from it. The DDG will then be used at run-time for rescheduling by the cycle- or operation-scheduling algorithm. The trade-off performance gain by the use of DDG is the space needed to store the DDG.

Alternatively, a better encoding for the DDG fields could be provided. Notice in the above DDG description, each dependence was stored *twice*: once as a part of the operation of its origin, and once as a part of the operation of its destination. This is termed as the *operation–centric* representation of the DDG. Alternatively, the DDG may be stored as a list of dependences, or in a *dependence–centric* manner, which may lead to substantial saving in space. This DDG representation shown below:

- Number of DDG Components (1 byte)
  (max 256 components)
- The dependence dictionary for each DDG component:
  - Number of dependences in the dictionary (2 bytes)
    (max 64K dependences)
  - For each dependence:
    - Id of the source node (1 byte)
    - Id of the destination node (1 byte)
    - Type of the dependence (1/2 byte)

For the above example, this would require $(1+2+(1+1+1/2)*128) = 323$ bytes, is an increase of approx. 15%, which is little more than than half of the space requirements of operation–centric representation of the DDG.

7.3 Measurement of space requirements

This section presents preliminary results on the space requirements of the object-code annotation segment for benchmarks *130.li* and *134.perl* in the SPECInt95 benchmark suite. Results are shown in Tables 7.1 and 7.2. The following pieces of information were saved for the entire program: *register liveness information, speculation information, side-exit probabilities, function-call frequencies, function-execution frequencies, machine model id,* and *region boundary info*. The data dependence graph was also saved, but not for the entire program. A select list of regions from the benchmark, as prescribed by the EC Oracle, was used to pick and select the DDGs. This saved the amount of space required substantially. The format used to store DDGs was node-centric. The information was collected from code scheduled for two distinct machine models: SU-BR1-T4, and SU-BR1-T8. The code consisted of superblock and basicblock code.

Table 7.1: *130.li*: Object-code annotation space requirements for code scheduled for two machine models.

Description	SU-BR1-T4	SU-BR1-T8
Total number of procedures	357	357
Total number of regions	2246	2246
Total number of operations	33444	36002
Total number of DDGs saved	41	41
Size of annotation segment (uncompressed bytes)	374676	378668
Size of annotation segment (compressed bytes)	85526	85896
Increase in the size of the binary	24.2%	23.0%

Table 7.2: *134.perl*: Object-code annotation space requirements for code scheduled for machine model SU-BR1.

Description	SU-BR1-T4
Total number of procedures	275
Total number of regions	9973
Total number of operations	136828
Total number of DDGs saved	51
Size of annotation segment (uncompressed bytes)	1235752
Size of annotation segment (compressed bytes)	268248
Increase in the size of the binary	24.5%

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Today's high-performance computer architectures heavily depend on the compiler to statically extract the instruction-level parallelism in common programs. Compilers use the knowledge of program behavior, known as the program *profile*, and the knowledge of the machine microarchitecture for this purpose. The basic idea is to concentrate the effort of optimization on most heavily executed paths in code, while optimizing the hardware resource usage. Profiles are collected at development time using a set of common, most-representative set of program inputs. Several times, however, the input set used to collect the profile is not a fair representation of inputs that the program is subject to. This often to leads to invalidation of assumptions made by the compiler about the behavior of the program. This is a deep flaw of the

profile-based optimization technique. Dependence on the knowledge of machine microarchitecture (the machine "model") introduces another problem. Code scheduled for one microarchitecture may see substantial performance loss when executed on a different microarchitecture. Even worse, this dependence could lead to the problem of inter-generation *incompatibility* in the completely statically scheduled VLIW architectures. Implications of binary incompatibility on the success of an architecture in the marketplace are well-know, and hence the problem must be solved using simple, effective techniques dynamically.

This thesis has described *Evolutionary Compilation* as a solution to these problems. In the evolutionary compilation framework, several components are added to the OS. Behavior of programs as they executed is monitored using special-purpose, programmable hardware. The hardware is programmed by the OS to watch for specific events during execution of the program. At suitable instances (such as page-fault handling), the OS invokes a module called the *EC Oracle* to decide if the program profile has shifted over time, thus leading to performance degradation. If the outcome of the oracle is positive, code evolution is deemed warranted. The oracle generates a list of regions in the code, rescheduling which would be beneficial. The *evolutionary compiler*, another piece of the EC framework, then accesses a database of information about the program object-code called *annotations* saved during initial compile, to perform the rescheduling optimizations. Support from the machine ISA encoding is beneficial to support features of EC, such as size-invariance of code when subjected

to code transformations.

When code that is incompatible with the current generation of the machine is presented for execution, the OS *loader* detects this event at first-time page faults. At the time, the loader invokes a dynamic rescheduler module in the EC framework, which also accesses the object-code annotations, and generates code such that the correctness of execution on the current generation of the machine is guaranteed. All rescheduling transformations are performed in software, and are initiated by the OS. The cost of these transformations is the time-overhead of rescheduling. Caching techniques to amortize the cost, such as the *Persistent Rescheduled-Page Cache (PRC)* were presented and studied in detail in this thesis. The PRC saves translated pages of code, and organizes them in an OS-wide cache. Different placement-replacement policies were also studied for PRC. Detailed description of the set of object-code annotation that is useful in rescheduling transformations was presented. Space requirements for a subset of the information were studied.

8.2 Future Work

The weakness of Superblocks to withstand profile shifts in code was shown in this thesis. The superblocks are linear regions of code, which contain only a *fallthru* path of a branch. In the event of profile shifts, it was suggested that code organized as superblocks may be subjected to *region re-formation* in order to adapt to the shifts in program behavior. The issues involved in achieving this, algorithms to perform

the region re-formation transformations, are an area of open research.

A large amount of profile shift occurs at function-call level. Users, at an early stage of using a program use different functionalities in code than in the later stage. Due to this shift in usage pattern a change in function call/execution frequencies may be evident in some highly interactive programs. As a result of this, function call *inlining* decisions made early-on during the initial compile may become invalid as time progresses. The program code may be *de-inlined* and *re-inlined* dynamically, to reflect this change in behavior. The algorithms, support from object-code annotations to achieve this, and the heuristics involved in selecting procedures to be de-inlined and re-inilined is an area of future work.

Third, loop code, which has been scheduled using any of the software pipelining techniques deserves further inquiry. Some initial ideas on how to un-modulo schedule a loop (for a type of modulo-scheduled [24] loops called the *kernel-only (KO)* loops) was presented in [86]. Further investigation of these ideas is needed. Also, in the same vein, study of loop structures which are *unrolled* is needed. Depending on the changes in the program behavior, loop *re-rolling* may have to be performed in some cases, or the degree of unrolling may have to be intensified in other.

Bibliography

[1] T. M. Conte, B. A. Patel, and J. S. Cox, "Using branch handling hardware to support profile-driven optimization," in *Proc. 27th Ann. Int'l Symp. on Microarchitecture*, (San Jose, CA), Nov. 1994.

[2] K. M. Menezes, *Hardware-based profiling.* PhD thesis, Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911, Oct. 1997.

[3] J. S. Cox, D. P. Howell, and T. M. Conte, "Commercializing profile-driven optimization," in *Proc. 28th Hawaii Int'l. Conf. on System Sciences*, vol. 1, (Maui, HI), pp. 221–228, Jan. 1995.

[4] W. W. Hwu and P. P. Chang, "Trace selection for compiling large C application programs to microcode," in *Proc. 21st Ann. Workshop on Microprogramming and Microarchitectures*, (San Diego, CA.), Nov. 1988.

[5] W. W. Hwu and P. P. Chang, "Exploiting parallel microprocessor microarchitectures with a compiler code generator," in *Proc. 15th Ann. Int'l Symp. Computer Architecture*, (Honolulu, Hawaii), pp. 45–53, May 1988.

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

[7] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proc. 16th Ann. Int'l Symp. Computer Architecture*, (Jerusalem, Israel), pp. 242–251, May 1989.

[8] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, no. 7, pp. 478–490, July 1981.

[9] D. Bernstein and M. Rodeh, "Global instruction scheduling for superscalar machines," in *Proceedings of the ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, (Toronto, Canada), pp. 241–255, June 1991.

[10] S.-M. Moon and K. Ebcioğlu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," in *Proc. 25th Ann. Int'l Symp. Microarchitecture* [87], pp. 55–71.

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

[12] J. A. Fisher, "Global code generation for instruction-level parallelism: Trace Scheduling-2," Tech. Rep. HPL-93-43, Hewlett-Packard Laboratories, June 1993.

[13] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. Rau, and M. Schlansker, "Profile-driven instruction level parallel scheduling with application to superblocks," in *Proc. 29th Ann. Int'l Symp. Microarchitecture* [88], pp. 58—67.

[14] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proc. of the ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pp. 16–27, June 1990.

[15] R. A. Bringmann, *Enhancing instruction level parallelism through compiler-controlled speculation*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, 1995.

[16] B. L. Deitrich and W. W. Hwu, "Speculative hedge: regulating compile-time speculation against profile variations," in *Proc. 29th Ann. Int'l Symp. Microarchitecture* [88].

[17] R. P. Colwell, R. P. Nix, J. J. O'Donnel, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems.*, (Palo Alto, CA), pp. 180–192, Oct. 1987.

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

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

[20] J. C. Denhert and R. A. Towle, "Compiling for the Cydra 5," *J. Supercomputing*, vol. 7, pp. 181–227, Jan. 1993.

[21] D. A. Dunn and W. Hsu, "Instruction scheduling for the HP PA-8000," in *Proc. 29th Ann. Int'l Symp. Microarchitecture* [88], pp. 298–307.

[22] A. Holler, "Optimization for a superscalar out-of-order machine," in *Proc. 29th Ann. Int'l Symp. Microarchitecture* [88], pp. 336–348.

[23] J. S. O'Donnell, "Superscalar vs. VLIW," *Computer Architecture News (ACM SIGARCH)*, pp. 26–28, Mar. 1995.

[24] B. R. Rau, "Dynamically scheduled VLIW processors," in *Proc. 26th Ann. Int'l Symp. Microarchitecture*, (Austin, TX), pp. 80–90, Dec. 1993.

[25] S. Melvin, M. Shebanow, and Y. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *Proc. 21th Ann. Int'l Symp. on Microarchitecture*, (San Diego, CA), pp. 60–66, Dec. 1988.

[26] M. Franklin and M. Smotherman, "A fill-unit approach to multiple instruction issue," in *Proc. 27th Ann. Int'l Symp. Microarchitecture* [89], pp. 162–171.

[27] G. Silberman and K. Ebcioğlu, "An architectural framework for supporting heteregeneous instruction–set architectures," *Computer*, vol. 26, pp. 39–56, June 1993.

[28] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Comm. ACM*, vol. 36, pp. 69–81, Feb. 1993.

[29] J. Turley, "Alpha runs x86 code with fx!32," *Microprocessor Report*, vol. 10, Mar. 1996.

[30] P. Stears, "Emulating the x86 and DOS/Windows in RISC environments," in *Proc. Microprocessor Forum*, Oct. 1994.

[31] R. Cmelik and D. Keppel, "SHADE: A fast instruction-set simulator for execution profiling," in *Fast Simulation of Computer Architectures* (T. M. Conte and C. E. Gimarc, eds.), Boston, MA: Kluwer Academic Publishers, 1994.

[32] E. Altman and K. Ebcioğlu, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," in *Proc. 24nd Ann. Int'l Symp. Computer Architecture*, (Denver, CO), June 1997.

[33] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the Hyperblock," in *Proc. 25th Ann. Int'l Symp. Microarchitecture* [87], pp. 45–54.

[34] W. A. Havanki, "Treegion scheduling for VLIW processors," Master's thesis, Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911, July 1997.

[35] S. Banerjia, W. A. Havanki, and T. M. Conte, "Treegion scheduling for highly parallel processors," in *Proc. Euro-Par'97*, (Passau, Germany), Aug. 1997.

[36] M. S. Schlansker and V. K. Kathail, "Techniques for critical path reduction of scalar programs," Tech. Rep. HPL-95-112, Hewlett-Packard Laboratories, Palo Alto, CA, 1995.

[37] "TINKER machine language manual," 1995. Department of Electrical and Computer Engineering, North Carolina State University, Raleigh NC 27695-7911.

[38] V. Kathail, M. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, Feb. 1994.

[39] K. M. Dixit, "CINT92 and CFP92 benchmark descriptions," *SPEC Newsletter*, vol. 3, no. 4, 1991. SPEC, Fairfax, VA.

[40] T. M. Conte and S. W. Sathaye, "Dynamic rescheduling: A technique for object code compatibility in VLIW architectures," in *Proc. 28th Ann. Int'l Symp. Microarchitecture*, (Ann Arbor, MI), Dec. 1995.

[41] Hewlett Packard, *How HP–UX works: Concepts for the System Administrator (R9.0)*. Palo Alto, CA: Hewlett Packard, 1991.

[42] Data General, *Programming in the DG/UX Kernel Environment (R4.11)*. Westboro, MA: Data General, 1995.

[43] G. Kane, *MIPS RISC architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

[44] S. Weiss and J. E. Smith, *POWER and PowerPC*. San Francisco, CA: Morgan Kaufmann, 1994.

[45] Data General, *Analyzing DG/UX System Performance (R4.11)*. Westboro, MA: Data General, 1995.

[46] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison-Wesley, 1989.

[47] M. J. Bach, *The design of the UNIX operating sustem*. Englewood Cliffs, NJ: Prentice–Hall, 1986.

[48] M. Nelson, B. Welch, and J. K. Ousterhout, "Caching in the Sprite network file system," *ACM Trans. Comput. Sys.*, vol. 6, pp. 134–154, Feb. 1988.

[49] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: a distributed personal computing environment," *Comm. ACM*, vol. 29, no. 3, pp. 184–201, 1986.

[50] C. May, "MIMIC: A Fast System/370 Simulator," in *Proc. ACM SIGPLAN '87 Symp. Interpreters and Interpretive Techniques*, pp. 1–13, June 1987.

[51] A. E. Eichenberger and E. Davidson, "A reduced multipipeline machine description that preserves scheduling constraints," in *Proc. ACM SIGPLAN 1996 Conf. Programming Language Design and Implementation*, (Philadelphia, PA), May 1996.

[52] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau, "Optimization of machine descriptions for efficient use," in *Proc. 29th Ann. Int'l Symp. Microarchitecture* [88].

[53] V. Bala and N. Rubin, "Efficient instruction scheduling using finite state automata," in *Proc. 28th Ann. Int'l Symp. on Microarchitecture*, (Ann Arbor, MI), Nov. 1995.

[54] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proc. 14th Ann. Workshop on Microprogramming*, pp. 183–198, Nov. 1981.

[55] B. Rau, C. Glaeser, and R. Picard, "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," in *Proc. 9th Ann. Int'l Symp. Computer Architecture*, (Austin, TX), pp. 131–139, Apr. 1982.

[56] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proc. of the ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pp. 318–327, June 1988.

[57] J. C. Denhert, P.-T. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proc. Third Int'l. Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, (Boston, MA), pp. 26–38, Apr. 1989.

[58] A. Nicolau and R. Potasman, "Realistic scheduling: compaction for pipelined architectures," in *Proc. 23rd Ann. Int'l Symp. on Microarchitecture*, (Orlando, FL), pp. 69–79, 1990.

[59] B. Su and J. Wang, "GURPR*: A new global software pipelining algorithm," in *Proc. 24th Ann. Int'l. Symp. on Microarchitecture*, (Albuquerque, NM), pp. 212–216, Nov. 1991.

[60] B. R. Rau, M. Schlansker, and P. P. Tirumalai, "Code generation schemas for modulo scheduled DO-loop and WHILE-loops," Tech. Rep. HPL-92-47, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, 1992.

[61] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Ann. Int'l Symp. Microarchitecture* [89].

[62] N. Warter, *Modulo Scheduling with Isomorphic Control Transformations*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 1994.

[63] M. G. Stoodley and C. G. Lee, "Software pipelining of loops with conditional branches," in *Proc. 29th Ann. Int'l Symp. Microarchitecture* [88], pp. 262–273.

[64] J. C. H. Park and M. Schlansker, "On predicated execution," Tech. Rep. HPL-91-58, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, 1991.

[65] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling: A model for compiler-controlled speculative execution," *ACM Trans. Comput. Sys.*, vol. 11, pp. 376–408, Nov. 1993.

[66] S. A. Mahlke, *Exploiting instruction level parallelism in the presence of branches*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, 1996.

[67] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 266–275, May 1991.

[68] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox, "Hardware-based profiling: An effective technique for profile-driven optimization," *Int'l Journal of Parallel Programming*, vol. 24, Feb. 1996.

[69] C. Loeffler, A. Lightenberg, and G. Moschytz, "Practical fast 1-D DCT algorithms with 11 multiplications," in *Proc. 1989 Int'l Conf. on Acoustics, Speech, and Signal Processing*, pp. 988–991, 1989.

[70] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[71] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inform. Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[72] T. M. Conte *et al.*, *The TINKER Machine Language Manual*. North Carolina State University, Raleigh, NC 27695-7911, Apr. 1995.

[73] W. A. Havanki, S. Banerjia, and T. M. Conte, "Treegion scheduling for wide-issue processors." To appear in *Proc. 4th Int'l Symp. High Performance Computer Architecture (HPCA-4)*.

[74] D. Wall, "Predicting program behavior using real or estimated profiles," in *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, (Toronto, Ontario, Canada), pp. 59–70, June 1991.

[75] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proc. 5th Int'l. Conf. on Architectural Support for Prog. Lang. and Operating Systems*, (Boston, MA), pp. 85–95, Oct. 1992.

[76] T. M. Conte and W. W. Hwu, "The validity of optimizations based on profile information," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1992.

[77] D. Freedman, R. Pisani, and R. Purves, *Statistics*. New York: W. W. Norton and Company, 1978.

[78] L. Gonick and W. Smith, *The Cartoon Guide to Statistics*. New York: Harper-Perennial, 1993.

[79] D. Melamed, "NLP rsearch software library." http://www.cis.upenn.edu/ melamed/genproc.html, 1998.

[80] J. R. Ellis, *Bulldog: A compiler for VLIW architectures*. Cambridge, MA: The MIT Press, 1986.

[81] T. M. Conte, *Systematic computer architecture prototyping*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 1992.

[82] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*. New York, NY: John Wiley and Sons, 1976.

[83] W. Y. Chen, *Data preload for superscalar and VLIW processors*. PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, IL, 1993.

[84] A. S. Huang, G. Slavenberg, and J. P. Shen, "Speculative Disambiguation: a compilation technique for dynamic memory disambiguation," in *Proc. 21st Ann. Int'l Symp. Computer Architecture*, (Chicago, IL), pp. 200–222, May 1994.

[85] S. G. Abraham, V. Kathail, and B. L. Deitrich, "Meld schuedling: relaxing scheduling constraints across region boundaries," in *Proc. 29th Ann. Int'l Symp. Microarchitecture* [88], pp. 308–321.

[86] T. M. Conte and S. W. Sathaye, "Properties of rescheduling size invariance for dynamic rescheduling-based VLIW cross-generation compatibility," Technical report, Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911, Apr. 1997. Submitted for publication.

[87] *Proc. 25th Ann. Int'l Symp. Microarchitecture*, (Portland, OR), Dec. 1992.

[88] *Proc. 29th Ann. Int'l Symp. Microarchitecture*, (Paris, France), Dec. 1996.

[89] *Proc. 27th Ann. Int'l Symp. Microarchitecture*, (San Jose, CA), Dec. 1994.