DYNAMIC INSTRUCTION REUSE

by

AVINASH SODANI

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

University of Wisconsin—Madison

2000

Abstract

Traditionally, improvements in processor microarchitecture have come from observing program characteristics and devising mechanisms to exploit them. This thesis presents a new phenomenon exhibited by programs and proposes a novel microarchitectural technique for exploiting it to improve processor performance. The phenomenon, called *dynamic instruction repetition*, is that instructions in programs often execute repeatedly with the same input values and produce the same results over and over again. The new microarchitectural technique proposed in this thesis exploits this phenomenon to reduce the work that needs to be done in executing programs. This technique, called *dynamic instruction reuse*, detects that instructions are producing the same results repeatedly, and instead of re-executing them, reuses the results from the instructions' previous executions. This technique improves performance because of several reasons, one of which is its ability to collapse data dependences by completing dependent instructions simultaneously.

This thesis makes two main contributions:

1. It studies the phenomenon of instruction repetition, presenting numerous characterization results and performing detailed analyses to better understand the causes of this phenomenon.

2. It introduces and studies the concept of dynamic instruction reuse. It presents four instruction reuse schemes. These schemes reuse results of instructions from a hardware table called the *Reuse Buffer (RB)*, where the results are stored previously. The validity of these

old results is established by checking whether the current operand values are the same as those used to calculate the old results. The thesis also studies the size and associativity requirements for the storage needed for saving instruction results, and presents four new policies for managing this storage efficiently. Finally, this thesis studies the interactions of instruction reuse with other key microarchitectural features in processors.

The experimental results show that there is abundant instruction repetition in programs, and that significant percentage of this repetition can be reused. Although the resultant performance improvements are not commensurately high, they are still significant in many cases.

Acknowledgements

I would like to thank my advisor, Guri Sohi, for his guidance and support during my graduate studies, and for the invaluable training I received as his student. His influence is sure to stay with me for a long time and help me in my career in many ways. As his student, I also cherished the freedom he offered to explore and develop new ideas. This freedom kept my interest alive in my research all through out the Ph.D.

I would like to thank Mark Hill, Jim Goodman, Charles Fischer, Jim Smith, and David Wood for serving on my preliminary and defense committees. Their incisive evaluation of my work helped me focus my research in the correct direction. Mark Hill and Jim Goodman also severed as readers for this thesis. This thesis benefitted immensely from their efforts.

Many people helped me during my years in graduate school. Andy Glew, Andreas Moshovos, Shubu Mukherjee, and T.N. Vijaykumar were always willing to offer advice. I also learnt a lot about making good talk slides from Andreas. It was a pleasure sharing an office with Andy. The numerous discussions I had with him on variety of topics helped me develop a better appreciation for fields beyond computers.

This thesis work would not have been possible without the love, support, and constant encouragement from my family. The immense pride that my parents and brothers take in my smallest of achievements was a constant source of motivation. My wife Shilpa cheerfully put up with my spending countless hours at work. It was her love and support that made the final and, arguably, the most strenuous year of graduate school much easier to withstand.

Contents

Chapter 1

Introduction

Over the past decade, microprocessors have become immensely powerful. This growth in performance has been made possible not only by improvements in semiconductor technology (resulting in higher clock frequencies) but also by advancements in the processor microarchitecture (resulting in more work performed per clock cycle). With computers becoming ubiquitous and the way increasingly complex tasks being entrusted upon them, the need for faster processors is likely to grow unabated in the near future. To satisfy this requirement, it is important not only to improve the semiconductor technology, but also to innovate in the field of microarchitecture.

Microarchitectural innovations are often inspired by commonly observed behavior of programs. Designers have frequently introduced new microarchitectural features for exploiting patterns in program behavior to improve processor performance. Some examples of commonplace microarchitectural features in modern processors that exploit program behavior are caches, branch prediction, and the out-of-order execution paradigm. These features exploit different traits in programs and, hence, improve processor performance in different ways. Caches exploit locality of memory references, a property exhibited by most programs, to reduce memory access time. Branch prediction exploits regularity in branching behavior, another property exhibited by programs, to streamline instruction-fetch. The out-of-order execution paradigm exploits the presence of significant amounts of instructions-level parallelism in programs to hide the latency of long-running operations. Thus, as shown by these example, the knowledge of program characteristics is central to improving processor microarchitecture. To further improve processor microarchitecture, we need to seek out new program characteristics and devise mechanisms to exploit them.

In this thesis, we present a new phenomenon exhibited by programs and propose a novel microarchitectural technique for exploiting this phenomenon to improve processor performance. The phenomenon is that in programs, instructions often execute repeatedly with the same inputs and, therefore, produce the same results over and over again. That is, if an instruction executes with operand values *v1* and *v2* and produces an output *v3*, then during program execution this instruction may execute with *v1* and *v2* as inputs and produce *v3* as output many times. We call this phenomenon *dynamic instruction repetition*, or simply, *instruction repetition*.

The microarchitectural technique that we propose exploits this phenomenon to reduce the amount of work that needs to be done for executing a program. This technique detects that instructions are producing the same results repeatedly, and instead of re-executing them, reuses the results from their previous executions. The repetition is detected by ascertaining that the current operand values of the instructions are the same as those used to compute the previous results. We call this technique *dynamic instruction reuse*, or simply, *instruction reuse.*

What are the benefits of reusing instructions? There are several. First, a reused instruction need not be executed. Hence, the pipeline resources (e.g., issue window entry, functional units, data cache ports) that would have been used for its execution can now be used for processing other waiting instructions. Second, when an instruction is reused, its results become known earlier than they would have through regular execution. This permits other instructions that are dependent on these results to execute sooner. Third, as we shall illustrate shortly, this mechanism allows useful work to be salvaged from the work that is discarded due to mis-speculation in processors. This helps alleviate the penalty of such mis-speculations. Fourth, reuse collapses data dependences: dependent instructions, which would normally execute sequentially, can be reused in parallel. Hence, reuse has the potential to break the dataflow limits on the execution times of instructions.

In this thesis, we study the phenomenon of instruction repetition and develop the instruction reuse technique for exploiting it. We present an overview of the contributions of this thesis in Section 1.2. However, before that, to develop a better feel for instruction repetition, we illustrate why this phenomenon occurs, in the next section. In Section 1.3, we describe the related work, and finally, in Section 1.4, we conclude this chapter by presenting the outline of the rest of the thesis.

1.1 Scenarios for Instruction Repetition and Reuse

Instructions get repeated because of two main reasons: (i) speculative execution, and (ii) the nature of the program itself. We present two scenarios to illustrate these reasons below. In each example, we also mention how exploiting that form of repetition may improve performance.

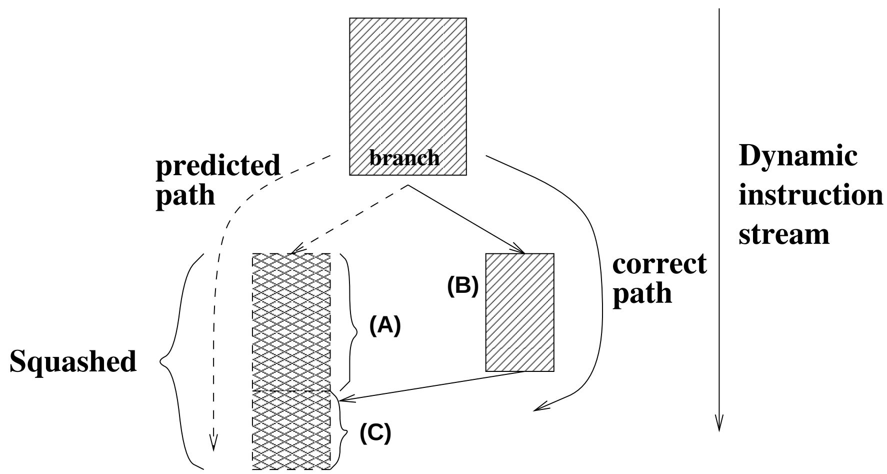


Figure 1.1 Scenario where execution on the (mis)predicted path converges with the execution on the correct path. In such cases certain instructions from part (C) need not be re-executed when encountered on the correct path.

1.1.1 Scenario 1: Squash Repetition and Reuse

Squash Repetition: In the first scenario, the instructions are repeated because of the speculative execution of programs. When executing instructions speculatively, processors discard executed instructions on mis-speculations. These discarded instructions are sometimes executed again, resulting in repetition. For example, consider the scenario shown in Figure 1.1. When a branch instruction is encountered, its outcome is predicted, and instructions from the predicted basic block (block A) are executed speculatively. In addition to executing instructions from block A, the processor may execute instructions from another block (C), which is control independent of the branch. If the branch were mispredicted, instructions executed from both blocks A and C would be discarded, and execution would resume at block B, from where it would proceed to block C. Instructions in block C that were discarded, but whose operands are not affected by instructions in either blocks A or B, would end up being *repeated.* Since this repetition is engendered by squashes, we term it as *squash repetition*.

Squash Reuse: In the above example, if results of the instructions in block C were buffered, then they could be reused after detecting that their operands are the same as at the time of the first execution. Since this reuse is enabled by squash repetition, we call it *squash reuse*.

Benefits: This form of reuse alleviates the mis-speculation penalty, but to understand why, we need to see what constitutes the mis-speculation penalty. The mis-speculation penalty consists of two components: (i) the cycles that are wasted executing instruction on the wrong path, and (ii) the cycles that are spent filling up the pipeline after the squash. Squash reuse alleviates both these components as follows. First, since it reuses work that was performed on the wrong path, not all cycles used in executing the discarded instructions are wasted. Second, when the reuse takes place just after the squash, it hides the pipeline-fill latency for the reused instructions, alleviating the second component.

1.1.2 Scenario 2: General Repetition and Reuse

General Repetition[1]: In this scenario, repetition occurs because of the very nature of programs — *i.e.*, because of the way programs are written. To understand this statement, let us consider two practices pervasively employed while writing programs and see how they may generate instruction repetition. First, we write programs to be *generic* in nature — *i.e.*, we don't write them for fixed input values; rather, we write them so that they are capable of operating on a variety of input values. But, if during execution, the program encounters the same

1. An anecdote: Our initial purpose for coming up with the instruction reuse technique was to reduce the branch mis-prediction penalty by recovering useful work from squashes. The fact that instructions are repeated in general was discovered, quite serendipitously, while we were studying the reuse techniques for the above purpose. In fact, after discovering that many non-squashed instructions also get repeated, I actually spent a considerable amount of time trying to filter out these "unwarranted" repetition to stop them from clouding the squash reuse results!

inputs values repeatedly (e.g., the same keywords in *gcc* or the same letters in *compress*) then it is likely that the instructions within the program will also execute with the same input values repeatedly. This will result in repetition of instructions. Second, we express computation in programs in a *concise* manner. That is, if we have to perform an operation on an array, we don't write a separate statement for each element of the array; instead, we express the task using a loop, where each iteration performs the operation on a single (or a small number) of array elements. To allow computation to be expressed in this manner, we need to include the loop-control instructions (instructions that will "unroll" the computation dynamically) with each loop. These loop-control instructions may be repeated (along with other dependent instructions) when the loop is invoked repeatedly. (This repetition may occur even when the loop body may be performing a totally different computation).

We illustrate the above situation with an example shown in Figure 1.2. In this example, the function `func` searches for a value `x` in a `list` of a particular `size`. The function `main_func` calls `func` several times, searching for a different element in the same list with each call. When `func` is called, it iterates through the `list`, element by element, searching for the value until the end of the `list`, and exits when the value is found. Instructions corresponding to the loop in `func` are shown in Figure 1.2(b). Figure 1.2(d) shows the dynamic instances of these instructions which are generated by the first call to `func`. In each iteration of the loop, instruction 2 is dependent upon the `size` parameter, instructions 3 and 4 are dependent upon the `list` parameter, instruction 5 is dependent upon the `list` as well as the value being searched for, and instruction 6 is dependent on the induction variable. If `func` is called again (Figure 1.2(e)) on the same `list` (and same `size`), but with a different search

```
int func(x, list, size) {
    int i;
    for(i=0; i<size; i++) {
    if(x==list[i]) return i;
    }
    return -1;
}
```

(a)

```
1  i = 0
2  if(i >= size) jump out
3  p = list + i
4  val = Memory[p]
5  if(x == val) jump found
6  i++
7  jump 2
```

(b)

```
main_func(a, b, c) {
  ...
  func(a, list, size);
  ...
  func(b, list, size);
}
```

(c)

```
*1  i = 0
*2  if(i >= size) jump out
*3  p = list + i
*4  val = Memory[p]
 5  if(a == val) jump found
*6  i++
*7  jump 2
*2  if(i >= size) jump out
*3  p = list + i
*4  val = Memory[p]
 5  if(a == val) jump found
*6  i++
    ...
```

(d)

```
*1  i = 0
*2  if(i >= size) jump out
*3  p = list + i
*4  val = Memory[p]
 5  if(b == val) jump found
*6  i++
*7  jump 2
*2  if(i >= size) jump out
*3  p = list + i
*4  val = Memory[p]
 5  if(b == val) jump found
*6  i++
    ...
```

(e)

Figure 1.2 Example illustrating that often times instructions perform the same computation over and over again. The dynamic instructions marked "*" would perform the same computation for both the calls to function `func` shown in the figure.

key, then all the different dynamic instances of instructions 1-4 and 6 will produce the same outcomes as they did the previous time the function was called.[2] Only the dynamic instances of instruction 5 produce results that might differ from the previous call to `func`. This *repetition* of the results of the dynamic instances of instructions 1-4 and 6 is directly attributable to the fact that `func` was written to be a generic list search function, but in this particular case, only one of its parameters changed between different calls to it. Even if `func` was called with all its parameters being different for each call, the different dynamic instances of the instruction 6 (i=0, i=1, i=2, ...) in the second call to `func` would end up producing the same values as they did in the first call to `func`, a consequence of using loops to express the desired computation in a concise manner. (Actually, if the `size` parameter was also different, then only *min*(`size1,size2`) dynamic instances of instruction 6 would produce the same values.).

Since, the form of repetition as exemplified above occurs because of the general nature of programs, we call it *general repetition*.

General Reuse: In the above example, if we buffered the (`size`) dynamic instances of instructions 1-4 and 6, we will be able to reuse them when they get repeated. This form of reuse that is enabled by general repetition, we call *general reuse.*

Benefits: To see how the performance might benefit from general reuse, let us consider the advantages of reusing instances of instructions 1-4 and 6 in the above example. First, the dynamic instances of instructions 1-4 and 6 do not have to pass through all the different phases of execution (ALU, result bus, register write, etc.), thereby reducing the demand for processor resources. (In the above case, accesses to the data cache are also eliminated — these

2. a total of `size` dynamic instances of instructions 2-4 and 6

end up becoming accesses to the buffer which holds previous instruction results.) Second, the critical path to carry out the total computation involved in `func` can be cut down considerably. Without dynamic instruction reuse, the critical path through the computation, as expressed above, would be `size+3` steps, `size` steps to generate all the dynamic instances for the induction variable `i`, plus 3 steps to executed instructions 3, 4, and 5 of each iteration (which form a dependence chain). In other words, the height of the dataflow graph for the above computation is `size+3` steps. In the best case, the critical path, *i.e.*, the height of the dataflow graph through the computation, is reduced to only 1 step with instruction reuse. This is because the outcomes of all the dynamic instances of instructions 1-4 are already known, and all the dynamic instances, being independent of one another, could all execute at the same time. Although, in practice, the available buffer space would place a limit on how much of the computation can be collapsed, the above example goes to show the potential that instruction reuse has for breaking the dataflow limit "inherent" in programs.

1.2 Thesis Contributions

In this thesis, we make two main contributions: (i) we study the phenomenon of instruction repetition; and (ii) we introduce and study the concept of dynamic instruction reuse. Each of these contributions are elaborated below.

1.2.1 Instruction repetition

We perform an elaborate study of the phenomenon of instruction repetition. The purpose is to develop a better understanding of the phenomenon, so that we can exploit it effectively.

The study consists of two parts. In the first part, we perform a thorough characterization of the phenomenon. Here we answer questions such as what percentage of all dynamic instructions get repeated, what percentage of all static instructions generate repeated instances, what fraction of static and dynamic instructions account for most of the repetition, and so on.

Although the above characterization gives us various statistical facts about the phenomenon, it does not provide us with much insight into its causes. In the second part of the study, we perform an empirical analysis of the phenomenon to better understand what may be causing it. For this purpose, we categorize the instructions in programs based on the type of data used (e.g., external input, internal data) and the type of work performed (e.g., address calculations, function prologue and epilogue), and then determine the amount of instruction repetition arising for each category. This breakdown gives us an idea about the primary sources of repetition, and thereby, its causes.

We draw numerous observations from our results. Of these, two are especially interesting. First, we observe that the phenomenon of instruction repetition is pervasive — more that 75% of dynamic instructions are repeated for several benchmarks. Second, we see that for most benchmarks, the majority of the repeated instructions use data that originate from within the program itself rather than from external inputs. This observation suggests that the phenomenon of instruction repetition may be more a property of the program itself than of input data.

1.2.2 Instruction Reuse

The second and the main contribution of this thesis is the concept of instruction reuse — *i.e.*, the idea that the previous work by instructions can be non-speculatively reused when they perform the same work again. The bulk of this thesis is devoted to developing and understanding this concept. The work on instruction reuse can be divided into three categories: (i) devising and studying reuse schemes; (ii) studying the storage issues for instruction reuse; and (iii) investigating the sensitivity of instruction reuse performance to other microarchitectural features existent in processors. We elaborate next on each of these categories.

1.2.2.1 Instruction reuse schemes

We present four schemes for implementing instruction reuse. These schemes preserve results of instructions (along with other information needed to establish their validity at a later time) in a hardware table called the *Reuse Buffer (RB)*. When an instruction is encountered again, its results from RB are reused if they are still valid. The validity of the results is established by checking whether the current operand values are same as those used to calculate the results. The four schemes differ in the type of information they use to establish the sameness of operands. The scheme S_v uses operand values; the scheme S_n uses operand names; and the schemes S_{v+d} and S_{n+d} use the dependence between instructions, along with the operand values and operand names, respectively. The use of dependence information facilitates the reuse of dependent chain of instructions.

We evaluate the concept of instruction reuse with extensive simulations. We present results such as the number of instructions reused and the amount of performance gained by reuse. We show reuse characteristics such as the reusability of different instruction types and the contribution of different instruction types to total reuse. We also present a break down of total reuse into general and squash reuse. Our results show that a significant percentage of dynamic instructions in programs get reused, with more than 50% of dynamic instructions getting reused in several cases, and that the performance improvement due to reuse is also significant for several benchmarks, being more than 15% in many cases.

1.2.2.2 Storage issues for instruction reuse

One important part of the instruction reuse technique is the RB, which stores the results of instructions. To a large extent, the number of instructions that can be reused depends on how many valid results the RB can hold. We conduct a detailed study to better understand the requirements of this structure. This study is divided into two parts. First, we characterize the RB with respect to its three main parameters: size, associativity, and management policies. We present how the amount of instructions reused varies with each of these parameters. We present the maximum reuse rates, obtained using a management policy similar to the Belady's optimal management policy, for a range of RB sizes and associativities. This optimal numbers give us an upper bound on the reuse rates for different RB sizes and associativities — or, alternatively, tells us the minimum RB size and associativity required to capture a certain amount of reuse.

The second part of this study is motivated by the results of the first part, which show that a significant gap exists between the optimal and the actual reuse rates. This gap indicates that there is potential to improve the reuse rate further by efficient management of the RB. To bridge this gap, we devise and study four management policies for managing the RB efficiently. Two of these polices, *FnReused* and *FnReady*, attempt to improve RB utilization by controlling insertions in RB — *i.e.*, by inserting only the likely reusable instructions. The third policy, *RR*, attempts to improve RB utilization by controlling eviction from RB — *i.e.*, by evicting the likely unreusable instructions before the reusable ones. The fourth policy, a novel management policy called *Farthest in Future (FiF)*, attempts to improve RB utilization by managing it along the lines of the Belady's optimal management policy. This policy determines how far in the future each instruction is likely to get reused. The RB is then managed by scheduling instructions in it using this distance-to-reuse information, giving priority to the instructions that have shorter distance values. The FiF is a general management policy that can also be used for managing other forms of storage, e.g., caches; however, in this thesis, we only use the FiF for managing the RB, leaving the task of evaluating it for other storage structures as future work.

The success of these new policies in improving RB utilization is mixed. For some benchmarks, we see a significant improvement (over the existing policies) in reuse rates using the new policies; for others the improvements with the new policies are small (or slightly negative). Due to its general nature, the FiF policy performs better than other policies in most cases. However, policies FnReady and RR may be comparatively inexpensive to implement, and, hence, the improvements in reuse rates caused by them may be noteworthy.

1.2.2.3 Sensitivity analysis

It is important to not only study how a new microarchitectural technique performs by itself, but also to understand how it interacts with other microarchitectural features. To cultivate such an understanding for instruction reuse, we study its sensitivity to various key processor parameters, such as (i) instruction window size, (ii) pipeline width, (iii) pipeline length, (iv) branch prediction accuracy, (v) memory latency, and (vi) reuse latency. This study consists of two parts. In the first part, we first present a detailed qualitative discussion on how and why instruction reuse may be sensitive to each of the parameters. In the second part, we present several simulation results to provide a quantitative measurement of the amount of sensitivity.

1.3 Related Work

The idea of not having to redo computation is not new — it has been used before in several different contexts. A technique called *memoization* [29, 5, 8] has been used for functional and logic programs [47, 30]. The outcome of a function (or a rule) is saved in a table. If the function or the rule is encountered again with the same parameters then the result from the table is used instead of re-evaluation. Memoization is also used to reduce the running time of optimizing compilers, where the same data dependence test is carried out repeatedly.

The observation that the instructions produce the same results repeatedly and that this phenomenon is widespread in ordinary programs has been made more recently by several researchers. Lipasti et al. [27, 26] observed that many instructions produce the same values as their last instance (or last few instances). They termed this recurrence of instruction results as value locality. Similar results were also reported by Mendelson and Gabbay [18, 19]. The phenomenon of instruction repetition — where not only the results but also the instruction operands are repeated — was first reported by us [43] (although the phenomenon was not termed as such in that paper).

Several researchers have studied the repetition of values elaborately. The study that we will present later in this thesis was first reported in [44]. Calder et al. [12] presented several statistical results on this phenomenon. Sazeides and Smith [40] tracked the creation, propagation, and termination of value locality in to better understand its causes.

Many ways of exploiting this phenomenon have also been proposed. Several researchers

[27, 26, 19, 39, 48] have suggested exploiting this phenomenon for predicting results of instructions in advance and performing dependent computation in parallel. We propose exploiting this phenomenon for reducing the amount of work that needs to be performed for executing instructions using instruction reuse technique. This technique was first reported by us in [43]. In the software arena, researchers have proposed exploiting this phenomenon using dynamic software optimizations, such as function memoization and code specializations [6, 17, 21], and static compiler optimizations, such as partial redundancy elimination [32, 10, 9].

Several researchers have performed work [23, 36, 37, 34] that is related to our method of exploitation. This prior work presents different techniques that obviate re-execution of repeating instructions by reusing their previous results. However, there are several important differences between our technique and theirs in terms of, for example, the ability to collapse the chain of dependent instructions or the type of instructions targeted. However, these differences and the mechanics of these techniques themselves can be best understood after we discuss the details of our technique. Hence, we defer the discussion on this subject until the *Related Work* section in Chapter 4.

Since our initial publication of the concept of instruction reuse [43], significant amount of work has been done in this area. The reuse concept has been extended to basic-block level [24] and trace-level [20]. Molina et al. [31] extended the instruction reuse technique to do computation reuse, *i.e.*, reuse of work done by other static instructions. Chou et al. [14] studied a different microarchitecture technique for performing squash reuse. Connors and Hwu [16] studied how compiler assistance can be used for reusing large regions of code. Reusability of instructions in other application domains, such as multi-media, has also been studied [15].

1.4 Thesis Outline

The rest of the thesis is laid out as follows. In Chapter 2, we describe our experimental framework. In Chapter 3, we present an empirical analysis of the phenomenon of instruction repetition. In Chapter 4, we present the instruction reuse technique. In Chapter 5, we present a characterization of the RB and study policies to manage it efficiently. In Chapter 6, we study the sensitivity of instruction reuse to other microarchitectural features. Finally, in Chapter 7, we summarize this thesis and provide directions for future work.

Chapter 2

Experimental Framework

In this chapter, we describe the experimental framework used in this thesis. All our experiments are performed using two processor simulators. We describe these simulators in the next section and mention the types of experiments for which they are used. Then, in Section 2.2, we describe the base processor microarchitecture that we simulate. Finally, in Section 2.3, we describe our suite of benchmarks and present some of their execution characteristics.

2.1 Simulators

The experiments in this thesis can be divided broadly into two categories: (i) one that studies program behavior and program structure for understanding instruction repetition (performed in Chapter 3); and (ii) the other that evaluates and studies the technique of instruction reuse (performed in Chapters 4 to 6). For each of these categories, we use a different type of simulator: a *functional* simulator for the former and a *timing* simulator for the latter. Both of these simulators are written in C, using several components from a preliminary version of the now publicly available *Simplescalar* toolset [11]. These simulators are execution-driven and they interpret an instruction-set derived from the MIPS-1 ISA [25]. We describe these simulators and their purpose in greater detail next.

2.1.1 Functional simulator

The functional simulator only models the architectural behavior of the processor at instruction level — *i.e.*, it simply reads and executes the instructions from the program binary (without modelling any microarchitectural details or any execution times).

This simulator is used for several purposes in our studies. As mentioned before, we use it in Chapter 3 for analyzing the dynamic behavior of programs — collecting various run-time statistics and tracking different sources of dynamic repetition. We also use the functional simulator in several other ways, such as, verifying the timing simulator on-the-fly, implementing perfect branch prediction, and skipping initial parts of the benchmarks during timing simulations; we discuss more about these uses later in this chapter.

2.1.2 Timing simulator

The timing simulator is used to evaluate and study the Instruction Reuse (IR) technique. This simulator models the microarchitectural behavior of an out-of-order, superscalar processor (shown in Figure 2.1) at the cycle-by-cycle level.

We developed the simulator for the out-of-order engine and the IR technique. These two components were then integrated with many other supporting components from the Simplescalar toolset — e.g., loader, memory-module, cache-module, branch predictors, and syscall-module — to give us a complete simulator. The design of the out-of-order simulator was influenced by the required support for IR. As we will see in Chapter 4, IR interacts with several parts of the pipeline. So that we can simulate these interactions faithfully, we model the base pipeline faithfully: instructions (and values) flow through the pipeline cycle-by-cycle and the various micro-operations are performed in appropriate pipe-stages (we do not fake them).

Also, to study squash reuse, we model the speculative execution and mis-speculation recovery faithfully: the processor is allowed to execute down the mispredicted path until the misprediction is detected and then the recovery is made by squashing the pipeline, just as it would be in a real pipeline.

For some experiments in Chapter 6, we require a perfect branch predictor — *i.e.*, a predictor that predicts all branches correctly. We implement the perfect branch predictor using the functional simulator. This simulator executes instructions before they enter the timing simulator, generating their results — and, hence, the branch results (when these instructions are branches) — in advance. We simulate perfect branch prediction by using the branch result generated ahead of time as predictions.

We also use the functional simulator to validate the timing simulator on-the-fly. Since the timing simulator is much more complex than the functional simulator, it is more susceptible to errors than the latter. Sometimes, these errors may cause the timing simulator to generate incorrect results. We detect such errors with the help of the functional simulator as follows. We run the timing and the functional simulator simultaneously. Every instruction that emerges from the timing simulator pipeline is executed on the functional simulator, and the results obtained from the timing simulator are compared with those obtained from the functional simulator. An error is detected when the two results are different.

The above validation only verifies the functional correctness of the timing simulator; it does not verify its timing correctness. Verifying that the timing results are correct is a hard problem, and there is no straightforward way of doing so. We take several steps to ensure that the timing results of our simulator are consistent:

- During the process of writing the simulator, we used several self-constructed micro-benchmarks for testing the behavior of the pipeline. Most of the micro-benchmarks were strings of 5 to 10 instructions, which were generated manually, and injected into the pipeline. This approach allowed us to verify that the flow of instructions through the pipeline was as would be expected from the dependences between them. This was helpful in validating the various pipeline interactions.

- In our simulator, we have several independent counters that count different events during execution (e.g., the number of instructions executed, the number of instruction squashed, etc.). We placed several *assertions* in our simulators to check the various invariances that can be derived from these counters, such as,

  1. # instr executed - # executed instr squashed = # instr committed + # executed instr in pipeline

  2. # loads from D-cache + # loads satisfied by stores in store-buffer - # executed load squashed = # loads committed + # executed load in the pipeline

  These invariances were checked every cycle. They helped link the different parts of the simulator, and ensured that changes made in one part were consistent with other parts. These invariances were especially useful in catching inconsistencies introduced due to fresh changes to the simulator.

- In our simulator, instructions actually flow through the pipeline; we do not fake the process. Such a design helps uncover many timing (or interaction) errors since it reduces the number of interaction errors that are completely silent (as may be the case if we were doing trace-based simulation). An error in pipeline interactions often causes wrong instruction or data to flow through the pipeline. This can either cause wrong execution that gets detected by our functional verifier or causes other forms of errors such as segment fault or deadlock, which again can be detected and fixed.

- Instruction reuse technique, for which the timing simulator is used, has a good property from the point of view of validation. It not only affects the timing but also the functionality of the simulator. (In this regard, it differs from other performance enhancing techniques, such as cache or prefetchers, which only affect the timing). If we reuse an instruction incorrectly, the processor will end up using an incorrect value, leading to incorrect program execution. This, again can be detected by either our functional verifier or through other non-silent ways such as deadlocks or segment faults. The error can then be fixed accordingly. The fact that all our simulations with IR were functionally correct lends high confidence to the reuse rate results.
- We implemented several *sanity-check* routines in our simulator that were called periodically (every 10,000 cycles) to check the consistency of the simulator data structures. For example, we check that the *number of unresolved branches == number of checkpoints taken for branch recovery*, or *depth of the retstack > number of returns in pipeline*. These sanity-check routines helped check the consistency of the data structures, making sure that the simulation was proceeding correctly.
- Finally, we performed extensive debugging using the debugger, *gdb*, single-stepping through every newly written piece of code to verify the timing information it generates.

Next, we describe the microarchitecture and parameters of the simulated processor.

2.2 Processor Microarchitecture

The pipeline and the microarchitecture of the processor that we simulated in our timing simulator are shown in Figure 2.1 (a) and (b), respectively. The pipeline consists of six stages: *Fetch*, *Decode & Rename*, *Register Read*, *Issue*, *Execute*, and *Commit*. All stages except the *Execute* stage are a single cycle in length; the *Execute* stage is of variable length, depending upon the latency of the executing instruction. In Figure 2.1 (a), we also depict the variable number of cycles that an instruction may have to wait before being issued. This pipeline structure is typical of currently available dynamically-scheduled processors (like, Pentium-III, HP-PA8500).

Next, we describe the various microarchitectural operations performed for processing an instruction. The *Instruction Fetch Unit* (IFU) fetches instructions from the I-cache (part of IFU) and places them in the *Instruction Queue* (IQ). The IFU also prepares the address of the

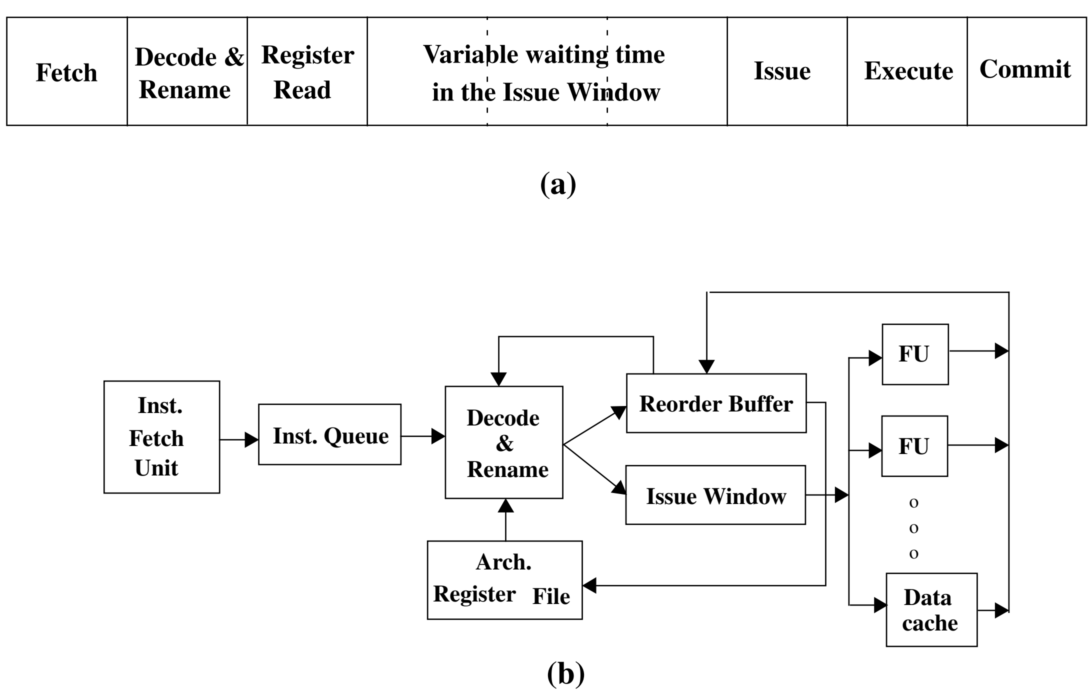


Figure 2.1 (a) Stages in the pipeline. (b) Microarchitecture modelled in the simulator.

next fetch using the branch prediction engine (also part of IFU). The instructions are read from the IQ by the *Decode and Rename* (D&R) unit. This unit decodes the instructions and renames their operands; it also allocates entries for them in the *Reorder Buffer* (ROB) [42] and the *Issue Window* (IW). In the *Register Read* stage, the register operand values are read from the architected register file or from the ROB, whichever contains the latest version of the register. The instructions and the operand values are placed in the pre-allocated IW entries; if an operand value is not ready, a tag (ROB index) identifying its producer is stored instead. This tag is used to snoop the value when it is broadcast after the producer finishes execution. The instructions are also placed in the pre-allocated ROB entries, where they await in-order retirement. The issue logic selects and dispatches *ready* instructions (*i.e.*, instructions whose all operand values are available) from the IW to Function Units (FU) for execution. Load instructions are issued to the data cache only when there are no store instructions with unknown addresses ahead in the pipeline. If a load address matches the address of a store ahead in the pipeline, the store value is bypassed to the load (and the data cache access for the load is obviated). After an instruction completes execution, its results are written back into its ROB entry and are also broadcast to the IW entries where they are snooped by the instructions awaiting these results. Instructions are retired when they become the head of the ROB and the architectural state of the machine (register file and memory) are updated with their results.

The baseline configuration of the timing simulator is shown in Table 2.1

Instruction fetch	4 insts per cycle. Only one taken branch per cycle. Cannot fetch across cache line boundaries in the same cycle.
L1 Instruction cache	64K bytes, 2-way set assoc., 32 byte line, 6 cycles miss latency.
Branch predictor	Gshare [28], with 10-bit history register and 16K entry counter table. Return Stack with 64 entries.
Out-of-Order execution mechanism	Issue of 4 operations/cycle, 64 entry RUU (which is the ROB and the IW combined) [46], 64 entry load/store queue. Max of 16 unresolved branches. Loads executed only after all preceding store addresses are known. Values bypassed to loads from matching stores ahead in the load/store queue.
Architected registers	32 integer, hi, lo, 32 floating point, fcc.
Functional units (FU)	4-integer ALUs, 2 load/store units, 2-FP adders, 1-Integer MULT/DIV, 1-FP MULT/DIV.
FU latency (total/issue)	int alu-1/1, load/store 1/1, int mult 3/1, int div 20/19, fp adder 2/1, fp mult 4/1, fp div 12/12, fp sqrt 24/24.
L1 Data cache	64K bytes, 2-way set assoc., 32 byte line, 6 cycles miss latency. Dual ported, non-blocking.
L2 cache	Perfect (all accesses hit)

Table 2.1 Base simulator parameters

2.3 Benchmarks

2.3.1 Description

The benchmark suite we use in this thesis consists of 21 programs: 8 SPEC ‘95 integer programs [4], 10 SPEC ‘95 floating-point programs, and 3 self-picked graphics programs. Table 2.2 shows the names and inputs for all the benchmarks. Since the three graphics benchmarks —*Viewperf+Mesa*, *MPEG-2 decoder*, and *POV-Ray* — are not as well-known as other benchmarks, we describe them further here. *Viewperf*, a benchmark developed by SPECopc^{sm} [2], measures the performance of graphics systems that implement the OpenGL$^{®}$ API [1] by

Benchmarks	Inputs	Total dynamic inst.	#dynamic inst. simulated	# initial dynamic inst. skipped
SpecInt ‘95				
go	null.in (ref)	35.7B	1B	500M
m88ksim	ctl.in (ref)	38.8B	1B	500M
ijpeg	vigo.ppm (train)	1.44B	944M	500M
perl	scrabbl.pl, scrabbl.in (train)	556M	556M	-
vortex	vortex.in (train)	2.67B	1B	500M
li	au.lsp puzzle0.lsp xit.lsp	10.2B	1B	500M
gcc	reload.i	921M	921M	-
compress	bigtest.in (ref)	42.3B	1B	2.5B
SpecFP ‘95				
tomcatv	train input with ITER = 50	2.44B	1B	500M
swim	train input with X=10, Y=10	849M	849M	-
su2cor	train input with LSIZE= 8 8 8 8	4.6B	1B	500M
hydro2d	train input with ISTEP=10	4.67B	1B	500M
mgrid	train input with NTIMES=1	368M	368M	-
applu	train input	642M	642M	-
turb3d	train input with nsteps=4	6.4B	1B	500M
apsi	train input	2.67B	1B	500M
fpppp	train input	499M	499M	-
wave5	train input	3.54B	1B	500M
Graphics				
Viewperf+Mesa	Viewset: AWadvs-02	2.58B	1B	500M
Mpeg-2 decoder	hhilong.m2v	1.80B	1B	500M
POV-Ray	swirlbox.pov	1.08B	1.08B	-

Table 2.2 Benchmarks

rendering and manipulating 3D images using this API. *Mesa* [35] is a publicly available implementation of the OpenGL API that we used with Viewperf. *MPEG-2 decoder* [33] decodes and plays a movie encoded in MPEG-2 video format. *POV-Ray* [3] is a scene rendering application that creates 3-D images, with realistic lighting effects, using a rendering technique called ray-tracing.

Some benchmarks have command line parameters other than the inputs specified in Table 2.2. These parameters are listed in Table 2.3.

2.3.2 Compilation

All C benchmarks were cross-compiled to the Simplescalar ISA using *gcc* (version 2.6.3) with the following optimization flags: -O3, -funroll-loops, and -finline-functions. The assembler and the linker used were *gas* (version 2.5.2) and *gld* (version 2.5), respectively. The Fortran benchmarks were first converted to C using AT&T's *f2c* program and then compiled using

Benchmarks	Command line parameters
m88ksim	-c
ijpeg	-compression.quality 90 -compression.optimize_coding 0 -compression.smoothing_factor 90 -difference.image 1 -difference.x_stride 10 -difference.y_stride 10 -verbose 1 -GO.findoptcomp
gcc	-quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -O
viewperf+mesa	-pg DYN -rm POLYGON -nf 10 -cp FRAME -zb -nll 2 -bf -tx advs2.mtv -magf LINEAR -minf LINEAR_MIPMAP_LINEAR -te MODULATE -xws 720 -yws 720 -grab grab1.scr
MPEG-2 decoder	-f -o0 rec%d
POV-Ray	-W320 -H200 -F +D -Q4

Table 2.3 Additional command line parameters for some benchmarks

the C compiler. To allow for graphics display, we cross-compiled and linked the X11 library with the graphics benchmarks.

Since we used the f2c translator it is likely that the Fortran benchmarks were not as optimized as they would have been had they been compiled using a Fortran compiler. This can affect our results is some ways. Due to (probable) inefficient compilation, these programs may contain redundancies. This may increase the amount of repetition we see. Moreover, the translation may add many “support” instructions in the binary, which may also increase the amount of repetition we observe. Consequently, the amount of reuse we capture may also be more than what we would see for Fortran benchmarks compiled with a Fortran complier. However, more instructions in the translated code can also hurt reuse rate, because with more instructions there will be more contention in the RB and the reusable instructions may be evicted before being reused. Unfortunately, we have no way of discerning the amount of inefficiency induced by using f2c. However, the points discussed above should be borne in mind while interpreting the floating point results in this thesis.

2.3.3 Execution

In this section, we describe how we run our simulations. In Table 2.2, we show the number of dynamic instructions present in a complete run of each benchmark (for the input shown in column 2). To finish the simulations within a reasonable period, we only simulate a part of the complete run for benchmarks with large dynamic instruction counts. The actual number of instructions simulated is shown in column 4 (# instructions simulated). To ensure that our simulated portion of the benchmark does not entirely consist of the initialization phase, we skip the first 500M instructions (for most benchmarks), executing them on a fast functional simulator, before simulating the 1B instructions. In Column 5, we show the number of instructions skipped for each benchmark. In the case of *compress*, we skip the first 2.5 billion instructions since *compress* has an unusually long initialization phase in which it internally generates the input file.

In Table 2.4, we present five baseline results for all benchmarks to show their relative characteristics. These results are base IPC (instructions per cycle), I- and D-cache miss rates, branch prediction rate, and the return stack hit rate (number of returns predicted correctly). These results were obtained using the base processor described in this chapter with the configuration shown in Table 2.1. Overall, most benchmarks have an IPC between 2 and 3. The I-cache misses are low for most benchmarks, except for *fpppp*. The D-cache misses are low for SpecInt '95 and graphics benchmarks (except for *compress*), but are relatively high for the SpecFP '95 benchmarks. The branch prediction rates and the return stack hit rates are high for most benchmarks, except for *go* and *ijpeg*.

In Table 2.5, we show the second set of inputs for our benchmarks. We use these inputs in Chapter 3 to investigate the sensitivity of the phenomenon of instruction repetition to program inputs.

Benchmarks	Base IPC	Cache Misses		Branch Prediction Rate	Return Stack Hit Rate
		I-Cache	D-cache		
SpecInt '95					
go	1.74	0.13%	0.89%	76%	100%
m88ksim	2.43	0.00%	0.01%	95%	100%
ijpeg	2.69	0.00%	0.63%	88%	99.9%
perl	2.46	0.00%	0.98%	96%	99.8%
vortex	2.78	0.19%	1.07%	98%	100%
li	2.30	0.00%	1.28%	96%	99.8%
gcc	2.05	0.29%	0.17%	91%	100%
compress	2.28	0.00%	9.56%	91%	100%
SpecFP '95					
tomcatv	2.87	0.00%	4.89%	98%	99.9%
swim	2.56	0.00%	5.97%	98%	100%
su2cor	2.34	0.00%	7.45%	94%	100%
hydro2d	2.40	0.00%	10.03%	99%	99.3%
mgrid	2.41	0.00%	2.23%	96%	100%
applu	2.69	0.00%	3.78%	93%	100%
turb3d	2.77	0.00%	1.74%	94%	100%
apsi	2.11	0.01%	1.11%	96%	100%
fpppp	1.63	4.04%	0.05%	94%	100%
wave5	2.21	0.00%	2.75%	97%	100%
Graphics					
Viewperf+Mesa	2.10	0.32%	0.97%	94%	100%
Mpeg-2 decoder	2.87	0.00%	0.76%	94%	100%
POV-Ray	2.16	0.59%	0.46%	94%	100%

Table 2.4 Base IPC, I- and D-cache misses, branch prediction rates, and the return stack hit rates, for all benchmarks. Cache misses are percentages over total cache accesses. Branch prediction rates are percentages over total number of dynamic conditional branches. Return stack hit rate are percentages over number of dynamic returns

SpecInt '95	Second set of inputs
go	2stone9.in
m88ksim	train.in
ijpeg	specmun.ppm
perl	primes.pl, primes.in
vortex	vortex.in (ref)
li	au.lsp tak2.lsp xit.lsp
gcc	amptjp.i
compress	test.in (train)

Graphics	Second set of inputs
Viewperf+Mesa	Viewset: CDRS-04
Mpeg-2 decoder	mei16v2.m2v
POV-Ray	mist.pov

SpecFP '95	Second set of inputs
tomcatv	ref input
swim	ref input with X= 100, Y= 100
su2cor	test input
hydro2d	test input with MPROW = 200
mgrid	test input
applu	test input
turb3d	test input with itest = 0
apsi	train input with x = 32, z = 8
fpppp	test input
wave5	test input with grid 625x20 particle dist 2500 50

Table 2.5 Second set of input for the benchmarks

Chapter 3

An Empirical Analysis of Instruction Repetition

In Chapter 1, we described the phenomenon of instruction repetition. Before we can discuss the methods for exploiting this phenomenon, we need to develop a better understanding of the phenomenon itself. We not only need to be aware of its various characteristics — such as percent of total dynamic instructions repeated, or groups of instructions generating most repetition — but we also need to understand the underlying causes that give rise to this phenomenon. Only after gaining such an understanding will we be able to exploit this phenomenon effectively.

To achieve this goal we perform two main tasks in this chapter: (i) we supply various characteristics regarding the phenomenon of repetition and (ii) we present an empirical analysis of instruction repetition to better explain what may be giving rise to this phenomenon. We begin by, first, qualitatively describing the causes of repetition and then introducing the different types of analyses we perform in this chapter.

3.1 Qualitative Description of Causes of Repetition

What causes instruction repetition? In Chapter 1, we briefly addressed this question and stated that instruction repetition occurs because of the repeating nature of input values and the structure of programs themselves. In this chapter, we elaborate further. To understand why program inputs and structure may cause repetition, let us consider how a typical program is written. In Figure 3.1, we show a piece of code: a function, *func*, that searches for an element *x* in an array *list* of size *size* (same as the code example used in Section 1.1.2). The structure of

```
int func(x, list, size) {
    int i;
    for(i=0; i<size; i++) {
     if(x==list[i]) return i;
    }
    return -1;
}
```

Figure 3.1 A code fragment to exemplify the typical structure of a program. This example also occurs in Figure 1.2 of Chapter 1.

this program reveals the following characteristics about the way we write programs: (i) we write programs to be *generic* in nature: *i.e.*, we often don't write them for particular input values only; instead we write them to be capable of handling a variety of input values (e.g., different values of *x* or *list* in the above function); (ii) we express computation *concisely* using loops — e.g., in the above case, we did not write a unique statement for checking each element of the list; instead, we wrote one static check statement and used a loop to apply it to every element on the list; and, (iii) we break our programs into separate modules, like functions, to simplify a complex task. Another common feature in programs (not exemplified explicitly in the above example) is data structures: we normally organize the program data, based on their logical grouping, into arrays, structs, lists, etc. For supporting such manners of programming, there exists several “extra” instructions in a program apart from the “computation” instructions which perform the actual task of the program. For supporting loops in programs, we have instructions that “run” the loop and help generate the dynamic program from the concise representation. To support the use of functions in programs, we have instructions that save and restore register state when entering and exiting a function. Similarly, to support complex data-structures, we need a fair amount of computation (and, hence, instructions) to access the individual elements in these data-structures.

How do these program characteristics engender repetition? We describe five ways. First, a program often encounters the same input data values repeatedly, causing the code which was written to be generic in nature to perform the same computation again. For example, programs that scan through text files (like *gcc*, *compress* and *grep*) may encounter repeated occurrences of the same items such as words, spaces, and characters. In the example shown in Figure 3.1, the function may be called repeatedly to search for different elements in the same list, which may result in instructions operating on the list value to perform the same computation repeatedly. Second, the loop-control instructions, which perform the task of unravelling the concisely expressed computation, may get repeated when the job of unravelling the computation is performed repeatedly for different invocations of the same piece of code (e.g., the loop control instructions would get repeated when the same loop is invoked again). These instruction would get repeated even if the computation performed is entirely new. Third, the instructions devoted to accessing the elements of a complex data-structure may get repeated when the same elements are accessed repeatedly, even if the value being accessed is different. Fourth, the register save-restore code of a function may get repeated if the registers do not change between two calls to the function. Finally, many instructions in programs have immediate values as operands, e.g., simple initialization instructions or groups of instructions loading a large constant in a register. These instructions (and instructions dependent on them) get repeated on re-execution since their operands are constants.

3.2 Quantitative Analyses: Introduction and Rationale

After identifying the causes of repetition qualitatively, we are now ready to analyze the phenomenon quantitatively. But, at this point, we are faced with the dilemma as to what sort of analysis should we conduct. What sort of investigation would satisfyingly reveal the nature of the phenomenon? Unfortunately, a direct answer to this question in not possible, at least not at present, and we make no attempt to obtain such an answer. Instead, we perform several different types of analyses, each providing a different way of looking at the phenomenon, and, hopefully, all together providing a better understanding about the nature of the phenomenon. These analyses fall into two broad categories: a category that attempts to characterize the phenomenon and a category that attempts to isolate the contribution of different “parts” of programs to the phenomenon.

To characterize instruction repetition, we carry out analyses similar to what others [12, 39] have carried out for related phenomenon: we analyze the instructions of a program as a whole. We call this a *statistical* analysis. Here we ask questions of the form: how much repeatability exists? how many static instructions account for a certain fraction of the repeatability? etc.

While a statistical analysis allows us to characterize the phenomenon, it fails to give us insight into the causes of instruction repetition. Answers to questions of the form: how much of the repeatability is due to repeated inputs? how much can be attributed to instructions that unwind the dynamic computation? etc. are not available. To answer these questions, we need to categorize both the instructions that are executed as well as the instructions that are repeated, into different classes (e.g., instructions that operate upon external inputs, or those that operate upon global variables).

Categorizing instructions into different classes requires us to capture dynamic *slices of instructions*, that is, dynamic paths through programs traced by the flow of data (e.g., a slice of instructions executed in a function that depends upon its first argument). In capturing slices of computation, we are faced with the question of whether to consider data dependences, control dependences, or both. Control dependences determine *which* static instructions are entered into the dynamic instruction stream, and data dependences determine the outcome of those instructions. Since our purpose is to understand the repetitive behavior of instructions that are present in the dynamic instruction stream and not with how static instructions are entered into the dynamic instruction stream, we do not consider control dependences when dividing the dynamic instruction stream into dynamic slices. We base our decisions and analysis solely on data dependence relationships (in fact, the notion of a control dependence is somewhat meaningless in a *dynamic* instruction stream).

In the next section, we formally define the various terms used in the remainder of the thesis. Then, in Section 3.4, we briefly describe the experimental setup that is specific to the experiments performed in this chapter. In Section 3.5, we characterize instruction repetition, and in Section 3.6 we analyze the sources of repetition. In Section 3.7, we discuss and further investigate some of the results presented in this chapter, and, finally, we summarize and conclude this chapter in Section 3.8.

3.3 Definitions

In this section, we define three terms that we use in this thesis: *dynamic instruction repetition*, *static instruction repetition*, and *unique repeatable instance*. First we define them informally and then follow it with more formal definition.

We start out by defining *dynamic instruction repetition*. Repetition occurs when different dynamic instances of the same static instruction have repeated outcomes. An instruction can generate a repeated outcome if its operands are repeated (the common case). However, the outcome of an instruction can be repeated even if its operands are not repeated (e.g., the outcome of a compare instruction can be the same with vastly different inputs). In some cases, the result of an instruction may not be repeated even if its operands are repeated, because of the side effects of other instructions (e.g., a load instruction reading different values from the same memory address). In this thesis, we say that (a dynamic instance of) an instruction is *repeated* if both the inputs and the outputs of the instruction are repeated; *i.e.*, the instruction produces the same outputs for the same set of inputs as a previous instance of the instruction. At places in this thesis, we use the term *repeatability* to mean the phenomenon of instruction repetition.

We further clarify the concept of repetition. First, consider an instruction ‘I’ that executes with operand values *v1* and *v2* and produces *v3*. Then a later (not necessarily the next) instance of ‘I’ will be considered repeated if that instance also executes using *v1* and *v2* as inputs and produces *v3* as output. Second, a load (like any other instruction), is considered repeated when its operand values (which are used to compute the load address) and the result (the value loaded from memory) are the same as some earlier instance of the load. Finally, a store is considered repeated when its operand values (which are used to compute the store address) and it store value (the value that will be stored to memory) are both the same as in some earlier instance of the store.

The above definitions explain what is meant by a repeated dynamic instruction. Now, we state the meaning of *a repeated static instruction*. <u>A static instruction is said to be repeated if it generates at least one repeated dynamic instruction.</u>

Next, we define a *unique repeatable instance.* A unique repeatable instance is the basic dynamic instance (of a static instruction) that gets repeated. For example in Figure 3.2 the

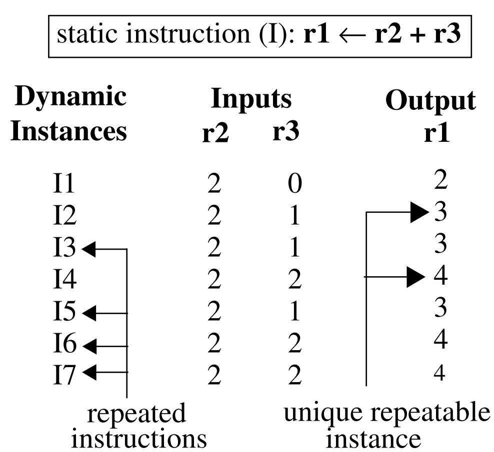


Figure 3.2 Unique repeatable instances — *basic* dynamic instances which get repeated.

static instruction (I) generates seven instances. The instances I2 and I4 are the first (hence unique) occurrence of the instance that gets repeated subsequently as I3, I5, I6, I7. We call I2 and I4 *unique repeatable instances*. Note that I1 does not fall in this category (although it is unique) because it does not get repeated.

Finally, we restate the above definitions more formally using the following notations. Let $I(PC)$ stand for a static instruction at the address PC, and let $I_i(PC)$ stand for the *ith* dynamic instance of that static instruction. A dynamic instance of a static instruction can also be represented with the following, more detailed, notation: $I_i(PC, op1_i, op2_i, res_i)$, where i denotes the *ith* instance of I; $op1_i$ and $op2_i$ stand for the operand values and res_i stands for the result value. If $I_i(PC)$ and $I_j\ (PC)$ are two dynamic instances and $i < j$ then it means that $I_i(PC)$ occurs earlier in program order than $I_j(PC)$. Two dynamic instances are said to be equal, *i.e.*, $I_i(PC, op1_i, op2_i, res_i) == I_k(PC, op1_k, op2_k, res_k)$, if they are instances of same static instruction and $op1_i == op1_k$, $op2_i == op2_k$, $res_i == res_k$. With these notations, the three definitions presented earlier in this section can be restated more formally as follows:

Dynamic instruction repetition:

A dynamic instruction $I_j(PC)$ is said to be repeated if $\exists i \mid i < j$ and $I_i(PC) == I_j(PC)$

Static instruction repetition:

A static instruction $I(PC)$ is said to be repeated if $\exists i$ and $\exists j \mid I_i(PC) == I_j(PC)$

Unique repeatable instance:

A dynamic instruction $I_i(PC)$ is said to be a unique repeatable instance

if $\exists j \mid i < j$ and $I_i(PC) == I_j(\mathrm{PC})$, but $\nexists h \mid h < i$ and $I_h(PC) == I_i(PC)$

3.4 Experimental Setup

The experimental setup used for this study is described in Chapter 2. We use our functional simulator to execute the benchmark program and perform the analysis during execution.

To track instruction repetition, we buffer each new instance of a static instruction that is generated during the course of execution. An instance is considered repeated if it uses the same operand values and produces the same result as one of the previously buffered instances of the same instruction. We buffer up to 2000 unique instances (*i.e.*, instances that use different input values or produce different output value) per static instruction for each benchmark and perform LRU replacement when additional unique instances are encountered.[1]

In Table 3.1, we show the number of dynamic instructions executed (column 2), the number of static instructions present (column 4), and the percentage of static instructions executed (column 5) for each benchmark (other columns in this table are discussed in the next section).

Since the analysis was performed only on a portion of a program, it is likely that the results of the analysis are not representative of the whole program run. To address this issue, we simulated the programs for 10 billion instructions[2] (or until completion) and collected the statistics on *overall local* analysis. (We discuss what this analysis is and its purpose later in the chapter.) The statistics from the long simulations tallied with those obtained (and presented later) from the short simulations. Although this verification does not necessarily imply that the results of the analysis are representative of the complete run, they serve to give us more confidence in our results.

Since the phenomenon we are analyzing is dependent on the properties of data, it is reasonable to suspect that the results may be sensitive to the program inputs chosen. As men-

1. A version of this work presented earlier at ASPLOS '98 [44] did not use any replacement policy. It only buffered the first 2000 unique instances and ignored all unique instances thereafter. This difference causes some results in the two studies to differ slightly (e.g., percentage of instructions repeated, number of unique repeatable instances observed, etc.), with the results in [44] generally being a more conservative evaluation of the phenomenon than those presented here.

2. We didn't have to track repetition for these experiments, and hence both the time and memory requirements were small.

tioned in Chapter 2, we ran similar experiments using other program inputs (shown in Table 2.5) and found similar trends with the second set of inputs. (Some results for the second set of inputs is shown in Appendix A.) In this chapter we present results only for the inputs shown in Table 2.2.

3.5 Statistical Analysis: Characterizing Instruction Repetition

In this section, we attempt to get a feel for the characteristics of repeatability in the program as a whole (*statistical* analysis). In our first set of data, we try to get a feel for how much instruction repetition exists and how many program instructions contribute to repetition. Table 3.1 shows the repetition results for all benchmarks. The second column (*Total*) shows the number of instructions that were executed dynamically, and the third column (*Repeat*) shows the percentage of dynamic instructions that were classified as repeated. In general, we see that a significant percentage of dynamic instructions get repeated, especially for the integer and graphics benchmarks, where the dynamic repetition rate is greater than 70% for all benchmarks, except for *compress* (where the repetition rate is 57%). For the floating-point benchmarks the repetition rates are comparatively lower, ranging between 40-70%, except for *mgrid* (19%), *fpppp* (37%), and *turb3d* (90%).

The remaining columns of Table 3.1 deal with static instructions. In the fourth column we show the number of static instructions present in each program. In the last two columns, we present the percentage of static instructions that get executed (*% of Total*) and the percentage of executed static instructions that show repetition (*% of Exec*). We observe that only a small

Benchmarks	Dynamic Instructions		Static Instructions		
	Total (millions)	Repeat (%)	Total	Executed	Repeated
				% of Total	% of Exec
SpecInt '95					
go	1000	93.8	84,552	62.9	93.4
m88ksim	1000	98.8	37,824	4.5	97.7
ijpeg	942.2	79.8	58,894	25.4	98.1
perl	555.6	85.1	73,850	22.3	65.7
vortex	1000	96.6	125,018	28.3	93.5
li	1000	89.9	23,026	8.4	99.8
gcc	421.4	88.2	299,988	39.5	87.7
compress	1000	57.7	13,798	13.1	66.3
SpecFp '95					
tomcatv	1000	56.2	20,926	17.1	89.1
swim	849	49.2	22,154	42.3	74.8
su2cor	1000	49.5	36,872	39.9	89.6
hydro2d	1000	43.9	32,084	24.0	98.0
mgrid	368	19.3	23,264	44.0	75.9
applu	642	60.9	34,130	60.4	70.8
turb3d	1000	90.0	35,904	16.9	80.7
apsi	1000	65.9	63,134	30.9	90.9
fpppp	499	36.6	44,170	65.6	83.5
wave5	1000	61.0	65,448	12.1	57.0
Graphics					
Viewperf+Mesa	1000	77.4	775,758	1.0	93.3
Mpeg-2 decoder	1000	73.1	22,902	23.0	94.3
POV-Ray	1081	82.5	263,148	17.1	56.5

Table 3.1 Table shows the benchmark programs, the total dynamic instructions executed and the percentage of dynamic instructions repeated. It also shows the total static instructions in each program, and the percentage executed and repeated.

fraction of the total static instructions get executed dynamically but a large fraction of those executed are repeated. This trend is true for all the benchmark programs. Thus, repetition is not a phenomenon which is exhibited by only a small fraction of the static instructions that are executed. However, a few static instructions might be accounting for a large number of repeated instructions, and we study that next.

In Figure 3.3, we show the percentage of the repeated static instructions which account for a certain fraction of the total dynamic repetition. We observe that for all but five benchmarks less than 20% of the repeated static instructions account for more than 90% of the dynamic repetition. The exceptions are *m88ksim* (56%), *applu* (22%), *fpppp* (29%), *wave5* (27%), and *viewperf* (35%). For *m88ksim, wave5*, and *viewperf*, although the corresponding percentages of the repeated static instructions are higher, the absolute number of repeated static instructions in these cases is small to begin with.

Table 3.1 and Figure 3.3 suggest that many instructions are repeated but do not tell us how many different values generated by the instructions contribute to the repeatability. We measure this next. In Figure 3.4, we show for all three benchmark groups the contribution of instructions with a certain number of *unique repeatable instances* (defined in Section 3.3) to the overall dynamic repeatability. For example, in *go*, 25% of the dynamic repeatability is due to instructions with 1 unique repeatable instance and another 12% is due to instructions that have 2-10 unique repeatable instances. We observe that repetition is not limited to instructions producing few unique repeatable instances only. Instructions which produce many unique repeatable instances also account for a sizeable amount of the dynamic repetition (except in *tomcatv*). For example, instructions producing between 101 and 1000 unique instances account for 47% of the repetition in *ijpeg*, 28% in *li*, 55% in *hydro2d*, and 18% in *pov-ray.*

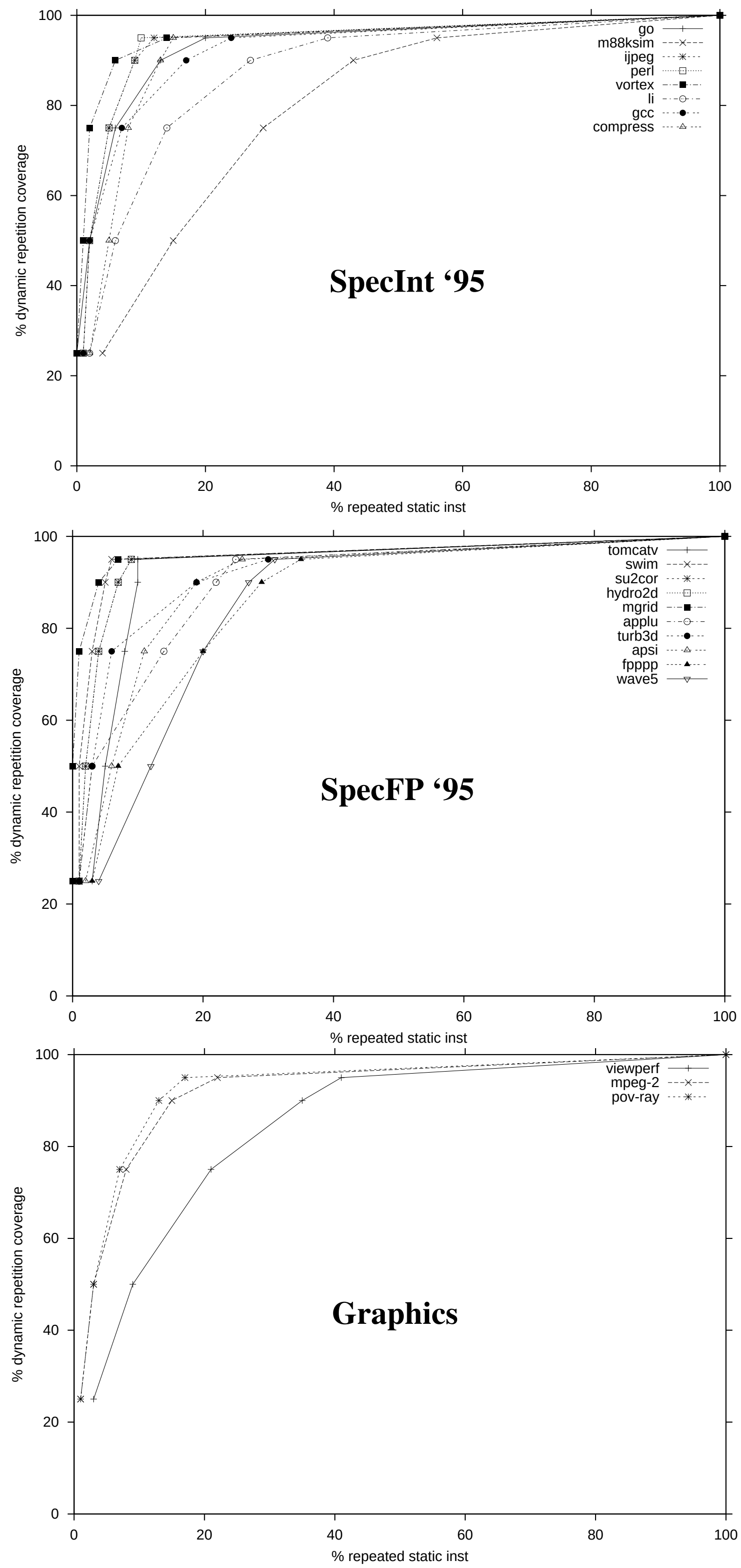


Figure 3.3 Static instructions coverage of dynamic repetition. This graph shows that very few (less than 20% for most cases) of the static instructions which get repeated generate most (more than 90%) of the repetition observed dynamically.

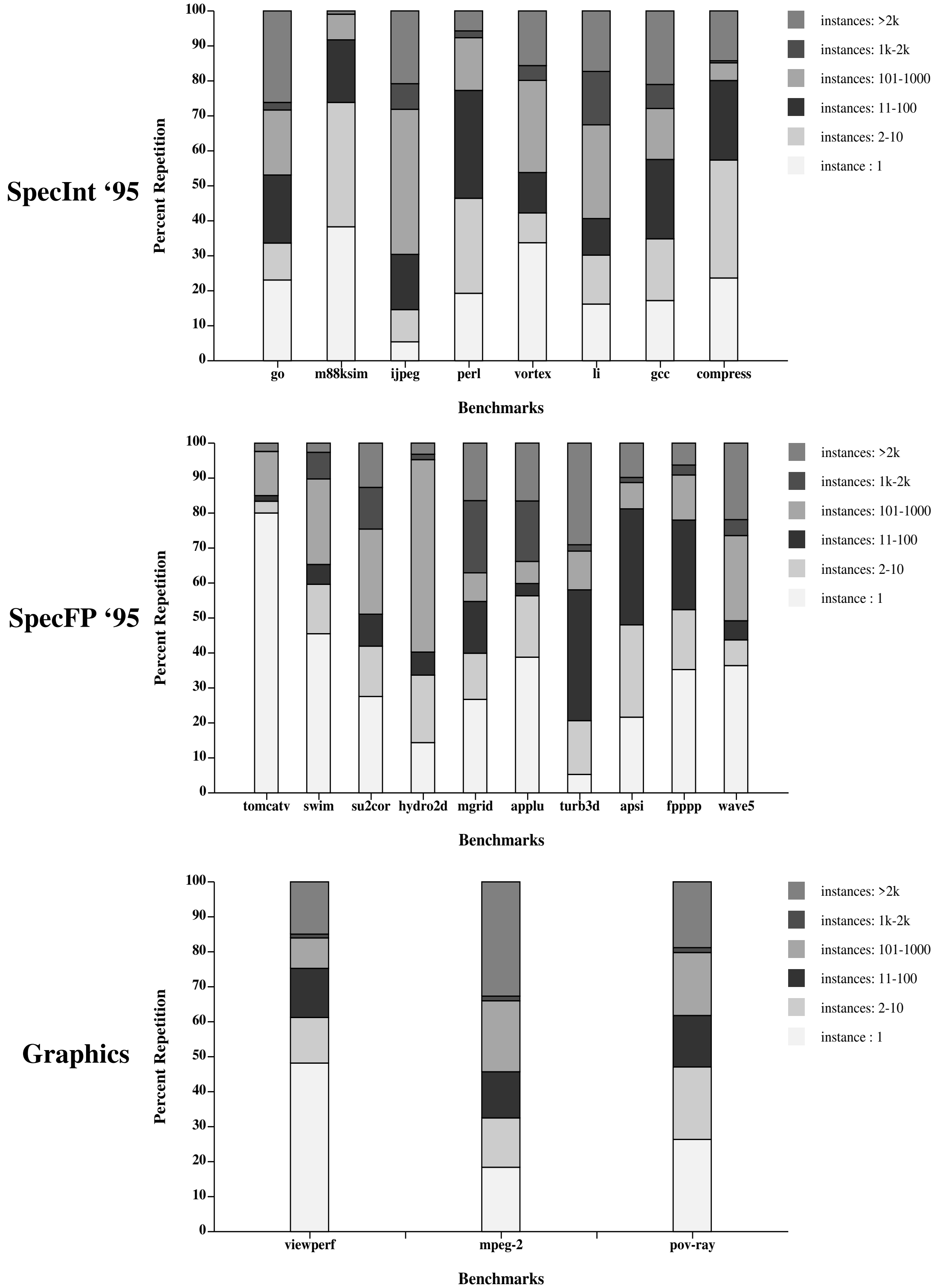


Figure 3.4 Contribution to total dynamic repetition of static instructions generating different number of unique repeatable instances. As seen, repetition is not limited to instructions generating few unique repeatable instances only, e.g., significant repetition is seen from instructions which generate between 100 to 1000 unique repeatable instances.

SpecInt Benchs	Unique Repeatable Instances		SpecFP Benchs	Unique Repeatable Instances		Graphics Bench	Unique Repeatable Instances	
	Count	Avg. Repeats		Count	Avg. Repeats		Count	Avg. Repeats
go	22,595,020	42	tomcatv	1,041,603	539	Viewperf	45,471,952	17
m88ksim	103,354	9555	swim	5,886,407	71	Mpeg-2	17,036,206	43
ijpeg	24,278,066	31	su2cor	21,012,330	24	POV-Ray	27,305,631	32
perl	2,209,087	214	hydro2d	734,890	597			
vortex	7,476,760	129	mgrid	1,380,997	51			
li	8,805,941	102	applu	20,142,921	19			
gcc	17,749,001	21	turb3d	11,884,880	76			
compress	18,498,014	31	apsi	19,483,424	34			
			fpppp	7,003,833	26			
			wave5	35,175,001	17			

Table 3.2 Number of unique repeatable instances and average number of repetitions for each.

This suggests that we need to track multiple repeatable instances of instructions in order to capture a large fraction of the repeatability in a program.

To get a feel for the total number of instruction instances we need to track in order to capture a certain fraction of the repeatability, we turn to the data in Table 3.2 and Figure 3.5. In Table 3.2, we show the number of unique repeatable instances (column *count*) in the program (the sum of all the unique repeatable instances of all instructions that are repeated). We also show the average number of times that a repeatable instance is repeated (column *Avg. Repeats*). The numbers show that all the observed repetition is generated by relatively few unique repeatable instances and that a unique repeatable instance gets repeated several times on average.[3] In Figure 3.5, we show the fraction of the unique repeatable instances that

3. The number of unique repeatable instances reported here are higher than what the actual count would be. This is because instances get evicted from the 2000-entry repetition-tracking buffer. An evicted instance when re encountered is again counted as a unique repeatable instance.

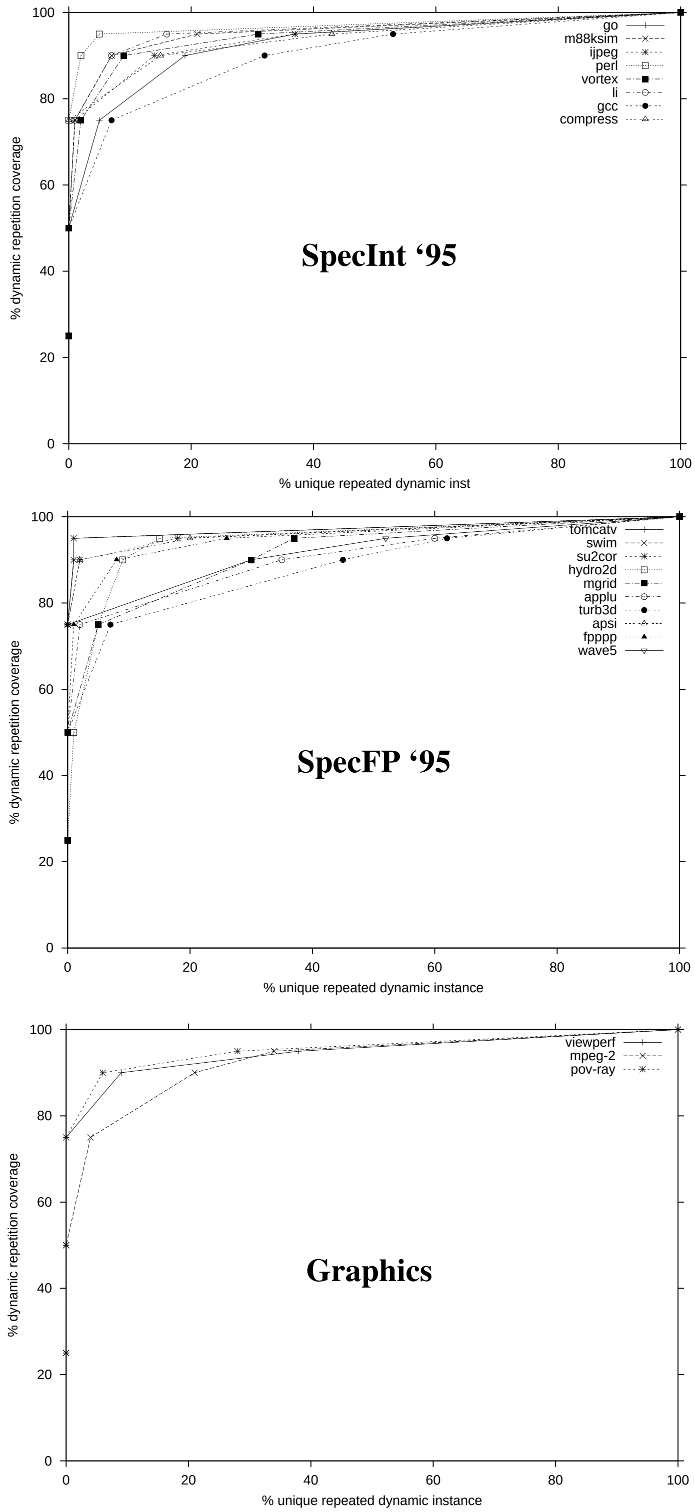


Figure 3.5 Coverage of repeatability by the unique repeatable instances shown in Table 3.2. For example, in most cases 80% of the repeatability is generated by less than 20% of the instances shown in column 2 of Table 3.2.

account for a certain fraction of the dynamic repetition. We observe that in most of the cases, less than 20% of all the repeatable instances account for more than 80% of the dynamic repetition.[4]

3.6 Analysis to Understand the Causes of Repetition

Having performed the statistical analysis, we now address the second purpose of the chapter: to understand the causes of repetition. Ideally, we would like to identify the repeatability due to a particular program function, e.g., which instructions in the dynamic execution of a program correspond to addressing a particular data structure, and how many of these instructions are repeated? But, what is the best way to breakdown programs for our analysis purposes? To overcome this dilemma, we analyze the programs at three different levels —*global* (whole program), *function-level*, and *local* (within functions). At each level we divide the programs into categories that intuitively seem to provide useful information at that level.

3.6.1 Global Analysis

At the global level, we can classify program instructions into three broad categories: (i) instructions whose inputs are influenced by external program input, or *external input* instructions, (ii) instructions whose inputs are influenced by initialized global variables, or *global init*

4. The dynamic repetition coverage numbers from specint'95 programs presented here differ from those presented in [44]. The difference stems from the way the dynamic instance were sorted before determining the coverage. In [44] we used an approximate method: we determined the number of repeated and number of unique instances generated by each static instruction. Then we calculated the average repetition per unique instance for each static instruction. We then sorted the list based on this average before accumulating the coverage. In this thesis, on the other hand, we employ a more accurate method. We keep track of the number of repeated instances generated by each unique dynamic instance during simulation, and use this number to sort the list of unique repeatable instances. In any case, this difference does not change the basic conclusion drawn from the result — *i.e.*, very few unique repeatable instances generate most of the observed repetition.

data instructions, or (iii) instructions whose inputs are influenced solely by *program internals.* (Instructions classified as program internal either operate upon immediate values, or [transitively] operate upon values generated by instructions that operate upon immediate values.) Sometimes instructions use uninitialized registers: for example, when an uninitialized callee-saved register is saved on a function entry. We classify such instructions in a separate (fourth) category called *uninit*.

To perform the analysis, we trace the flow of data through the program during execution. We tag each data item with the category name to which it belongs and propagate these tags along with the data to the dependent instructions. This propagation traces slices of instructions for each source category. The category of an instruction is determined by the categories of its input operands. We use a supersede rule, *external input* $>_s$ *global init data* $>_s$ *program internal* $>_s$ *uninit*, to determine the category of an instruction where two slices with different categories meet. In this rule, A $>_s$ B (A supersedes B) implies that if slices of A and B meet, the resultant slice will be that of A. We chose this rule to assign higher priority to a source that is likely to be "less repeatable".

We present the results of this analysis in Table 3.3, which consists of three tables, one for each of the benchmark groups — integer, floating-point, and graphics. For each benchmark group, we show three types of results: *overall*, *repeated*, and *propensity.* The overall results show the percentage of all dynamic instructions in each of the categories; the repeated results show the percentage of all repeated dynamic instructions in each of the categories; the propensity results show the percentage of dynamic instructions belonging to each category that got repeated (*i.e.*, amenability or propensity of each category to repetition, hence the name). More precisely, the propensity value of a particular category is

*(# of repeated instructions in the category)*100/(# of total dynamic inst. in the category),*

where the numerator is obtained from the repeated results and the denominator is obtained from the overall result.

Overall Results: In general, the results are different for the different groups. For integer benchmarks, we see that for most of the instructions (more than 50% in all benchmarks except *perl*) the inputs come from slices which originate from program internals (e.g., initialization statements). About 12% to 30% instructions inputs come from slices which originate from global initialized data. Also, for most integer programs, less than 20% of the dynamic instructions use values that come from slices which originate from external input. This shows that most of the computation performed in the program is on the data internal (or “hardwired”) to the program. This should not come as a surprise: in addition to the “computation” instructions themselves that operate on data values, programs contain a lot of “overhead” instructions, such as instructions that perform addressing and program control. This observation also serves as a basis for decoupled architectures that divide the instruction stream into an addressing stream and a computation stream [41].

In contrast to integer benchmarks, floating-point and graphics benchmarks have a much larger percentage of dynamic instructions that use values coming from slices which originate from external inputs. Except for *mgrid* and *turb3d*, in all floating-point benchmarks most dynamic instructions obtain their inputs from *external inputs* slices (e.g., 84% for *apsi*, 79% for *fpppp*, 57% for *applu*). The same is true for the two graphics benchmarks, *viewperf* and *pov-ray*, with 59% and 48% of dynamic instructions in this category, respectively. However, even for these two benchmark groups, the data “hardwired” in programs (*internals*+*global init*

Categories	go	m88k	ijpeg	perl	vortex	li	gcc	comp
Overall	% of all dynamic instructions							
internals	86.3	54.6	63.2	46.1	53.6	47.0	59.8	68.5
global init data	13.8	26.3	20.3	18.8	28.5	12.4	25.1	29.5
external input	0.0	19.0	16.5	33.6	17.9	39.8	15.1	2.0
uninit	0.0	0.1	0.0	0.5	0.0	0.8	0.1	0.0
Repeated	% of all repeated dynamic instructions							
internals	86.2	54.4	60.5	51.6	54.3	48.1	63.1	77.1
global init data	13.8	26.2	22.4	22.1	28.8	13.8	27.2	22.9
external input	0.0	19.3	17.2	25.8	16.9	37.3	9.6	0.0
uninit	0.0	0.1	0.0	0.6	0.0	0.9	0.1	0.0
Propensity	% of all dynamic instructions in each category							
internals	93.7	98.5	76.4	94.2	97.8	92.0	42.6	65.0
global init data	94.4	98.3	87.9	98.8	97.8	99.6	43.8	44.7
external input	97.1	99.9	83.3	64.5	90.8	84.3	25.9	0.0
uninit	98.7	100.0	99.3	99.0	99.0	100.0	43.6	60.6

(SpecInt)

Categories	tomcatv	swim	su2cor	hydro2d	mgrid	applu	turb3d	apsi	fpppp	wave5
Overall	% of all dynamic instructions									
internals	36.7	33.1	16.8	44.2	4.1	24.9	12.9	7.8	8.4	21.9
global init data	26.1	10.8	32.5	10.0	95.9	18.1	87.1	8.0	12.9	30.9
external input	37.2	55.8	50.7	45.8	0.0	57.1	0.0	84.0	78.7	47.2
uninit	0.0	0.4	0.0	0.0	0.0	0.0	0.1	0.2	0.0	0.0
Repeated	% of all repeated dynamic instructions									
internals	58.9	35.0	33.0	36.7	13.8	38.8	13.6	11.9	22.9	34.7
global init data	31.8	21.9	43.4	19.6	86.2	23.9	86.4	12.1	35.2	25.5
external input	9.3	42.4	23.7	43.7	0.0	37.3	0.0	75.8	41.9	39.9
uninit	0.0	0.8	0.0	0.0	0.0	0.0	0.1	0.3	0.0	0.0
Propensity	% of all dynamic instructions in each category									
internals	90.1	52.1	97.0	36.4	65.8	94.9	94.8	99.9	99.8	96.6
global init data	68.3	99.9	65.9	86.0	17.3	80.5	89.2	99.5	99.6	50.3
external input	14	37.4	23.1	41.9	86.0	39.8	99.7	59.5	19.5	51.5
uninit	99.9	99.9	97.2	56.3	76.8	96.1	99.9	99.9	63.2	0.0

(SpecFP)

(Graphics bench on the next page)

Table 3.3 Breakdown in terms of sources of input: program internals (constants), global init data, external input, and uninit. Overall shows the breakdown of the complete program. Repeated shows the break down of the repeated instructions. Propensity shows the percentage of dynamic instruction in each category that got repeated.

Categories	viewperf	mpeg-2	povray
Overall	% of all dynamic instructions		
internals	24.6	67.5	29.8
global init data	15.1	13.0	20.2
external input	58.8	19.5	47.7
uninit	1.5	0.0	2.3
Repeated	% of all repeated dynamic instructions		
internals	31.7	69.3	33.7
global init data	18.9	15.0	24.9
external input	47.4	15.8	38.6
uninit	2.0	0.1	2.9
Propensity	% of all dynamic instructions in each category		
internals	99.6	75.0	91.3
global init data	97.0	84.3	99.7
external input	62.4	59.2	65.3
uninit	100.0	100.0	99.4

(Graphics)

Table 3.3 (continued) Breakdown in terms sources of inputs for graphics benchmarks.

data) provide inputs to significant percentage of dynamic instructions. For example, in five floating-point benchmarks (*tomcatv*, *hydro2d*, *mgrid*, *turb3d*, and *wave5*) and in all graphics benchmarks, the categories *internals* and the *global init data* taken together constitute the biggest source of input data for dynamic instructions, corroborating our earlier observation that a significant portion of the total computation takes place on data that is internal to the programs.

Repeated Results: The distribution of the repeated results across the global categories is similar to that of the overall results: *i.e.*, the amount of repetition seen in a particular category is commensurate with the amount of computation in that category.

For integer benchmarks, most repetition comes from the "hardwired" part of the program, *i.e.*, most of the instructions which get repeated operate on the data that is internal to the program. This suggests that repeatability may be a phenomenon inherent to the way programs are expressed and less sensitive to the external input. (As mentioned earlier, we have observed similar results using different input files for the programs.) This is also true for floating-point and graphics programs, but in these cases the external input category also contributes significantly to the total repetition (e.g., *apsi*, *hydro2d*, *viewperf*, *pov-ray*). Since much of the repetition occurs because of the "hardwired" part of the program, it would appear that compiler should have eliminated this "redundancy" in the first place. We defer the discussion on this issue until Section 3.7.

Propensity Results: We see a significant percentage of dynamic instructions in each category get repeated. As expected, both *internals* and *global init data* show a high propensity for repetition (greater than 80% for most cases). The *external input* category in general shows a comparatively low propensity for repetition (for most floating-point programs, it is less than 50%). However, in a few cases — e.g., *m88ksim* (99%), *vortex* (86%), and *pov-ray* (65%) — a significant percentage of instructions in the *external input* category get repeated. Although some benchmarks such as *go, compress, mgrid,* and *turb3d,* show a high propensity for repetition in the external input category, we point out that there are very few instructions in this category for these benchmarks. Similarly, even though the percentages for *uninit* are high, we again note that this category has very few instructions (compared to the other categories).

3.6.2 Function Level Analysis

Functions (or procedures) are a common way of expressing a computation that gets invoked repeatedly. Often they are written to be general purpose (parameterized by arguments), and a specific task is performed by invoking them with argument values appropriate for that task. One reason why repetition occurs is because functions often get invoked repeatedly with the same argument values (*argument repetition*). Accordingly, we measure the repetition in function arguments and present the results in Table 3.4. The second column shows the number of static functions called, and the third column shows the number of dynamic calls to these functions. The fourth column shows the percentage of all dynamic calls in which all the arguments were repeated argument values (*i.e.*, these functions had been called earlier with the exact same set of argument values), and the fifth column shows the percentage of dynamic calls in which no arguments were repeated (*i.e.*, none of the argument values in these function calls have occurred earlier as arguments). A strikingly large number of times the functions show *all-argument repetition*: for every benchmark, except *wave5*, more than 50% of dynamic function calls show all-argument repetition, with many of them — such as *go*, *ijpeg*, *viewperf*, and *mpeg-2* — having more than 75% of their dynamic functions with all-argument repetition. On the contrary, the functions seldom show *no-argument repetition.* Except for a few floating-point benchmarks (e.g., *wave5, apsi, fpppp,* and *su2cor*), for most benchmarks the percentage for *no-argument repetition* is less than 2%.

Do the above results suggest that large numbers of function calls are redundant? Not necessarily since not all of the computation in a function depends solely on its arguments. We will revisit this issue in Section 3.7 when we investigate some of the results of this chapter further. Nonetheless, the repeatability of all or some of the arguments of functions suggests an important source of repetition in instruction execution. (The percentage of calls with some arguments repeated can be calculated from the data in Table 3.4. We have also seen that argument repetition is not limited to single argument functions.)

Benchs	No. of funcs	No. of dynamic calls	Dynamic calls with ALL args repeated	Dynamic calls with NO args repeated
SpecInt '95				
go	481	11M	78%	0.49%
m88ksim	390	17M	83%	0.03%
ijpeg	528	1.5M	98%	0.01%
perl	477	6.4M	76%	1.36%
vortex	1,077	21M	67%	0.07%
li	473	30M	97%	0.31%
gcc	2,027	5.6M	59%	9.00%
compress	131	28M	66%	1.14%
SpecFP '95				
tomcatv	218	0.43M	55%	0.01%
swim	229	10M	76%	0.00%
su2cor	259	7.0M	86%	10.9%
hydro2d	267	307	56%	6.51%
mgrid	232	21,264	88%	1.92%
applu	236	8,690	67%	4.93%
turb3d	249	3.5M	74%	0.00%
apsi	326	5.0M	57%	21.6%
fpppp	268	0.29M	65%	19.2%
wave5	329	11M	16%	60.0%
Graphics				
viewperf	3,282	12M	89%	0.37%
mpeg-2	217	17M	94%	1.36%
pov-ray	2,228	17M	66%	1.81%

Table 3.4 Function Level Analysis. For each benchmark we show the number of functions, number of function calls encountered during execution, the percentage of function calls with *all-argument repetition*, and the percentage of function calls with *no-argument repetition.*

3.6.3 Local Analysis

To further our understanding of instruction repetition, we continue our analysis within each function — we call this *local* analysis. We divide dynamic instructions into different categories using two broad classification criteria: (i) the source of input data used by instructions and (ii) the specific task performed by groups of instructions.

In general, the data used within a function come from one of the following sources: (i) *arguments*, (ii) *global* data, (iii) *returned values*, and (iv) *function internals*. *Arguments* are the values explicitly passed to functions at the time of their invocation. *Global* data are the values which are global to the program (they either reside in the data segment or on the heap) and were not passed as arguments. *Returned values* are the values explicitly returned from other function calls. *Function internals*, like program internals in our global analysis, operate on immediate constants. Thus, using the first criterion for division, we will classify a slice of computation, for example, as *arguments* if it originates by operating on function arguments.

We identify the following categories for instructions based on the task performed: (i) *prologue*, (ii) *epilogue*, (iii) *global address calculation*, (iv) function *returns*, and (v) operations on stack pointer (*SP*). *Prologue* and *epilogue* represent the overhead incurred for calling a function. They perform, respectively, save and restore of callee-saved registers on entry and exit to functions. Just as addressing and loop control are "overhead" for a generic and compact representation of a computation, function prologue and epilogue are overheads associated with a modular programming style. G*lobal address calculation* is comprised of sequences of instructions which calculate the address of a global variable either using immediate values or using global pointer register, *gp* (a special register provided in MIPS architecture that points to the data segment). Since these instructions perform a very specific task, we group them separately from *function internals* (even though they operate on immediate values). *Returns* is comprised of function returns. The category SP consists of operations on stack pointer (e.g., adding an offset to stack pointer to form an address of a variable on the stack). We keep *returns* and *SP* separate from the other categories because their repeatability depends (partly) upon the present depth of the stack, and we wish to analyze the repeatability due to this influence separately.

We realize that the two broad classification criteria that we have chosen are not completely disjoint and also that the categories within them may not be the best possible way of dividing a function, but we believe that this division is a good first step in understanding the causes of instruction repetition.

As in global analysis, we categorize the instructions dynamically while executing the program on our simulator. We tag the data values with their appropriate source category, e.g., data loaded from the data segment are tagged as *global*, and we use function calling conventions to identify arguments and return values. The category in which an instruction is binned depends upon the categories of its input data. As in global analysis, an instruction with inputs from two different categories is categorized using the supersede rule *argument* $>_s$ *return value* $>_s$ *(global, heap)* $>_s$ *function internal.* The reason is to give preference to categories that may show more variability and less repeatability. Identifying the task-based categories such as *global address calculation*, *function returns*, and operations on *SP*, is straightforward. The *prologue* and *epilogue* are identified as follows. On entry into a function, we mark all registers as *unint* (except those used for passing the arguments). Store instructions that save *unint* registers are categorized as *prologue* whereas load instructions that load these saved values are categorized as *epilogue*. Instructions that allocate or deallocate space on the stack are also categorized, accordingly, as *prologue* or *epilogue*.

3.6.3.1 Overall Results

We show the percentage of total dynamic instructions within each category (*overall* analysis) in Tables 3.5, 3.8, and 3.11 (overall) for integer, floating-point and graphics benchmarks, respectively. The results vary from benchmark group to benchmark group (and also from program to program). However, in general, we glean the following from these results. Prologue and epilogue constitute a significant fraction of the dynamic program for integer and graphics benchmarks (e.g., as many as 24% of the dynamic instructions in *vortex*, 19% in *li*, 15% in *pov-ray*, and 12% in *viewperf*). But, they are not very prominent for the floating-point benchmarks where most benchmarks have less than 5% of dynamic instructions in these categories.

Although in global analysis we saw that most of the instructions for integer programs fell on slices originating from immediate values (*program internals*), in local analysis (Table 3.5) we see relatively fewer instructions derive their input values from immediate values (*function internals* and *global address calculations*). This is because several *program internal* slices span across functions and the information that they are internal slices (and that they might possibly be operating upon a compile time constant) gets hidden when these *program internal slices* cross function boundaries. These slices then show up as part of *global*, *heap*, or *argument slices*. This observation is also true for the graphics programs (Table 3.11). But, for the floating-point programs, we see that the *program internals* category in global analysis and the sum of the *function internals* and *global address calculation* categories in local analysis (Table 3.8) are comparable in value. This implies that in floating-point programs (unlike in

SpecInt '95

Categories	go	m88k	ijpeg	perl	vort	li	gcc	comp
prologue	3.12	4.93	1.16	7.33	12.40	12.08	8.48	1.90
epilogue	3.12	4.93	1.16	7.32	12.40	12.06	8.47	1.90
function internals	9.22	17.21	8.81	8.94	9.80	8.72	14.13	5.41
glb_addr_calc	15.66	14.77	0.44	4.47	3.35	0.54	2.95	10.27
return	1.12	1.75	0.16	1.13	2.10	3.04	1.29	2.79
SP	1.30	0.17	0.63	1.06	4.07	3.01	2.37	0.00
return values	1.77	4.46	4.29	2.54	2.90	3.69	2.88	16.72
arguments	12.46	15.65	34.30	22.53	36.62	8.69	19.33	5.02
global	52.24	26.94	2.05	9.20	5.41	11.72	16.77	55.98
heap	0.00	9.21	47.00	34.48	10.94	36.46	23.34	0.00

Table 3.5 Overall: Distribution of all dynamic instructions (% of all dynamic instructions).

Categories	go	m88k	ijpeg	perl	vort	li	gcc	comp
prologue	3.29	4.99	1.44	7.90	12.56	12.27	8.34	2.79
epilogue	3.29	4.99	1.44	7.88	12.56	12.24	8.34	2.79
functional internals	9.83	17.43	11.03	10.60	10.15	9.70	15.98	9.36
glb_addr_calc	16.69	14.96	0.55	5.30	3.46	0.59	3.46	17.79
return	1.19	1.77	0.20	1.34	2.18	3.38	1.51	4.83
SP	1.38	0.17	0.78	1.26	4.21	3.29	2.57	0.00
return values	1.83	4.51	4.60	1.10	2.86	3.78	2.54	8.50
arguments	12.28	15.62	30.99	22.21	35.84	8.77	17.48	4.18
global	50.22	26.24	2.52	8.43	5.42	12.77	17.74	49.75
heap	0.00	9.32	46.45	33.97	10.75	33.21	22.04	0.00

Table 3.6 Repeated: Distribution of all repeated instructions (% of all repeated dynamic instructions).

Categories	go	m88k	ijpeg	perl	vort	li	gcc	comp
prologue	99.03	99.99	98.49	90.66	97.81	91.35	39.71	84.55
epilogue	99.03	99.99	98.49	90.63	97.82	91.28	39.71	84.55
function internals	99.98	100.00	99.97	99.73	99.95	100.0	45.65	100.00
glb_addr_calc	99.98	100.00	99.98	99.99	99.99	98.26	47.43	100.00
return	99.99	100.00	99.97	99.99	99.99	99.99	47.21	100.00
SP	99.69	100.00	99.88	99.94	99.89	98.30	43.78	74.45
return values	96.95	99.99	85.69	36.39	94.97	92.13	35.62	29.38
arguments	92.43	98.59	72.12	82.94	94.53	90.76	36.50	48.08
global	90.15	96.22	98.56	77.09	96.64	97.99	42.69	51.30
heap	0.00	99.97	78.90	82.89	94.95	81.92	38.09	0.00

Table 3.7 Propensity: Percent of instructions in each category that get repeated (% of all dynamic instructions in each category).

SpecFP '95

Categories	tomcatv	swim	su2cor	hydro2d	mgrid	applu	turb3d	apsi	fpppp	wave5
prologue	0.16	3.10	1.84	0.00	0.02	0.01	1.04	2.60	0.23	2.51
epilogue	0.16	3.10	1.84	0.00	0.02	0.01	1.04	2.60	0.23	2.51
function internals	34.39	8.51	8.48	17.08	1.17	14.07	6.69	5.22	0.75	4.74
glb_addr_calc	0.06	22.70	1.62	13.81	0.01	7.51	0.45	0.58	6.14	5.47
return	0.04	1.19	0.71	0.00	0.01	0.00	0.36	0.50	0.06	1.07
SP	26.37	0.00	4.80	0.36	0.02	2.55	0.30	0.46	0.41	0.09
return values	37.03	4.23	12.34	0.00	0.03	0.01	18.34	2.03	6.89	5.80
arguments	0.55	9.50	42.63	4.25	32.04	5.57	58.53	51.69	1.78	10.13
global	1.13	47.66	25.76	64.49	66.70	70.29	13.27	34.31	83.51	67.66
heap	0.11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 3.8 Overall: Distribution of all dynamic instructions (% of all dynamic instructions).

Categories	tomcatv	swim	su2cor	hydro2d	mgrid	applu	turb3d	apsi	fpppp	wave5
prologue	0.28	5.92	3.27	0.00	0.09	0.01	1.15	3.62	0.47	2.55
epilogue	0.28	5.92	3.27	0.00	0.09	0.01	1.15	3.62	0.47	2.55
function internals	54.81	13.02	16.33	17.43	6.06	23.02	7.43	7.83	2.01	7.20
glb_addr_calc	0.11	18.70	3.27	4.38	0.04	12.19	0.50	0.88	16.78	8.61
return	0.08	2.42	1.43	0.00	0.03	0.00	0.40	0.76	0.16	1.76
SP	32.22	0.00	9.44	0.83	0.08	4.18	0.34	0.70	1.11	0.14
return values	9.16	8.09	5.32	0.00	0.14	0.00	20.19	1.83	4.55	3.15
arguments	0.96	19.29	28.47	0.38	53.84	4.67	57.18	59.24	3.63	9.85
global	2.00	26.64	29.21	76.99	39.64	55.91	11.66	21.54	70.82	64.20
heap	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 3.9 Repeated: Distribution of all repeated instructions (% of all repeated dynamic instructions)

Categories	tomcatv	swim	su2cor	hydro2d	mgrid	applu	turb3d	apsi	fpppp	wave5
prologue	98.60	94.14	87.86	77.22	95.59	91.03	99.91	91.68	73.97	61.82
epilogue	98.60	94.13	87.86	77.06	95.64	91.11	99.91	91.68	73.95	61.82
function internals	89.50	75.29	95.28	44.76	99.90	99.60	99.99	98.74	98.10	92.64
glb_addr_calc	99.99	40.57	99.99	13.92	96.40	98.75	99.99	99.99	99.93	95.99
return	99.99	100.00	100.00	69.61	98.45	95.98	100.00	99.99	99.90	100.00
SP	68.62	75.63	97.25	99.94	97.73	99.98	99.96	99.96	99.12	99.82
return values	13.89	94.09	21.31	49.88	93.67	35.13	99.04	59.17	24.14	33.18
arguments	96.87	99.95	33.03	3.87	32.42	51.08	87.90	75.52	74.35	59.34
global	99.99	27.52	56.09	52.35	11.47	48.41	79.07	41.36	31.01	57.91
heap	49.82	40.26	0.00	0.00	64.31	65.85	0.00	62.40	41.30	33.02

Table 3.10 Propensity: Percent of instructions in each category that get repeated (% of all dynamic instructions in each category).

Graphics

Overall

Categories	viewperf	mpeg2	povray
prologue	5.95	4.78	7.28
epilogue	5.95	4.78	7.28
function internals	8.37	5.69	6.44
glb_addr_calc	0.51	2.26	1.72
return	1.21	1.70	1.60
SP	1.08	0.09	1.63
return values	6.15	1.65	7.38
arguments	26.17	19.13	32.09
global	4.48	42.75	11.98
heap	40.11	17.17	22.60

Repetition

Categories	viewperf	mpeg2	povray
prologue	7.09	6.24	8.84
epilogue	7.09	6.24	8.84
function internals	10.82	7.78	7.78
glb_addr_calc	0.66	3.09	2.01
return	1.56	2.33	1.97
SP	1.40	0.12	1.99
return values	5.67	2.00	5.58
arguments	21.91	14.31	27.38
global	5.79	46.33	11.38
heap	38.00	11.55	24.23

Propensity

Categories	viewperf	mpeg2	povray
prologue	92.14	95.43	98.07
epilogue	92.14	95.43	98.10
function internals	100.00	100.00	97.62
glb_addr_calc	100.00	100.00	94.45
return	100.00	100.00	99.55
SP	100.00	99.98	98.12
return values	71.22	88.58	61.09
arguments	64.76	54.69	68.95
global	100.00	79.23	76.72
heap	73.29	49.20	86.64

Table 3.11 Local Analysis results for Graphics benchmarks. "Overall" shows the numbers are % of all dynamic instructions. "Repeated" shows Contribution of each category to total repetition (the numbers are % of all repeated dynamic instructions). "Propensity" shows the propensity of each category to repetition (the numbers are % of all dynamic instructions in that category).

integer and graphics programs) most slices which derive inputs from the immediate values do not cross function boundaries (and, hence, may be recognizable and exploitable statically).

In general, most of the dynamic instructions fall on *global*, *heap* or *argument* slices. A significant portion of the dynamic program is devoted to calculating the addresses of global variables, e.g., 16% for *go*, 15% for *m88ksim*, 23% for *swim*, and 14% for *hydro2d*. Categories *SP* and *returns* constitute few dynamic instructions (less than 2% in most cases). *Return value* slices also comprise few dynamic instructions (less than 5%) for all integer benchmarks, except for *compress* where they comprise 17% of dynamic instructions. This category is more prominent in floating-point and graphics benchmarks where it comprises of 37% dynamic instructions in *tomcatv*, 18% in *turb3d*, 12% in *su2cor*, and 7% in *pov-ray*.

3.6.3.2 Repetition Breakdown

We show the percentage of total repeated instructions for each category in Tables 3.6, 3.9 and 3.11 (repeated) for integer, floating-point, and graphics benchmarks, respectively. The amount of repetition that each category accounts for varies with the benchmark. But, in general, most of the repetition is accounted for by *arguments*, *global* (or *heap*), and *function internals*. *Prologue* and *epilogue* also contribute significantly to repetition, for integer and graphics benchmarks.

We show the *propensity* of each category to repetition — *i.e.*, the percentage of dynamic instructions in each category that got repeated — in Tables 3.7, 3.10, and 3.11 for integer, floating-point and graphics benchmarks, respectively. In general, we see that every category is amenable to repetition, especially for integer and graphics benchmarks (greater than 90% propensity for most cases). The propensity is especially high (as would be expected) for *function internals* and *global address calculations*. The percentages are high for *return* and *SP* as well, but we note that these categories have very few instructions (compared to the other categories). The propensities are lower (less than 50% in many cases) for *return values*, *global* (or *heap*), and *argument* categories.

Next we discuss the results and describe why each category may be getting repeated. (All percentage values presented below are from Tables 3.6, 3.9, and 3.11, unless specified otherwise.)

Global and Heap Values: For most benchmarks, between 20% and 70% of repeated instructions fall on slices originating from load instructions that read global values. This repetition can occur for several reasons. The runtime switches (which are mostly set using parameters that are input to a program) are often stored in global variables. These switches get initialized when the program begins execution and remain constant for the remainder of the execution. Often other program parameters, which remain constant for a given execution, are stored in global data structures (e.g., a table of frequencies for all letters used in Huffman encoding, or machine descriptions like function unit latency in a processor simulator). These data structures get initialized once per program execution (either at compile time or runtime) and remain unchanged thereafter. For some global variables, e.g., positions on a chess or a *go* board, the values may change infrequently or the variables may assume only a small set of values, causing the same values to flow down to the dependent instructions and hence resulting in repetition.

Function Prologue and Epilogue: These two categories comprise a significant percentage of total repetition for integer and graphics benchmarks (e.g., 15% for *perl*, 24% for *vortex*, 13% for *gcc,* 17% for *pov-ray*, and 14% for *viewperf*). This repetition occurs because functions often save and restore the same values of callee-saved registers from the same stack locations. Such a situation may happen, for example, when functions get called from the same call site repeatedly (hence the save and restore code accesses the same locations in the stack) and the values of callee-saved registers are the same as before (because, for instance, if they are not used in the caller function at all).

Function Arguments: A significant percentage of repeated instructions fall on argument slices (e.g., 26% for *ijpeg*, 22% for *vortex*, 59% for *apsi*, 57% for *turb3d*, and 27% for *pov-*

ray). As shown earlier in Table 3.4, many times functions are called repeatedly with some or all of their arguments having the same values as before. In such cases, the instructions which operate on these arguments may perform the same computation repeatedly. However, repetition of an argument doesn't necessarily result in repetition of its complete slice; the values coming from other slices (e.g., global slices) that merge with argument slices may change and hence prevent repetition. For example, in *ijpeg* only 77% of the instructions from this *function arguments* category (propensity results in Table 3.7) are repeated even though 98% of its dynamic functions are called with *all-argument repetition*[5] (the function-level analysis results are found in Table 3.4).

Function Return Value: Often, the value returned by function calls belongs to a small set of possible values (e.g., true or false). In such cases, the computation in the caller function which uses this return value may perform the same task repeatedly. Although repetition due to this category is generally low, it is significant in some cases: e.g., it is 9.3% for *compress*, 20% for *turb3d*, 9.2% for *tomcatv*, 5.7% for *viewperf*, and 5.6% for *pov-ray*.

Function Internals: Since these slices originate from instructions operating on immediate values, the different execution of these slices generate the same results (provided the governing control flow resolves in the same way for each execution). The percentage contribution to repetition for some of the benchmarks are 19% for *vortex*, 19% for *gcc*, 54% for *tomcatv*, 16% for *su2cor*, and 11% for *viewperf*.

Global Address Calculation: Instructions in this category either operate on immediate values or on register *gp* (which is a runtime constant). Hence they perform the same task every

5. In *ijpeg*, several functions are called with pointers to global arrays as arguments. Although the pointer values remain the same, the contents of the array change.

time they are executed. The percentage contribution of this category to repetition for some of the benchmarks are 18% for *go*, 15% for *m88ksim*, 19% for *swim*, and 12% for *applu*.

SP and Returns: The computation involving SP, such as adding an immediate to form an address of a variable, generates the same result if the value in SP is the same, which is the case when the same function is called from the same call depth repeatedly (e.g., a function called from the same call site repeatedly). The percentage contribution of SP to repetition is in general low (less than 5%) for most benchmarks, except for *tomcatv* (32%) and *su2cor* (9%). *Returns* get repeated when a function returns to the same call site repeatedly. The percentage contribution of *returns* to repetition is also low in general, except for a few integer benchmarks where the contributions are measurably high (e.g., 4.9% for *compress and* 3.3% for *li*). In most cases, the contribution of returns to repetition is less than 2%.

We make another observation from the repetition results similar to the observation made for the overall results. Although the results from global analysis for integer and graphics benchmarks show that most of the repeated instructions are part of *program internal* slices (Table 3.3, under *repetition*), in local analysis we see comparatively fewer repetitions fall on slices that originate from immediate values — *i.e.*, *function internal* and *global address calculation* slices. This indicates that much of the invariance flows into a function via arguments and global values and that this invariance may not be obvious (statically) inside the function. However, in the case of floating-point benchmarks the global and local results are comparable in this respect, which implies that such invariances may be easy to track statically for these benchmarks.

3.7 Discussion and Further Investigations

In the last few sections, several characteristics of sources of instruction repetition have surfaced. Many of these characteristics are striking. For instance, most of the repetition in programs fall on instruction slices originating from the hardwired values in programs, most of the dynamic functions exhibit all-argument repetition, and significant repetition is generated from function prologues and epilogues. These characteristics raise several questions. Why does such a behavior exists in programs? Why wasn't repetition eliminated at compile time? Can repetition be easily tracked for the purposes of exploitation (in software or hardware)? We don't yet know the conclusive answers to these questions, which can be both yes and no; the answers can only be found after much further research, which is beyond the scope of this thesis. However, attempting to address these questions, even partially, is still important as it will likely help develop a better grasp on the nature of the phenomenon (and, possibly, on appropriate methods for its exploitation). Consequently, in this section, we discuss and investigate further several of the results discovered in previous analyses to address some of the above questions. This section consists of three subsections: one devoted to discussing results from each of the three analyses — global, function-level, and local. We begin with the global analysis.

3.7.1 Global Analysis

Global analysis (Table 3.3) shows that most of the dynamic instructions and the repetition fall on the *program internal* slices and *global initialized* slices. These slices originate from immediate values and statically initialized data respectively, both of which are known at compile time. This information suggests that we may be able to optimize code statically to eliminate this source repetition. However, there may be some challenges in doing so, which we discuss below:

- The dynamic path through the program may not be known statically. Although the same definition of a value may reach a use repeatedly, this invariance may not be obvious at compile time.
- To ensure correctness, a compiler needs to assume dependences conservatively. On several occasions, global variables cannot be register allocated in the presence of pointers or function calls. Dynamically, loads of global variables may read the same value repeatedly.
- It may not be obvious within the body of a function, without sophisticated inter-procedural analysis, that a value is statically known if the value was passed to the function as an argument.
- Much instruction repetition is a result of code executed to dynamically recreate a computation from its static image. Exploiting this repetition statically may involve “unrolling” the dynamic computation statically, perhaps affecting the generality of the computation as well as the code size.
- Some instruction repetition is due to features of the instruction set and cannot be eliminated by optimizations such as constant propagation. For example, the number of bits in the immediate field of an instruction format limits the size of the immediate value that can be handled by an instruction. In such cases, larger constants are manipulated using a sequence of instructions, all of which would perform the same computation when executed repeatedly.
- In some situations, a loop invariant computation may not be register allocated, because of

the resultant increase in register pressure which might cause spills inside the loop.

3.7.2 Function Level Analysis

Function analysis (Table 3.4) shows that most of the functions are called with repeated arguments (*all-argument repetition*). This result immediately brings to mind a question: how many of these functions can be memoized (dynamically or statically)? We investigate this result further.

Memoization can be hindered if a function has side effects such as external input/output or stores to a global address or if it has implicit inputs through global variables. In Table 3.12, we show the percent of functions called with *all-argument* repetition that do not have any side effects or implicit inputs (hence may be candidates for memoization). We see different results for different benchmark groups. In integer benchmarks, almost every function has side effects or implicit inputs and may defy memoization (unless the side effects and other inputs themselves have a repeated pattern that can be detected statically). On the other hand, in most floating-point benchmarks and in two graphics benchmarks (*viewperf*, and *pov-ray*), a significant percentage of functions that are called with all-argument repetition lack both side-effects (external i/o or store to a global variable) and implicit inputs (e.g., load values from a global variable). These functions may be good candidates for memoization.

Another aspect of the repeatability of function arguments is the number of different values with which this repetition takes place. This aspect is important because it determines the tractability of this program behavior in hardware or software. To get an idea of this aspect, we determine for each function the most frequently occurring argument-set, the second most fre-

Benchmarks	Dynamic Functions w/o side effects or implicit inputs	
	% of all funcs	**% of funcs with all-arg repetition**
SpecInt '95		
go	0.0%	0.0%
m88ksim	7.8%	9.3%
ijpeg	0.3%	0.2%
perl	0.0%	0.0%
vortex	0.0%	0.0%
li	0.1%	0.0%
gcc	0.6%	0.9%
compress	0.0%	0.0%
SpecFP '95		
tomcatv	32.7	55.3
swim	79.2	86.4
su2cor	85.5	87.5
hydro2d	0.0	0.0
mgrid	1.0	0.0
applu	2.2	2.1
turb3d	37.1	42.4
apsi	56.7	51.6
fpppp	42.1	44.4
wave5	54.0	78.2
Graphics		
viewperf	22.4	25.2
mpeg-2	0.0	0.0
pov-ray	23.3	32.2

Table 3.12 Functions which do not have any side effects or any implicit input. The numbers are percentages of all dynamic functions (column 2) and percentages of functions with *all-argument repetition* (column 3).

quently occurring argument-set, and so on. Then we determine what percentage of the total all-argument repetition (shown in column 4 of Table 3.4) is because of the most occurring argument-set, the second most occurring argument-set, and so on. We show these numbers in Figure 3.6 for up to five most frequently occurring sets of arguments. These results show, for example, if we track for every function in *viewperf* the most frequently occurring argument-set (using, say, a direct-mapped hardware table) then we would capture 25% of the function calls with all-argument repetition, 37% if we track the top two frequently occurring argument-sets (using, say, a 2-way associative hardware table). We see that for integer benchmarks the coverage attained by the five most frequently occurring combination of argument values is not very high: in all but one case, even tracking all five most frequent sets of argument values does not allow us to cover more than 50% of functions with all-argument repetition. Whereas, for floating-point benchmarks, the coverage is considerably higher: the most frequently occurring set of argument values itself covers most of the all-argument repetition in many cases (e.g., in *tomcatv*, *apsi*, *applu*, and *wave5*). In graphics benchmarks, although the coverage is higher than in most integer benchmarks, there is significant all-argument repetition beyond the top five argument sets.

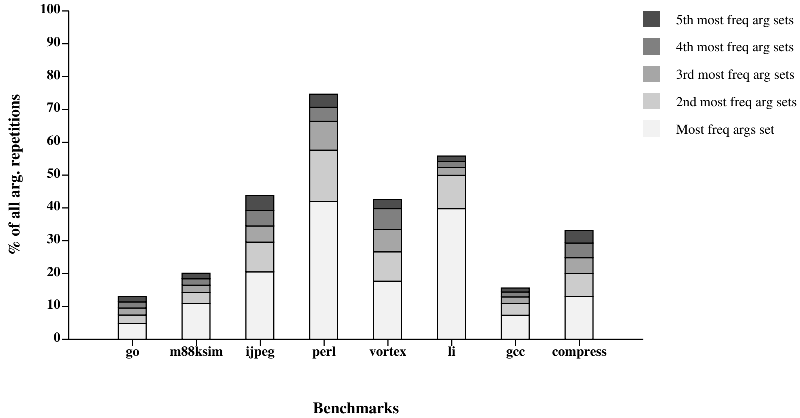
3.7.3 Local Analysis

Local analysis (Tables 3.5-3.11) shows that function prologue and epilogue are a significant contributor to both the number of instructions executed dynamically as well as to instruction repetition (especially in integer and graphics benchmarks). This overhead and repetition can potentially be optimized if the compiler had global information and could inline the function at the call site. From this point of view, it is interesting to find out how many static functions contribute to most of the prologue+epilogue repetition and to determine their sizes (since increase in code size is one of the issues in function inlining). In Table 3.13, we show the sizes (in number of static instructions) of the functions that are the top five contributors to the prologue+epilogue, as well as the fraction of all prologue and epilogue repetition accounted for by these five functions (*coverage* column) for the benchmarks. We observe that in all three benchmark groups most functions are greater than 50 instructions in size and may be considered large for inlining purposes.[6] Among the three benchmark sets, the floating point benchmark set has the greatest number of small functions (< 50 instructions in size) as major contributors to prologue+epilogue repetition (hence, these benchmarks may lend themselves well to function inlining). From the percent coverage values, we can also deduce that a considerable amount, greater than 40%, of prologue+epilogue repetition remains for many benchmarks even after considering the top five functions. Thus, simply focusing on a few big contributors may not eliminate most of the prologue+epilogue repetition.

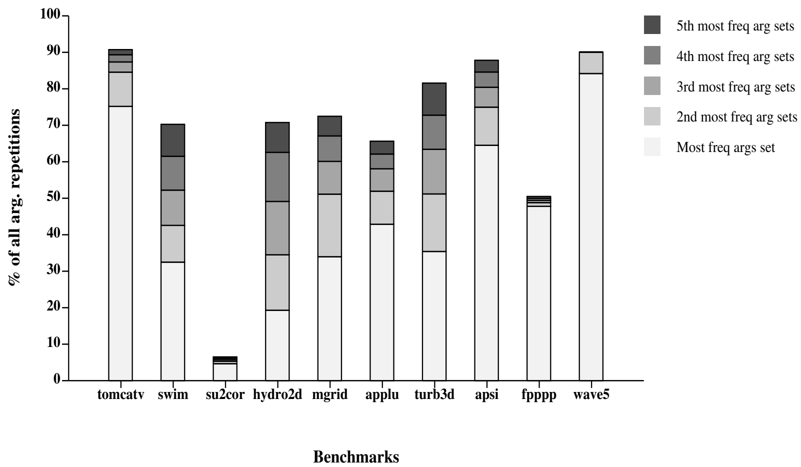
Local analysis also identifies other sources of instruction repetition, such as global slices, function internal slices, and instructions that compute global addresses. Another important aspect of the global slice repetition is the number of different values with which the repetition takes place. (Since function internal slices and slices that compute global addresses begin from constant values, they get repeated with a single value every time they are executed). As in the case of function argument (discussed in Section 3.7.2), the number of different values for global slices determines its tractability (whether in hardware or software) with fewer val-

6. Because the dynamic path length through a function can be smaller than the static instruction count, the prologue/epilogue can still be a significant contributor to the dynamic instruction count even for large functions.

SpecInt '95



SpecFP '95



Graphics

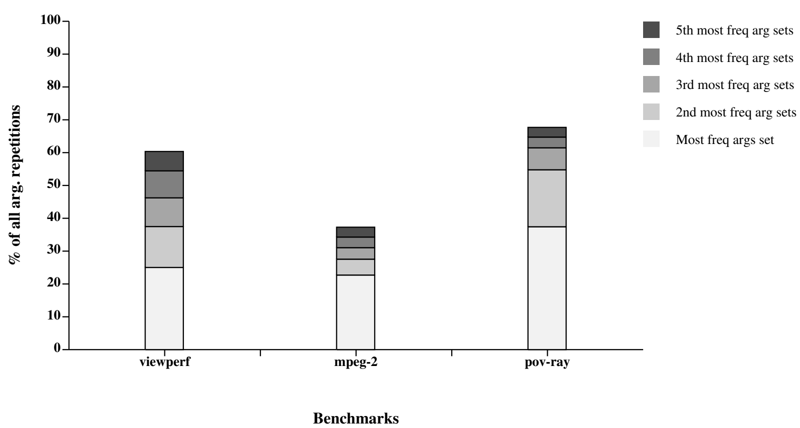


Figure 3.6 Percentage of all-argument repetition due to the five most frequently occurring argument set.

Bench.	1	2	3	4	5	coverage
			SpecInt '95			
go	addlist **113**	getefflibs **558**	lupdate **683**	ldndate **683**	livesordies **799**	40%
m88ksim	Data_path **143**	execute **883**	display_trace **150**	Pc **149**	test_issue **56**	66%
ijpeg	emit_bits **97**	encode_one_block **103**	fill_bit_buffer **93**	jpeg_idct_islow **643**	memcpy **55**	81%
perl	eval **6639**	memmove **97**	malloc **304**	str_nset **76**	str_sset **142**	59%
vortex	Mem_GetWord **53**	TmFetchCoreDb **125**	Chunk_ChkGetChunk **50**	Mem_GetAddr **49**	TmGetObject **49**	49%
li	mark **67**	xlobgetvalue **88**	xlsave **40**	livecar **46**	newnode **26**	62%
gcc	reg_scan_mark_refs **262**	mark_set_resources **309**	canon_reg **162**	mark_jump_label **259**	copy_rtx_if_shared **271**	17%
compress	getcode **86**	output **142**	readbytes **85**			100%
			SpecFP '95			
tomcatv	x_getc **47**	pow **118**	__finite **19**	ldexp **119**	pow_P **92**	42%
swim	__drem **176**	__finite **19**	sin **126**	cos **119**		100%
su2cor	log **158**	ldexp **119**	__finite **19**	__logb **48**	__drem **176**	95%
hydro2d	fct_ **824**	artdif_ **401**	filter_ **1394**	t2_ **141**	b2_ **189**	67%
mgrid	memcpy **55**	__mpn_divmod **360**	x_putc **35**	__mpn_mul_1 **18**	do_fio **219**	43%
applu	exact_ **156**	x_putc **35**	memcpy **55**	__mpn_divmod **360**	ne_d **412**	69%
turb3d	fftz2_ **257**	__drem **176**	fftz1_ **148**	cfft_ **363**	__finite **19**	88%
apsi	ekmlay_ **134**	pow **118**	__finite **19**	radb4_ **432**	ldexp **119**	54%
fpppp	ldexp **119**	fmtgen_ **414**	exp **91**	d_int **16**	floor **29**	74%
wave5	__drem **176**	__finite **19**	ldexp **119**	log **158**	cos **119**	86%
			Graphics			
viewperf	sample_2d_linear **728**	floor **29**	sample_lambda_2d **329**	gl_shade_rgba **486**	log **158**	44%
mpeg-2	putbyte **21**	Flush_Buffer **365**	Get_Bits **14**	form_component _prediction **679**	form_prediction **117**	91%
pov-ray	Ray_In_Bound **120**	Intersect_Box **528**	Point_In_Clip **56**	priority_queue_in sert **117**	All_Box_Intersecti ons **207**	28%

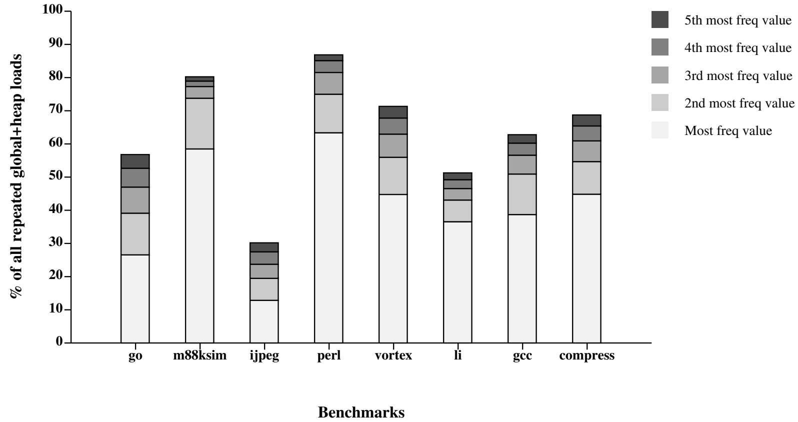
Table 3.13 We show names of five functions which are 5-top contributors to prologue+epilogue repetition. For each function we show its size in number of instructions. We also show the amount of prologue+epilogue repetition covered by these five functions.

ues meaning easily tractable and vice versa. To obtain this number of different values, we determine for each global load the most frequently occurring value, the second most frequently occurring value, and so on. (We consider global loads because they initiate the global slices). Then we determine what percentage of total global load repetition can be accounted for by the most frequently occurring values, the second most frequently occurring value, and so on. We show these results in Figure 3.7 for up to five most frequently occurring values. These results can be interpreted as follows: we will able to track about 40% of repeated load values that start the global slice if we track the most frequently load value (using a direct mapped hardware table, for instance) for all global loads in *gcc*, for example, and about 55% of repeated load values is we track top 2 most frequently occurring values for each load (using a 2-way associative hardware table, for instance). We see that in the floating-point benchmarks, the coverage attained by the most frequently seen load values is considerably higher than in the integer or graphics benchmarks — e.g., for *tomcatv*, *swim*, and *applu*, the most frequently seen value itself accounts for more that 80% of repetition on global slice. For integer and graphics benchmarks, global slices may need to be tracked for several possible values to capture more of the global repetition.

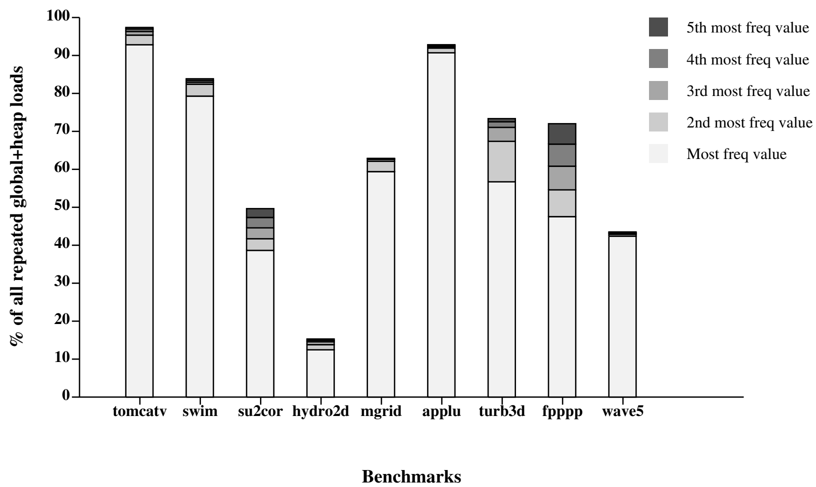
3.8 Summary and Conclusions

In this chapter, we empirically analyzed instruction repetition, which is the phenomenon that instructions operate on the same operand values and produce the same result repeatedly. We analyzed the SPEC ‘95 integer and floating-point benchmarks and graphics programs to understand the underlying characteristics of programs that give rise to this phenomenon.

SpecInt '95



SpecFP '95



Graphics

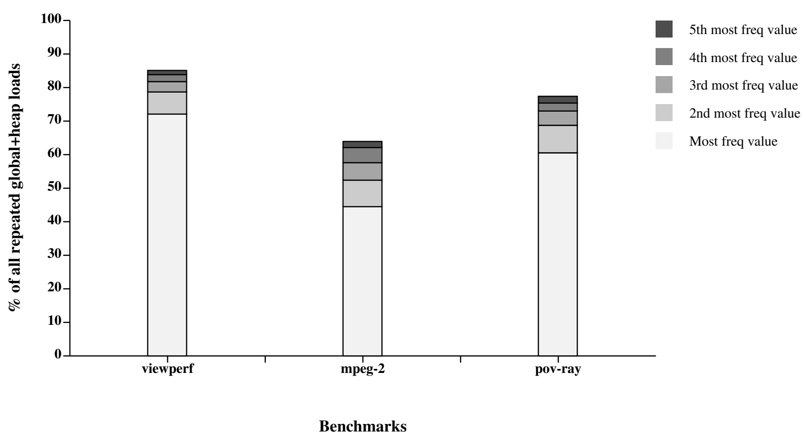


Figure 3.7 Percentage of all global+heap load repetition with the five most frequently repeated values.

We first characterized instruction repetition. We found that most of the dynamic instructions in programs are repeated (e.g., 99% for *m88ksim*, 93% for *vortex*, and 82% for *pov-ray*). We also found that although almost all of the executed static instructions contribute to repetition, for most benchmarks less than 20% of the repeated static instructions account for more than 90% of the dynamic repetition. However, instruction repetition is not limited to instructions producing a few instances dynamically; as much as 42% of the repetition in *ijpeg*, and 28% in *li* is due to instructions that produce between 101 to 1000 distinct values.

To better understand this phenomenon, we further analyzed the dynamic execution of these programs at three levels: (i) global, (ii) function, and (iii) local (inside functions). In global analysis, we tracked the data usage pattern of the program as a whole and determined the sources of repeated instructions (external input, global initialized data, or program internals). We saw that most of the instruction repetition fall on instruction slices originating from program internals values (such as immediate values) and global initialized data. We saw similar results when running the benchmarks with other inputs although we did not report these results in this chapter (they are reported in Appendix A). This suggests that repetition as a phenomenon is more a property of the way computation is expressed in a program than a property of input data (especially for the integer benchmarks).

In the function analysis, we saw that very functions often get invoked repeatedly with exactly the same set of arguments values (e.g., 98% of function call in *ijpeg*, 83% in *m88ksim*, 78% in *go*). In contrast, only a few function calls have no repeated arguments values (less than 2% for several benchmarks).

In the local analysis, we tracked the source of repetition. We classified the instructions of a function into different categories based on the source of data values used (e.g., function arguments) and the specific task performed (e.g., save and restore registers). We found that most of the repeated instructions fall either on global value or argument value slices. Instructions on function internals slices also get repeated frequently. Significant repetition is also seen due to function prologue and epilogue. For some benchmarks, the sequences of instructions that calculate the addresses of global variables also get repeated significantly.

Finally, we discussed and further investigated some of the results of the three analyses. We showed that most functions exhibiting all-argument repetition had side effects or other implicit inputs (hence may not be easy candidates for memoization). We presented number of different values with which the function arguments are repeated. For local analysis results, we identified functions that contribute the most to the function prologue and epilogue repetition and showed the number of different values with which load slices get repeated.

Chapter 4

Dynamic Instruction Reuse

In the previous chapter we discussed the phenomenon of instruction repetition. We presented its various characteristics and analyzed various programs to expose its various sources. In this chapter we are concerned with how to use this phenomenon to our advantage — *i.e.,* how to exploit it.

Ideally, on the matter of exploitation we would like to ask questions such as, “What are the different ways of exploiting repetition?” or “How best can we exploit instruction repetition?” However, instruction repetition is a phenomenon that was discovered fairly recently and, hence, the depth of understanding required to answer such questions is not yet available. In addition, how we exploit this phenomenon will also depend on what we wish to gain from it: if the goal is to make the processor run fast, then there may be one way of utilizing this opportunity; but if the goal is something else — say, to reduce the power consumption in processor — then the approach can be entirely different (and may be still unknown). The different uses of this phenomenon will only become apparent after we understand what aspects of it are exploitable and what are not, and this understanding can only be developed by actually undergoing the process of exploiting this phenomenon.

Thus, in this chapter, instead of attempting to explore the topic of exploitation comprehensively, we pursue one particular purpose for exploiting this phenomenon — a purpose that is somewhat self-suggestive from the nature of the phenomenon itself: *i.e.*, reducing the amount of work that needs to be done to execute a program. If the instructions are producing the same results repeatedly, then why execute them over and over again? Instead, *reuse* results from their previous executions and avoid their repeated executions. The topic of our study in this chapter is *dynamic instruction reuse,* the name we call the technique employed for accomplishing the above result.

4.1 Instruction Reuse

Instruction reuse (IR) is a *non-speculative*, hardware technique that dynamically detects that certain instructions are producing the same results repeatedly and reuses the earlier results of these instructions instead of re-executing them. This technique works in the following manner. As instructions execute, their results are stored in a hardware table called the *Reuse Buffer* (RB). When an instruction enters the pipeline, the RB is queried to see if a *valid* result for that instruction is available; if so, that result is reused from the RB and the execution of the instruction is complete. A reused instruction "skips" the remaining pipe stages and becomes ready for retirement. However, if a valid result for the instruction is not found in the RB, the instruction moves through the pipeline and get executed as usual.

IR is non-speculative because only valid results, *i.e.*, results which are guaranteed to be correct, are reused. The validity of a result is confirmed by establishing that the operand values that were used to calculate that result are the same as the current values of the operands. Based on how we perform this validity check, which we term the *reuse test*, we obtain different schemes for reusing instructions (which is described later in this chapter).

There are several benefits of exploiting repetition in this manner. First, since a reused instruction is not required to be executed, it does not occupy resources (e.g., issue window entry, functional units, data cache ports) in a pipeline, making these resources available for other instructions to use. Second, through reuse, instruction results become known earlier in the pipeline than they are through execution. This permits dependent instructions to execute earlier. Third, reuse breaks the serialization due to data dependences, and hence, has the potential to exceed the dataflow limit. For example, reusing a chain of dependent instructions, in effect, completes the individual instructions together, which would not be possible if the chain is executed.

The layout for the rest chapter is as follows. In the next section, we describe the Reuse Buffer, the hardware table used in IR. Thereafter, in Section 4.3, we present the different schemes for IR; in Section 4.4, we describe how IR may integrate with a generic pipeline; in Section 4.6, we discuss certain implementation issues of IR; in Section 4.7, we perform a quantitative evaluation of IR; in Section 4.8, we discuss the various related work, and finally, in Section 4.9, we summarize and provide conclusions.

4.2 Reuse Buffer

All reuse schemes, which we will describe in the next section, employ a Reuse Buffer (RB) for storing and maintaining the previous results of instructions. A generic structure of an RB is shown in (Figure 4.1). Each entry of the RB stores information pertaining to a single instruction. The exact contents of the RB entries is decided by the particular scheme chosen to

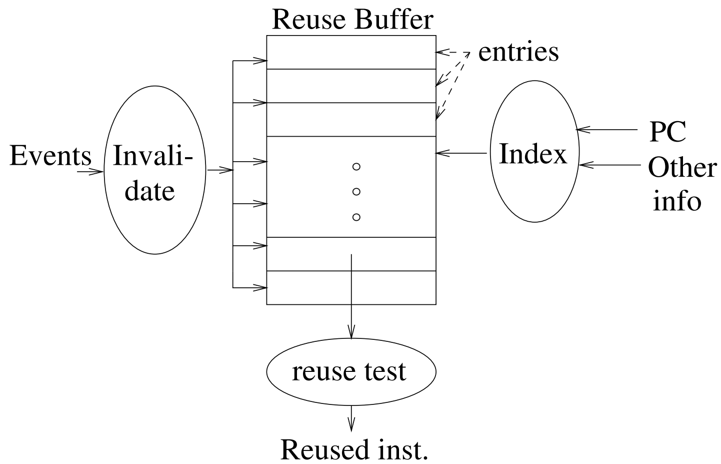


Figure 4.1 Generic Reuse Buffer. It is indexed by the PC (possibly combined with other information as discussed in Section 4.5.3), and it has mechanisms for selectively invalidating entries based on some event.

implement instruction reuse. Three more issues need to be dealt with: (i) how the information in the RB is accessed, (ii) how we know that the accessed RB entry (or entries) has reusable information, and (iii) how the buffer is managed.

The first issue is easily dealt with: the program counter (PC) of the instruction provides a convenient index for searching the RB. There are other ways of indexing in an RB, some of which we discuss in Section 4.5.3. One advantage of using the PC as an index is that it is available before any other information about the instruction; hence, RB access can begin early in the pipeline, enabling early reuse of instructions (or, permitting the use of a bigger, thus slower, RB). To accommodate multiple instances of a static instruction, the RB can be organized with any degree of associativity; the larger the associativity, the larger the number of dynamic instances of an instruction that can be held in the RB at a given time.

To deal with the second issue, we need to develop a *reuse test* which checks information accessed from the RB to see if there is a reusable result. Details of the test depend upon the reuse scheme, as we describe shortly.

There are two aspects to RB management: (i) deciding which instructions get placed in the buffer and (ii) maintaining the consistency of the buffer. The decision as to what to place in the buffer can range from a naive policy, *i.e.*, place all recently executed instructions in this buffer (if they aren't already present), to a more judicious policy that filters out instructions that aren't likely to be reused. This aspect of buffer management is dealt which in the next chapter; in this chapter we adopt the naive policy approach. Maintaining the consistency of information in the RB depends upon the reuse scheme, as we will see shortly.

4.3 Schemes for Instruction Reuse

In this section, we describe four hardware schemes to implement dynamic instruction reuse. These schemes mainly differ in the way in which reusable results are identified. The first scheme (S_v) tracks operand values for each instruction, the second scheme (S_n) tracks only operand names (register identifiers), and the third (S_{n+d}) and the fourth scheme (S_{v+d}) extend the first two schemes by the use of dependence relationships among the instructions for tracking reuse. For each scheme, we discuss the following issues:

- What information is stored in the RB?
- How is the reuse test performed?
- How is the information in the RB updated/invalidated?

4.3.1 Scheme S_v: Reuse based upon operand values

Scheme S_v is a straightforward implementation of the reuse concept. The operand values of an instruction are stored along with its result. Since the reuse test is based on operand val-

tag	operand1 value	operand2 value	address	result	mem valid

(a)

tag	operand1 reg name	operand2 reg name	address	result	result valid	mem valid

(b)

tag	operand1		operand2		address	result	result vaild	mem vaild
	src-index	reg name	src-index	reg name				

(c)

tag	operand1		operand2		address	result	mem vaild
	src-index	reg value	src-index	reg value			

(d)

Figure 4.2 RB entry (a) Scheme S_v (b) Scheme S_n (c) Scheme S_{n+d} (d) Scheme S_{v+d}.

ues, as we will see shortly, we call this scheme S_v, where 'v' stands for *value*.

When an instruction is decoded, its current operand values are compared with those stored in the RB. If they are the same, the result stored in the RB is reused. Loads, being a two-operation instruction, need special handling. Address-calculation can be reused if the operands for the address calculation did not change. However, the actual outcome of the load can only be reused if the addressed memory location was not written into by a store instruction. Information in the RB has to distinguish between the two. Likewise, stores are also special. While reusing the address calculation part of a store presents no problems (we treat it no differently from the address calculation for a load) we make no attempt to reuse the actual memory write — the memory write could have side effects outside the domain of the processing node. (Similar restrictions would apply to other instructions with side effects, e.g., loads in the I/O space.)

RB entry: An entry in the RB for this scheme is shown in Figure 4.2(a). The *tag* field stores part of the PC. The *result*, *operand value1,* and *operand value2* store the result and the operand values of the instruction. These fields are used to identify the instruction (or address calculation in the case of a load/store) that can be reused. The *memvalid* bit and the *address* field are used to determine if the actual memory access for a load instruction can be reused; the *memvalid* bit indicates whether the value loaded from memory (present in the *result* field) is valid, and the *address* field stores the memory address (*i.e.*, the outcome of the address calculation).

Reuse test: For testing reuse, the operands of an instruction are compared with the values in the *operand value* fields of the RB entry. A match indicates that *result* is valid (for non-load/store instructions) or *address* is valid (for loads and stores). For loads, in addition to testing the validity of the *address* bits, we also need to test the *memvalid* bit to see if the outcome of the load (in the *result* field) can be reused. If the operand values are not known at the time of the reuse test, then the instruction is not reused.

Invalidation: For non-load operations, the reuse test works because the operands uniquely determine the result; therefore invalidations are not needed to maintain the integrity of the test. For loads, a store to the same address invalidates the value in the *result* field. Accordingly, on a store, the *address* field of each RB entry is searched for a matching address, and the *memvalid* bit reset for matching entries.

4.3.2 Scheme S_n: Reuse based upon register names

In scheme S_n, we attempt to trivialize the reuse test (and also to reduce the size of each RB entry). Rather than store operand values, we store operand (architectural) register identifiers in the RB. When an instruction writes into a register, all instructions with a matching (source) register identifier in the RB are invalidated. Only the valid instructions are reused from the

RB. The advantage of this reuse test is that it can be done much earlier in the pipeline than the reuse test in scheme S_v since it does not require the operand values. Since the reuse test is based on operand names (and not value), we call this scheme S_n, where 'n' stands for *name*. The remaining details are as follows:

RB entry : An RB entry for this scheme is shown in Figure 4.2(b). Differences from scheme S_v are: (i) the *operand1* and *operand2* fields contain register names of the operands instead of actual operand values, (ii) there is a *resultvalid* bit, which indicates whether the result is valid. (This bit was not required in scheme S_v because the reuse test detected the stale results.) This bit is set when an entry is first inserted into the RB.

Reuse test: The reuse test is as simple as testing the state of *resultvalid* and *memvalid* bits. Address calculation for load/store instructions and results for all other instructions can be reused if the *resultvalid* bit is set; the result of a load instruction can be reused if both *resultvalid* and *memvalid* are set. (Since different instances of the same static instruction will have the same operand names,[1] we do not need to compare the operand names explicitly for reuse.) As mentioned above, since this reuse test does not require operand values, it can be potentially done earlier in the pipeline; this may result in the reuse being more beneficial.

Invalidations : As before, stores invalidate the loads from the same address (*memvalid* bit is reset). Moreover, when a register is written, the RB is searched for entries whose operand field matches the name of the register. The entries that match are marked invalid (*resultvalid* bit is reset).

1. This may not necessarily be the case for self-modifying code. Hence, for architectures that support self-modifying code, we will need to invalidate instructions in the RB when they get modified.

4.3.3 Scheme S_{n+d}: Reuse using register names and dependence chains

Scheme S_{n+d} extends scheme S_n by attempting to establish chains of dependent instructions, and to track the reuse status of such instruction chains. Since the reuse status of an instruction in the RB is established based on its operand names and/or its dependence information in this scheme, we call it scheme S_{n+d} (the letters 'n' and 'd' stand for *name* and *dependence* respectively).

Figure 4.3(a) motivates scheme S_{n+d}. The figure shows a dynamic stream of instructions on the left and the contents of the RB at different point in time on the right. I, J, K is a chain of dependent instructions; I_1, J_1, K_1 and I_2, J_2, K_2 are the dynamic instances of this instruction chain. With scheme S_n, only instruction I_2 could reuse the result of I_1 because results of J_1 and K_1 are invalidated by instruction R. Scheme S_{n+d} instead tries to establish the fact that instruction J_2 (J_1) depends solely upon instruction I_2 (I_1), and instruction K_2 (K_1) depends solely upon instructions I_2 and J_2 (I_1 and J_1) (Figure 4.3(b)). If instruction I_2 can be reused, so can instructions J_2 and K_2. Furthermore, if I_2, J_2, and K_2 are all fetched simultaneously from the RB, the reuse status of all three instructions could be established simply by establishing the

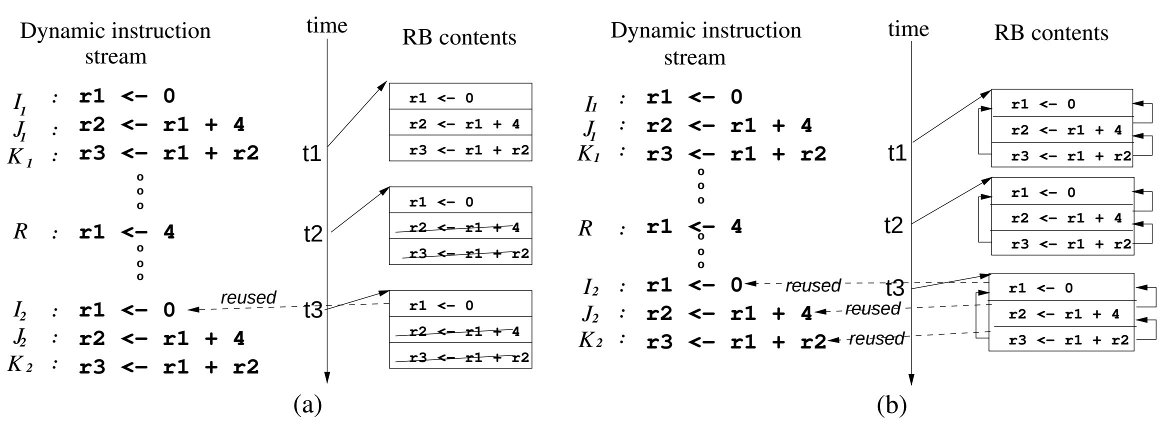


Figure 4.3 Dependent sequence of instructions (a) not handled in Scheme S_n, but (b) handled in Scheme S_{n+d}.

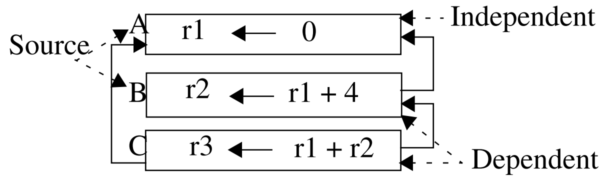


Figure 4.4 Instructions with data dependence links. The arrows point from the instruction using the value to the instruction producing the value.

reuse status of I_2, and verifying the dependence relationship (as we elaborate below). This is tantamount to obtaining the result(s) of chains of dependent operations in a single cycle. Scheme S_v, which does not maintain instruction dependence relationships, can’t establish the reuse status of a dependence chain as easily. In our example, the reuse status of I_2 would have to be established; the result of I_2 would be needed to establish the reuse status of J_2, and J_2’s result would be needed to establish the reuse status of K_2.

For the ensuing discussions, we define the following terms (illustrated in Figure 4.4). Instructions that produce values used by other instructions in the chain are called *source* instructions (e.g., A and B in the figure). Instructions whose source instructions are not in the chain, which implies that their data dependence information is not available, are called *independent* instructions (e.g., A). Finally, instructions whose source instructions are in the chain are called *dependent* instructions (e.g., B and C).

Dependence chains are created as entries are inserted into the RB. To facilitate this process, we use a mapping table called a *Register Source Table (RST)*. The RST has an entry for each architectural register; it tracks the RB entry which has (or will have) the latest result for that register. When an entry is reserved in the RB for an instruction, the RST entry for its destination register is updated to point to the reserved entry. If, however, an entry could not be reserved, then the RST entry for the destination register is set to invalid (since the latest producer of that register will not be in the RB). When an instruction is reused, the RST entry for its destination register is updated to point to the reused RB entry. The RST is similar in spirit to the rename map used in register renaming. In essence, the RST is used to link a consumer instruction to the latest producer instruction by pointing to the "physical register" (RB entry) of the producer. Accordingly, another way of looking at scheme S_{n+d} is to consider it as a "physical register" version of scheme S_n, which tracks dependences using architectural registers. We next present details of this scheme's operation and then illustrate it with an example.

RB entry: An RB entry (shown in Figure 4.2(c)) is similar to the one in scheme S_n, except for the addition of a *src-index* field. The dependence links are created by storing the RB index of the source instructions in this field. An invalid value is inserted in this field if the source doesn't exist in the RB.

Reuse test: The reuse status of independent instructions is established as it was in scheme S_n (*resultvalid* bit is set; *memvalid* is set in the case of load instructions). A dependent instruction is reused if its source instructions (in the RB), as indicated by the *src-index* field of its operands, are indeed the latest producers for its operands. This fact is established with the help of the RST, as we shall illustrate below with the help of an example (Figure 4.5).

State updates: As in schemes S_v and S_n, stores invalidate loads to the same address (*memvalid* is reset). As in scheme S_n, independent instructions are invalidated when their operands registers are overwritten (*resultvalid* is reset). Dependent instructions need not be invalidated on operand overwrites because their reuse status can be established using their dependence information. Instead, they are invalidated when their source instructions are evicted from the

RB, *i.e.*, when the dependence information is lost.[1] To perform this operation the RB needs to be searched for entries whose *src-index* field matches the index (in the RB) of the source instruction being evicted. The entries which result in a match are invalidated (*resultvalid* bit is reset).

Example: We illustrate the working of this scheme using the example shown in Figure 4.5 with the same dynamic stream of instructions as in Figure 4.3. Figure 4.5(a) shows the state of the RB and RST at the time when I_2 is encountered in the dynamic instruction stream. At this time, the results of instructions I_1, J_1 and K_1 are present in the RB with appropriate data dependence information (indicated by the links in the RB and the index values in the *src-index*

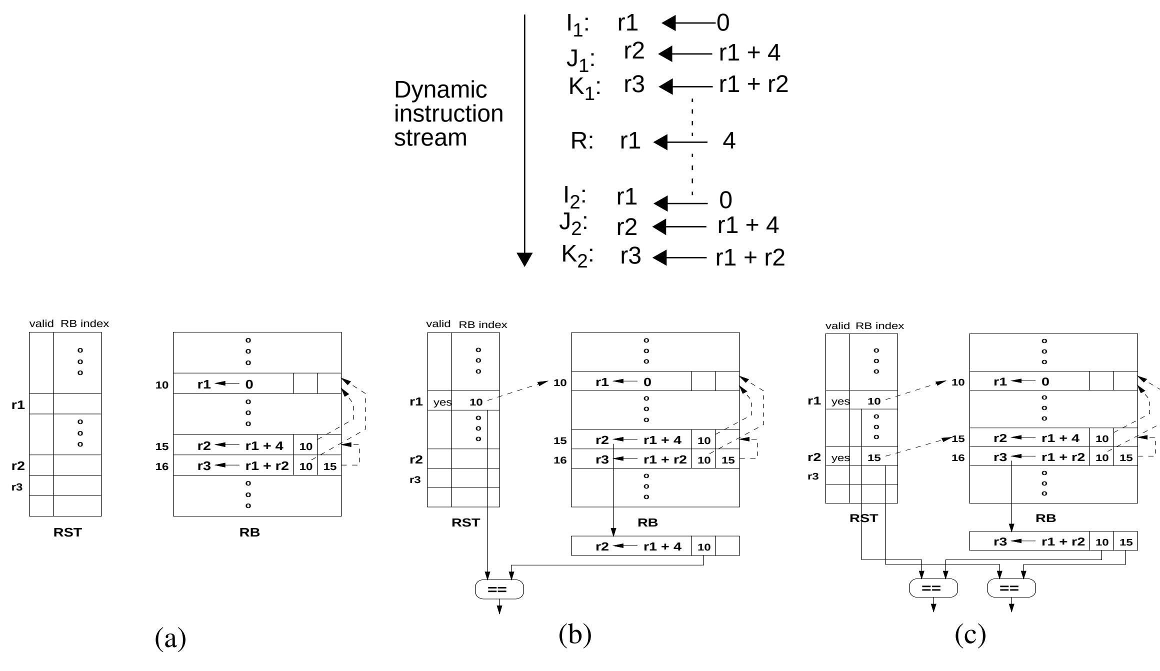


Figure 4.5 Illustrating the reuse test for dependent instructions. (a) State when I_2 is encountered. (b) Testing r2 <- r1 +4 for reusability. (c) Testing r3 <- r1 + r2 for reusability.

1. An optimization to this approach is to check whether the source instruction is the current producer for its destination register (this can be done using the RST). If so, then the dependent instructions are not invalidated; instead they are treated as independent instruction thereafter. In our simulations, we implemented this optimization.

field). Since instructions J_1 and K_1 are stored in the RB as dependent instructions, their results are not invalidated by instruction R (unlike scheme S_n). Instruction I_2 reuses the result of I_1 (since it is independent and valid), and the RST entry for *r1* is updated to point to the RB entry 10 (the latest producer for *r1*)(Figure 4.5 (b)). To establish the reusability of J_2, the *src-index* field for *r1* is compared with the RST entry for *r1* (Figure 4.5 (b)). A match indicates that the source for *r1* in the dependence chain (which is I_1) is also the current producer for *r1*; hence the result is reusable. Instruction K_2 gets reused in a similar fashion (Figure 4.5 (c)). The instructions I_2, J_2, and K_2 can be reused simultaneously if encountered in the same cycle. While performing the reuse test on each instruction, interdependence among them needs to be considered. The interdependence check resembles what is done while renaming registers for multiple dependent instructions in the same cycle.

4.3.4 Scheme S_{v+d}: Reuse using register values and dependence chains

Although the scheme S_v is the most accurate in detecting the reusable instructions among the three schemes presented so far, it is not very well suited for reusing chains of dependent instructions in a single cycle. For example, reusing two instructions, I and J, with J being dependent on I, would require that we first reuse I and then using the reused result of I we perform the reuse test for J. This whole operation may be difficult to do in a single cycle, especially for long dependence chains. To facilitate the reuse of dependent instructions, we augment the scheme S_v with the dependence-tracking ability of scheme S_{n+d}, giving us the scheme S_{v+d}.

As in scheme S_{n+d}, instructions in this scheme are stored in the RB with pointers to the RB entries containing their source instructions. The dependency chains are constructed using an

RST in the same way as they are constructed in scheme S_{n+d}. Most of the operations performed in this scheme are borrowed from scheme S_v or scheme S_{n+d}, as we describe below. In the following discussion, we use the terms *dependent* and *independent* instruction, which were define in Section 4.3.3. Next, we describe the various details of scheme S_{v+d}.

RB entry: An RB entry (shown in Figure 4.2(d)) is similar to the one in scheme S_v, except for the addition of a *src-index* field. Just like in scheme S_{n+d}, the dependence links are created by storing the RB index of the source instructions in this field. An invalid value is inserted in this field if the source doesn't exist in the RB.

Reuse test: The reuse status of independent instructions is established as in scheme S_v: the operand values are compared with the current values of those registers and the *memvalid* bit is used to determine the validity of loads. As in scheme S_{n+d}, a dependent instruction is reused by confirming that its source instructions (in the RB), as indicated by the *src-index* field of its operands, are indeed the latest producers for its operands. This fact is established with the help of the RST, as illustrated earlier in Figure 4.5.

State updates: As in other schemes, stores invalidate the loads to the same address (*memvalid* is reset). As in scheme S_{n+d}, the state of dependent instructions is updated when their source instructions are evicted from the RB, *i.e.*, when their dependence information is lost. The state can be updated in two ways: either (i) the dependent instructions can be marked invalid, or (ii) their *src-index* fields, pointing to the evicted source, are annulled (and thereafter, they are treated like independent instructions — *i.e.*, their validity is determined by value comparison). The first option is simple but conservative since it invalidates potentially useful instructions. The second option, on the other hand, retains the dependent instructions, but it requires additional space in RB entries since the operand values need to stored for the dependent instructions as well (so that value comparison can be performed if the dependent instructions become independent). Nevertheless, both update operations require that the RB be searched for the entries whose *src-index* field matches the RB index of the source instruction being evicted. These matching entries are either invalidated or converted into independent entries.

4.3.5 Summary of schemes

The four reuse schemes, which we presented in the preceding sections, mainly differed in the way the reuse tests were performed and the RB was kept consistent. Several mechanisms for implementing these two operations were presented. In this section, we summarize these mechanisms for each reuse scheme.

In Table 4.1, we show a check list of various mechanisms employed by each reuse scheme. We presented four mechanisms for performing reuse test. In scheme S_v, *value comparison* between the current operand values and those stored in the RB was used to reuse instruction. In scheme S_{v+d}, value comparison was used for reusing independent instructions. In scheme S_n, the *valid bit check,* which indicated validity of the operand values in the RB, was used to reuse instructions. This check was also used by scheme S_{n+d}, but only for reusing the independent instructions. The remaining two reuse test mechanisms were for reusing dependent instructions. In scheme S_v, the dependent instructions were reused by comparing the values in RB with those forwarded from the source instructions (*Value forwarding*). In schemes S_{n+d} and S_{v+d}, the dependent instructions were reuse by tracking and checking the

Mechanisms			S_v	S_n	S_{n+d}	S_{v+d}
Reuse Test	Value comparison		✔			✔
Reuse Test	Valid bit check			✔	✔	
Reuse Test	Dependent instruction reuse	Value forwarding	✔			
Reuse Test	Dependent instruction reuse	Dependence check			✔	✔
Maintaining consistency in RB		Load invalidation	✔	✔	✔	✔
Maintaining consistency in RB		Non-load invalidation		✔	✔	
Maintaining consistency in RB		Dependent instruction invalidation			✔	✔

Table 4.1 Various mechanisms for implementing different reuse schemes.

dependence between instructions (*Dependence check*). As illustrated earlier in Figure 4.3, scheme S_n cannot not reuse dependent instructions (hence we had no corresponding mechanism for it).

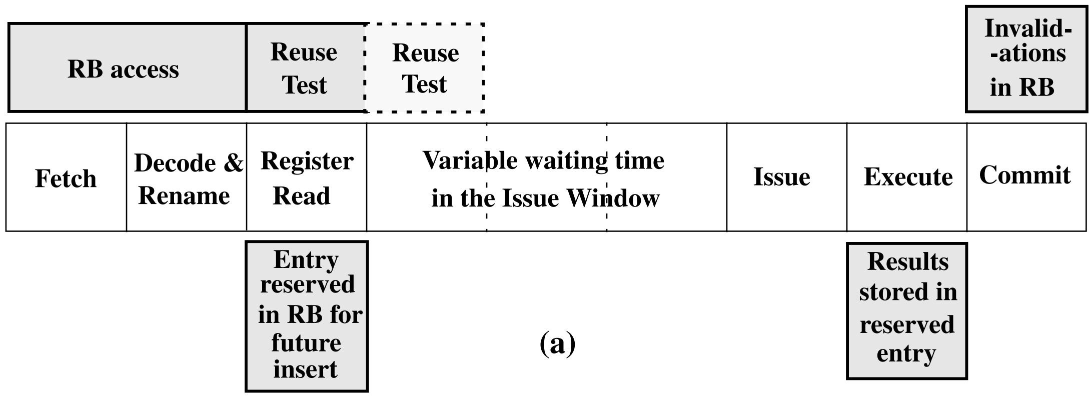
Three types of invalidations were used for keeping the RB consistent. Load invalidation, which marked the load instructions in the RB as stale when their memory locations were overwritten, was used by all four reuse schemes. Non-load invalidation was used in schemes S_n and S_{n+d} to invalidate instructions (independent instructions in scheme S_{n+d}) when their operands were overwritten. Dependent instruction invalidation was used in schemes S_{n+d} and S_{v+d} when instructions were evicted from the RB. Its purpose was to invalidate the instructions immediately following the evicting instruction in the dependent chain, and thereby avoid having dangling dependence pointers in the RB.

Later in this chapter (in Section 4.6), we discuss various issues with supporting invalidations in the RB and suggest some ways of dealing with these issues.

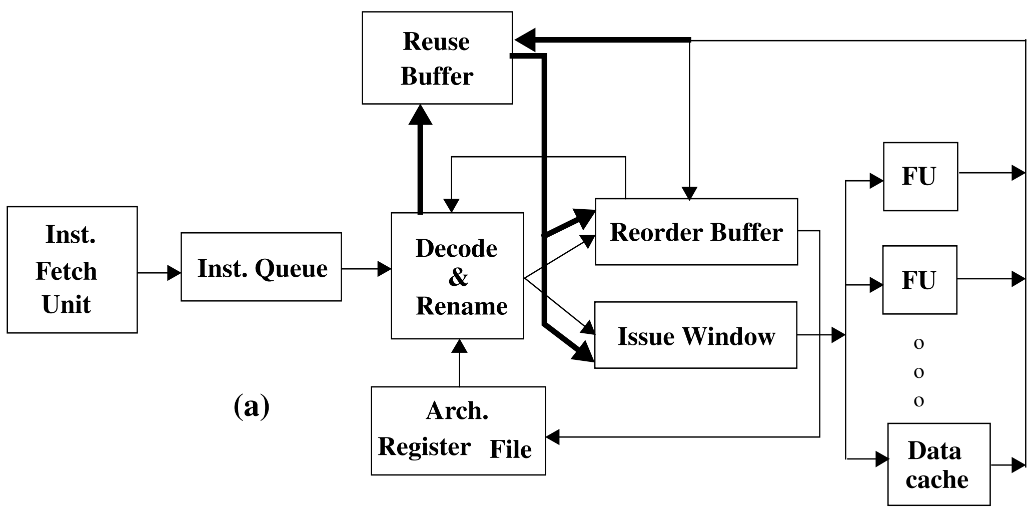
4.4 Microarchitecture with a Reuse Buffer

Figure 4.6 shows how an RB could be integrated with a generic microarchitectural pipeline. Figure 4.6 (a) shows the position of RB-specific operations — RB access, Reuse test, RB insert, and RB invalidations — in our simulated pipeline; and Figure 4.6 (b) shows the RB's position in the microarchitecture datapath. We next describe the working of the microarchitecture with the RB and then discuss the different issues involved in integrating the RB in a pipeline.

The *Instruction Fetch Unit* fetches and places the instructions in the *Instruction Queue*. Instruction decode and register renaming is done in the *Decode and Rename Unit*. In the Register Read stage, the operand values for the instruction are read either from the register file or from the *Reorder Buffer* (ROB) [42]. The RB access can be pipelined and can begin at the same time as the instruction fetch. At the Register Read stage the reuse test is performed on the entries read from the RB to see if their results are reusable. If a reusable result is found, the instruction does not need to be operated upon any further; it bypasses the *Issue Window* (IW), and proceeds directly to the ROB, where it is queued for retirement. Loads bypass the IW only if both micro-operations, address calculation and the actual memory operation, are reused. For some of the reuse schemes — e.g., scheme S_v — the reuse test may require more than one cycle to complete, as depicted by the extra Reuse test stage in Figure 4.6 (a). Multiple cycle reuse takes place as follows. The instruction is placed in the IW, while its reuse test is taking place. If the instruction is still not executed when the reuse test completes, the result obtained



(a)



(b)

Figure 4.6 (a) Position in the pipeline of RB-specific operations. (b) Generic microarchitecture with an RB. The additional data-path due to RB is shown in bold lines. The control lines (e.g., for access, invalidation etc.) are not shown.

from the RB is used and the instruction is not issued for execution. If, however, the instruction completes execution before its reuse test completes, then the reused result is ignored.

In both single- or multiple-cycle reuse, if a reusable result is not found in the RB, an entry is reserved in the RB where the result of the instruction will be placed after it is executed, setting it up for future reuse (in schemes S_{n+d} and S_{v+d}, the RST has to be updated accordingly). Once in the IW, instructions proceed as they would in any generic superscalar processor. After an instruction has executed, its results are stored in the reserved RB entry. In scheme S_v, the operand values are also stored in the entry at this time. When an instruction commits, depending on the reuse scheme, it invalidates appropriate results (or make other forms of state updates) in the RB.

Since one of the purposes of IR is to recover useful work from squashes (as discussed in Chapter 1), we allow speculative instructions to get inserted in the RB. However, we must take steps to ensure that the RB contents remain consistent and that no incorrect value is reused. What steps needs to be taken depends on the reuse test (and hence the reuse scheme) employed for performing reuse. For schemes S_v and S_{v+d}, nothing special needs to be done — the value-comparison based reuse test ensures that correct results are reused. For schemes S_n and S_{n+d}, however, the reuse test itself is not enough to sift out incorrect results — additional constraints needs to be enforced. We describe these constraints (and why are they needed) later, in Section 4.5.2. Schemes S_{n+d} and S_{v+d} use the structure RST, which like the rename table keeps track of the instructions in the RB which are the latest producers of registers. Since this table controls the reusability of instructions, it needs to be repaired accordingly after every misprediction. We do so by taking a checkpoint of the RST at every point where a speculative decision is made and restoring the RST appropriately on misprediction.

The RB contents also need to be maintained consistently in presence of context-switches and multiprocessor environment. Both these issues can be handled fairly easily. The issue of context switches can be handled in the same way as it is handled for virtual caches. The RB can either be invalidated on context switches or its entries can be augmented with process identifiers so that only entries from the current process can be reused. In multiprocessor environment, the loads in the RB of one processor will need to be kept coherent with the stores in other processors. This situation can be handled in the same way as is done for the L1-cache. The RB can maintain inclusion with the L2-cache, so that it is shielded from the external events. When a line in the L2-cache is replaced or invalidated (due to external events), the loads in the RB from that line can be invalidated.

Though we assumed that RB access takes a single cycle in our previous discussions, there is no need for this timing constraint since accesses may be pipelined. For example, the access can begin in the fetch stage of the pipeline after the PC of the instruction is available (since only the PC is required for indexing the RB, RB access can begin as early as the fetch stage, as illustrated in Figure 4.6 (a)). Other operations, such as invalidations, evicting entries to make way for new instructions etc., can be pipelined as well. For example, when the RB gets full, entries can be freed for future inserts. This will ensure that the free RB entries are always available, eliminating the search for a victim entry from the critical path. We will discuss more about RB invalidations in Section 4.6.

4.5 Reuse Schemes: Optimizations, Constraints and Variations

In the descriptions of the reuse schemes earlier in the chapter we focussed mainly on the basic operations necessary to understand the concept of reuse. In this section, we go a level deeper. We talk about some optimizations we can make to improve reuse rate; we discuss various constraints that need to be imposed upon the RB insertion and reuse policy to ensure that incorrect values are not reused; and we present other ways of performing certain operations in reuse scheme (like indexing the RB) and discuss their trade-offs. We begin by describing some optimizations we can make to the reuse schemes.

4.5.1 Optimizations

Preventing unnecessary invalidations: Reuse schemes use invalidations to mark instructions in the RB as stale whose invalidity (due to changes in the processor state) will not be otherwise detected by the reuse schemes. All schemes invalidate load instructions in the RB when stores write to matching memory addresses. Schemes S_n and S_{n+d} use invalidations to ascertain the reusability of the non-load instructions as well: these schemes invalidate instructions (independent instructions in case of scheme S_{n+d}) when their operand registers are overwritten during the course of execution. Many times these invalidations are unnecessary as the new value being written is the same as the old value; hence the instruction results in the RB would still be valid. We can prevent these unnecessary invalidations by comparing the new and the old values before invalidation and by only invalidating an instruction when the two values are different. Thus, in the case of load instructions in the RB, before invalidating a particular load we compare the store value with the load value, and if they are the same then the load is not invalidated. In schemes S_n and S_{n+d}, to prevent the invalidations of the non-load instructions we compare the old and the new operand values, allowing invalidation when the values are different.

Implementing this optimization for the non-load instructions (and for the address calculation part of the load instructions) in schemes S_n and S_{n+d} may require changes in the RB entry. This is because we do not store the operand values in the RB entry by default in these two schemes. To prevent invalidations, we may need to store them (so that the earlier values can be compared with the new operand values before invalidations), which will increase the size of the RB entries. However, an alternative solution is possible. Instead of storing the operand values in RB entries and comparing the newly created values of registers with their values stored in the RB, we can compare new values of registers with their old values in the register file. If the two values are the same, the invalidation signal is not sent to the RB at all (thereby, preventing the invalidation of the RB entries). Otherwise, invalidations are performed as usual. This approach avoids increasing the size of RB entries; however, it may require extra ports in the register file for reading the old values.

In the case of loads, however, performing this *one-point check* may not be feasible since reading the memory locations before writing them may not be possible. To implement invalidation-prevention, the store values will need to be compared with the load results stored in the RB. However, since the load values are stored in the RB anyway, doing so may not increase the RB entry size.

Resurrecting invalid instructions: Once the logic network for performing the RB invalidations is in place, we can use it to perform the opposite task as well — *i.e.*, *resurrection* or re-validation of the previously invalidated instructions in the RB. For example, while a store is invalidating loads in the RB, if it finds a load (with a matching address) that is invalid but has the same result as the store value then that load can be made valid again (*i.e.*, its memvalid can be set to true again). In schemes S_n and S_{n+d}, the invalid non-load instructions in the RB can be resurrected in a similar manner.

However, unlike in the case of the previous optimization, for resurrection, we will need to also store the operand register values in the RB for schemes S_n and S_{n+d} (which otherwise are not required to be stored). Hence, for non-load instructions, the size of the RB entry will increase for implementing this optimization.

We use both of the above optimizations in the schemes that we evaluate later in the chapter. We also show how important these optimizations are by comparing the results with and without these optimizations.

4.5.2 Constraints

Insertion constraints in scheme S_n: We mentioned earlier that we may want to insert speculative instructions in the RB so that we can recover useful work from control squashes. When allowing speculative instructions to enter in the RB, the insertion policy can no longer remain naive (*i.e.*, insert every instruction): certain conditions need to be checked to ensure that the instruction to be inserted will not lead to an incorrect reuse in the future.

How can speculative instructions cause wrong reuse? They can do so if the pipeline squash that takes place after they are inserted in the RB makes their results in the RB illegal and the reuse scheme employed has no mechanism for detecting this problem. We illustrate this with an example. Consider a piece of code shown in Figure 4.7. The code depicts a case where there are two statements defining the value of r1 (I1 and I3) and which of these two reach the use of r1 at I4 is decided by a branch at I2: if the branch is taken then the definition reaching I4 is I1 (r1 <- 0); otherwise it is I3 (r1 <- 1). First let us consider a scenario of wrong reuse with scheme S_n. Suppose I1 is executed and committed. The branch at I2 is initially predicted "not-taken", and instructions I3 and I4 are executed and inserted in the RB. Now, the branch is found to have been mispredicted, instructions are squashed, and the "taken" path is fetched. On the restart, instruction I4 is encountered on the "taken" path. The version of this instruction that was executed on the wrong path is still present in the RB in a valid state (since no intervening instruction overwrote the value of r1), and hence, it will get reused when the new instance of I4 is encountered. This will be an incorrect reuse because the version of I4 in the RB executed with a value of 1 for r1, while the current value of r1 is 0. So, what went wrong? The problem was that we inserted an "incorrect" execution of I4 (an execution that would have never occurred in a legal execution of the program) and had no way of detecting this problem before reusing the instruction. What we need to do is to insert only those speculative instructions in the RB whose results either remain valid even after they are squashed, or, if not then their results inserted in the RB should be made unreusable. To ensure this, we place

```
I1:    r1 ← 1
I2:    branch I4
I3:       r1 ← 0
I4:    r2 ← r1 + 4
```

Figure 4.7 A code sequence to illustrate the possibility of incorrect reuse when speculative instructions are inserted in the RB.

the following condition on the insertion of an instruction when using scheme S_n: a speculative instruction is inserted in the RB only if its source instruction is non-speculative. With this condition in place, the scenario of wrong reuse described above will not occur since the speculative execution of I4 will not get inserted in the RB because its source instruction (I3) is also speculative.

Insertion constraint for scheme S_{n+d}: Now, let us consider the same scenario for scheme S_{n+d}. In this case when instructions I3 and I4 are inserted in the RB, there will be a link pointing from I4 to I3. Because of this link, the speculatively executed I4 will not get reused (and correctly so) when I4 is encountered on the correct path since the producer of r1 will be I1 and not I3 (hence the dependency check will fail). However, the same problem that occurred in scheme S_n will arise if I4 were to be inserted as an independent instruction (for example, if for some reason I3 is not inserted in the RB). Thus, the condition for insertion of instructions in the RB for scheme S_{n+d} is insert a speculative instruction in the RB only if either its source instruction is non-speculative (just like for S_n) or if all its source instructions are present in the RB (and hence this instruction will get inserted in the RB as a dependent instruction).

Notice that no special conditions need to be enforced in the cases of schemes S_v and S_{v+d} as the reuse test based on value comparison unambiguously detects whether a result is valid.

Reuse constraints: Certain constraints need to be obeyed when performing instruction reuse in a pipelined processor. If the reuse status of a particular instruction cannot be definitely determined because of unresolved instructions ahead in the pipeline, then that instruction cannot be reused. For example, in all four schemes, load values are not reused if there is a store with unknown address or matching address ahead in the pipeline (only load addresses are reused in these cases). In schemes S_v and S_{v+d}, an instruction (an independent instruction in the case of S_{v+d}) is not reused if its current operand values are not available for comparison — *i.e.*, if the latest producer of its operand registers has not executed yet. Since schemes S_n and S_{n+d} do not use value comparison, they need to be more restrictive. If the source instruction of an instruction operand (an independent instruction operand in the case of scheme S_{n+d}) is present ahead in the pipeline (whether executed or not), the schemes S_n and S_{n+d} need conservatively assume that this source instruction will change the operand value, and hence, they don't reuse the instruction.

Having discussed the various constraints necessary to ensure correctness, we move on to the second topic of this section: we discuss the various options available for performing the different functions of the reuse mechanism.

4.5.3 Variations

Inserting instructions in the RB: Instructions can be inserted in the RB at different stages of the pipeline. In Section 4.4, we described one strategy for inserting instructions in the RB: allocate RB entries when instructions are decoded and populate these entries with values after the instructions have executed. Here we provide the rationale for adopting this policy and discuss other ways (and their trade-offs) for inserting instructions in the RB.

Another place in the pipeline where instructions can be inserted in the RB is the commit stage. This simplifies the maintenance of the RB state: nothing special needs to be done to ensure the correctness of the information in the RB or in the RST since wrong path instructions do not get inserted in the RB. However, inserting instructions here has a negative point too: since the speculative instructions do not get inserted in the RB, we may not be able to recover useful work from squashes (unless the squashed instructions were already present in the RB from their earlier execution).

Another option is to insert the instructions into the RB at the execute stage. Although this scheme allows reusing squashed instructions, it works only for schemes S_v and S_n. In schemes S_{n+d} and S_{v+d}, the dependence-links needs to be created between instructions that get inserted in the RB, and this requires that the insertion policy see the correct ordering between these instructions. This ordering information is not available at the execute stage of an out-of-order processor.

This leads us to the insert policy that we have already described in Section 4.4. We reserve the entries in the RB for the instructions at the decode stage, creating links between these entries based on the dependence relations that are visible at this stage. The index of these entries are then passed along with the instructions down the pipeline. After execution, the results of the instructions are written into their reserved RB entries using these indices. Although this will allow reusing squashed instructions, it makes the reuse scheme more complex: now the RST needs to be fixed after branch misprediction. This can be done by maintaining checkpoints of the RST at each branch (just like would be done for a rename table) and reverting the RST state to the checkpoint at the mis-predicted branch. Thus this solution requires extra hardware state for RST checkpointing.

Indexing RB: Earlier we presented one way of indexing into the RB, the way we employ in this thesis, by using the PC. There are other ways of accessing data in the RB; we describe some of them here. One approach is to use the operand values to form an index into the RB. The advantage of this approach is that it will map instances of an instruction that execute with different values to different RB locations, and thereby reduce collisions between such instances. This will allow capturing more reuse from instructions that execute with many different values. But the problem with this approach is that it serializes the indexing process for dependent chains of instructions: two instructions, if one is dependent on the other, cannot access the RB simultaneously since the result of the source instruction will be needed to form the index of the dependent instruction. This will prevent the reuse of the dependent chain of instructions in the same cycle.

Another approach can be to use the instruction itself as an index into the RB. This approach alleviates the problem inherent in the using operand values for indexing and allows dependent instructions to access the RB simultaneously. It has another advantage: it allows *computation reuse* — *i.e.*, it allows results computed by one static instruction to get reused by an instance of another static instruction, provided both static instructions have the same instruction word (*i.e.*, same opcode and same registers). The disadvantage of this approach is that it requires the instruction word itself to index into the RB, and hence, unlike the PC based approach it will not be able to overlap the RB access with the I-cache access.

Other information can be used along with the PC to reduce the collision between different instances of the static instruction. One such piece of information is the branch history register. The branch history is indicative of the path that program took to arrive at the current point. If an instruction executes with operand values different from the ones with which it executed earlier, then it is possible that sometimes this may be due to a different path followed by the program. Hence, using the branch history information in conjunction with the PC may allow us to map different instances of a static instruction to different sets in the RB, and hence, reduce the conflicts among them. In this thesis, however, we use just the PC for indexing in the RB and leave the evaluation of the above variations to future work.

RB organization: Until now we have only considered monolithic RBs. Other ways of organizing the RB are possible. We note that the *address* and the *memvalid* fields in an RB entry (along with the associative search for invalidations) are required only to maintain the integrity of the load values. The RB can be split into two buffers: one for storing load values, called the *load RB*, and the other for storing everything else (including entries for load addresses), called *the main RB*. Figure 4.8 shows a load entry for unified and partitioned RB. This RB organization has two advantages: first, the *address* and *memvalid* fields need not be maintained for entries storing non-load instructions, reducing the overall storage required for the reuse scheme; second, the main RB need not have the load invalidation logic as this logic would only be present in the load RB, which probably would be much smaller than the main RB. (The main RB will still have the invalidation logic for non-load instructions in scheme S_n and S_{n+d}.)

One disadvantage of this organization is that it may make the reuse process more complex. For example, now a pointer will need to be kept for loads from their load RB entry to their main RB entry (as shown in Figure 4.8) to ensure correct load value reuse; this pointer will also need to be checked for validity during the reuse test.

In this thesis, however, we assume a unified RB, and leave the task of evaluating different RB organizations for future work.

4.6 Invalidations in RB: Issues and Alternatives

To maintain the RB consistent with the rest of the processor state, we employ three different types of invalidations (shown in Table 4.1). Since these invalidations require CAM [49] accesses into the RB, supporting them may make the RB design complex. In this section, we present various issues in supporting invalidations in the RB and discuss some ways of addressing these issues.

Some of the issues are as follows. The CAM accesses are likely to be slow for large RBs since they require full access paths through the RBs. The latency of these accesses, and hence that of invalidations, may limit the size of an RB. CAM ports may be expensive to implement, making the support for multiple of them in an RB difficult. This may be an issue for scheme S_n and S_{n+d} which may require to perform several invalidations per cycle. Finally, significant power may be expended if invalidations are frequent, since invalidation operations drive invalidation lines that may carry large capacitive loads.

We discuss some of the ways in which the above issues may be addressed. The latency of the CAM logic can be reduced by partitioning the RB. Each partition will have fewer RB entries, making it feasible to provide fast invalidation paths. In most practical implementa-

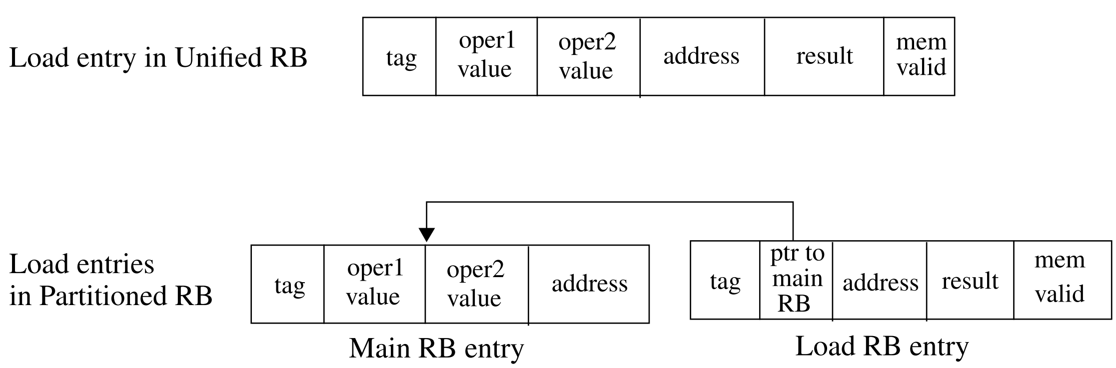


Figure 4.8 Load entries for scheme Sv in unified and partitioned RB.

tions, the RB would be partitioned anyway for other reasons (e.g., for reducing the decoder latency); the CAM logic can benefit from this partitioning. Other ways of RB partitioning are also possible; for example, we have already discussed one type RB partitioning for reducing the cost and latency of load invalidations in Section 4.5.3 (under *RB organizations*).

Although partitioning the RB may alleviate the invalidation latency, it may not solve other issues due to CAM logic — namely difficulties in providing multiple ports and power consumption. These issues may be especially problematic for the invalidations that occur frequently, such as non-load and the dependent instruction invalidations (load invalidations are caused only by stores and hence are relatively infrequent). One way to solve these issues is to use invalidations as a fall-back mechanism, instead of as a primary mechanism, for maintaining RB consistency. As a fall-back mechanism, it will be invoked infrequently, only when the alternative (and possibly less expensive) primary mechanism is unable to maintain consistency in the RB. Hence, the invalidation paths may not have to be as aggressively optimized, higher invalidation latencies may be tolerated, and fewer CAM ports in the RB may be sufficient. In the next two sections, we describe how we can reduce the frequency of non-load and dependent instruction invalidations.

4.6.1 Non-load invalidations

These invalidations are performed in schemes S_n and S_{n+d} for preventing the reuse of stale instruction instances. Their frequency can be reduced in several ways.

Exploiting existing reuse constraints: As mentioned in section Section 4.5.2 (under *Reuse constraints*), schemes S_n and S_{n+d} do not reuse instructions (independent instructions in case of S_{n+d}) if their source instructions (*i.e.*, instructions writing their operand registers) are present ahead in the pipeline. We can exploit this constraint to reduce the number of invalidations. Since the purpose of invalidations is to prevent the reuse of instructions whose operand registers have changed, we will only need to invalidate instructions in the RB when there is a danger of such an incorrect reuse. With the reuse constraint mentioned above, this danger will exist only when there is no source instruction for that register in the pipeline. Thus, we only need to invalidate instructions in the RB that use, for example, register r if the instruction being committed is the last source of that register in the pipeline. If there are other sources of register r in the pipeline, they will prevent the reuse of instructions that use register r (unless the instructions using r are linked to them through dependence pointers), thereby also prevent any incorrect reuse. The information whether there are more sources of a particular register in the pipeline can be obtained from the rename table.

Version number for registers: We can also reduce the number of invalidations by using version numbers for registers, instead of invalidations, for detecting stale instruction instances in the RB. A version number counter can be associated with every architectural register, which can be incremented every time a new value is written into its corresponding register. The version numbers for the operand registers can be stored in the operand field in the RB entry. The reuse test can then be changed from checking the valid bit to matching the operand register version numbers in the RB with the current ones. With this arrangement, the non-load invalidations will only be needed when version number counters overflow.

The above two mechanism may help reduce the number of invalidations significantly. This decrease can help reduce the number of CAM ports required in the RB for non-load invalidations and may also decrease the power consumption in the RB. However, if we reduce the number of CAM ports in the RB, we may need a small buffer to save the extra invalidations for times when there are more invalidations than the number of ports. The saved invalidations can be performed at a later time when the CAM ports are available. This buffer will have to be checked during the reuse test to see whether any outstanding invalidations exist for the operands of the instruction being tested for reuse (in which case, the instruction should not be reused). Although these mechanism are likely to reduce the RB complexity, they have one disadvantage: they will reduce the frequency of invalid-instruction resurrection, one of the reuse scheme optimizations discussed in Section 4.5.1.

4.6.2 Dependent instruction invalidations

These invalidations are performed in schemes S_{n+d} and S_{v+d} when instructions are evicted from the RB. The purpose of these invalidations is to clean up the dangling dependence pointers in the instructions immediately following the evicting instruction in the different dependent chains (note that the whole chains are not traversed). The frequency of these invalidations can also be reduced by using version numbers, instead of invalidations, to detect dangling dependence links. A version number counter can be maintained for each RB entry. This counter can be incremented every time a new instruction instance is inserted in its corresponding RB entry. When dependence links are made in the RB, the version number of the source RB entry can also be stored in the dependent RB entry. The reuse test for dependent instructions can then be changed from just checking the RB indices for establishing dependence to also checking the version numbers of the source instructions with the version numbers stored in the dependent instructions. The dependent instructions are not reused if there is a version number mismatch. With this mechanism in place, the dependence instruction invalidations will only need to be performed when a version number counter overflows.

In this thesis, however, we maintain consistency in the RB using the three types of invalidations. Further investigation of the various alternative mechanisms presented in this section are left as future work.

4.7 Experimental Evaluation

The description of our simulator along with the parameters for the base configuration were presented in Chapter 2. We extended this base simulator to incorporate the RB and the four instruction reuse schemes described earlier. The RB is integrated with the processor pipeline as described in section 4.4.

In our simulations, the RB is capable of supporting 4 reads, 4 writes, and 4 independent invalidations simultaneously. We assume that all RB accesses — read, write or invalidate — complete in one cycle, and that all schemes can reuse instructions in a single cycle (*i.e.*, in the Register Read stage). We also assume that schemes S_v, S_{n+d}, and S_{v+d} can reuse multiple dependent results in a single cycle. (The impact of multiple-cycle reuse test is investigated later, in Chapter 6.) The maximum length of a dependence chain reused in a cycle is equal to the read bandwidth of the RB, which is 4 in the simulated configuration. In our reuse schemes we employed the optimization of invalidation prevention and resurrection as described in Section 4.5.1. These optimizations were used for loads in all schemes and for non-load instructions in schemes S_n and S_{n+d}. In this chapter, we evaluate IR with one set of experimental parameters. The study of its sensitivity to some of these parameters (e.g., window size, issue width, pipeline length, and reuse latency) is conducted later, in Chapter 6.

4.7.1 Experiments and Results

In subsequent sections, we present the results of several experiments we conducted to evaluate the concept of dynamic instruction reuse. We first show the percentage of total dynamic instructions that are reused and present the impact of this reuse on the performance of the baseline processor. We then present what types of instructions get reused and how much contribution does each instruction type make to total reuse. Then, we categorize the total instruction reuse into squash reuse and general reuse, and show the contribution of either category to total speedup. We evaluate the importance of the various optimizations we described in Section 4.5.1, and then present statistics on the lengths of instruction chains reused. Finally, we perform a brief evaluation of the impact of RB associativity on percentage instructions reused and speedups (a more thorough evaluation of RB associativity is performed in Chapter 5).

For most of our experiments we use fully-associative RBs of three different sizes: 256, 1024, and 4096 entries with LRU replacement policy. As mentioned earlier, we make no attempt to be selective about what instructions get inserted into the RB; that will be the subject of investigation in Chapter 5.

4.7.1.1 Instructions Reused

In this section we present the percentage of total dynamic instructions reused for the four different schemes, with 3 different RB sizes for each scheme. Integer and graphics benchmarks are presented in Figure 4.9, while FP benchmarks are found in Figure 4.10. We see that all the analyzed benchmarks exhibit significant instruction reuse, especially for the larger buffer sizes. For example, scheme S_v with 4096 entries reuses 76% of dynamic instructions for *m88ksim* and *vortex*, 67% for *perl*, 48% for *gcc*, 48% for *viewperf*, and 37% for *povray*. Even for small RB sizes, the percentage of instructions reused are significant for several benchmarks (24% for *li*, 19% for *gcc*, 26% for *mpeg*, and 16% for *go*).

From the figure we can make several observations about how effective the reuse schemes are in reusing instructions, what impact increasing the RB size has on the reuse rate for different schemes, and how much reusability is exhibited by different benchmarks. We present several such observations below.

Comparing the four schemes with each other, we see that, in general, scheme S_v reuses the most number of instructions. This is because the reuse test in this scheme is based on direct value-comparison, and hence is the most accurate of all the reuse tests. (In some cases, other schemes work better; we describe the reasons when discussing those schemes below)

Scheme S_n reuses the least number of instructions (in general). This should be expected since it is the simplest and, hence, the most conservative of all schemes. Invalidations occur very frequently in this scheme (being done every time a register or memory location is written). Also, it does not reuse instructions if the sources of their operand registers are present ahead in the pipeline. Because of these restrictions the number of instructions that are available for this scheme to reuse are small, hence this scheme does not benefit from the increase in the RB size. However, frequent invalidations have one advantage: they help utilize small size RBs (256 entries) better by only retaining more likely instructions in the RB. In fact, scheme S_n performs better than scheme S_v for with a 256-entry RB for some benchmarks, like m88ksim, perl, and viewperf. Since the number of invalidations in scheme S_v are small, instructions that are not likely to be reused remain in the RB; hence the RB is not utilized as

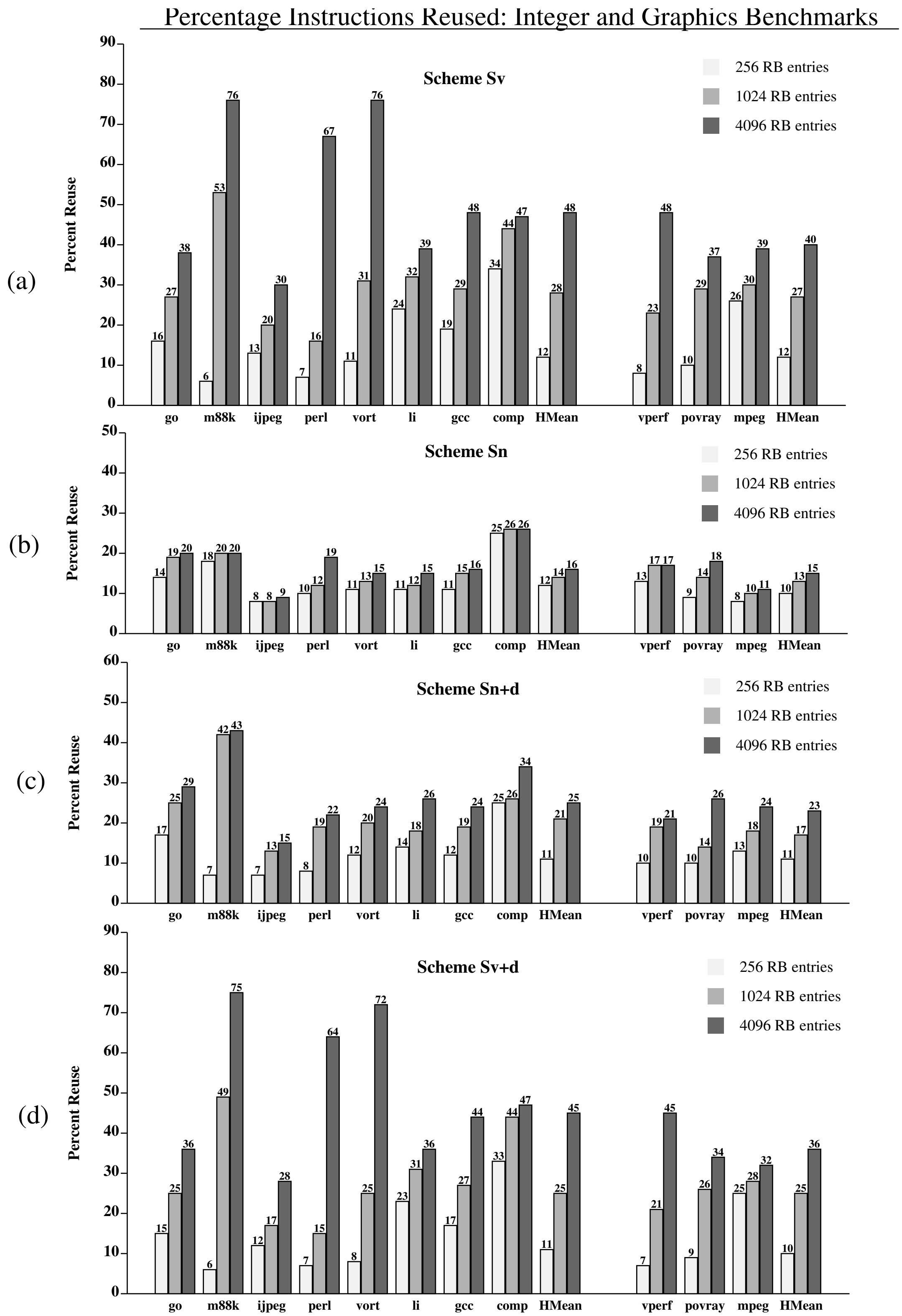


Figure 4.9 Percentage of instructions reused for RB sizes: 256, 1024 and 4096 entries. The RB in these experiments was fully associative. (a) Scheme S_v (b) Scheme S_n (c) Scheme S_{n+d} (d) Scheme S_{v+d}. HMean stands for harmonic mean.

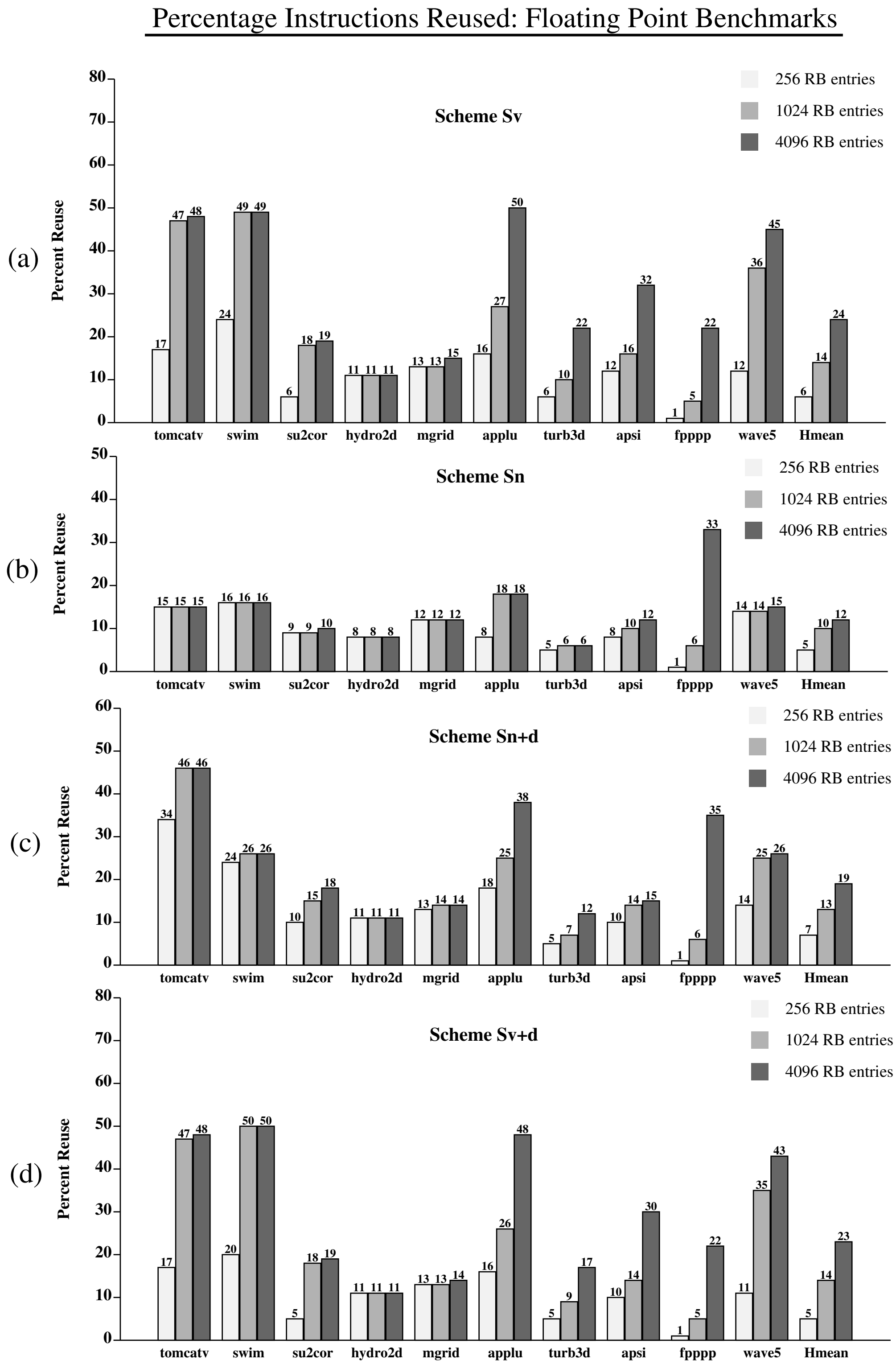


Figure 4.10 Percentage of instructions reused for RB sizes: 256, 1024 and 4096 entries. The RB in these experiments was fully associative. (a) Scheme S_v (b) Scheme S_n (c) Scheme S_{n+d} (d) Scheme S_{v+d}. HMean stands for harmonic mean.

efficiently. On average scheme S_n performs as good as scheme S_v with 256-entry RBs.

Scheme S_{n+d} has lower frequency of invalidations than scheme S_n (since only independent instructions are invalidated) and has the ability to reuse dependent chain of instructions. Consequently, it is able to reuse more instructions than scheme S_n, and it benefits when the RB size is increased to 1024 entries. However, invalidations and the fact that it does not reuse independent instructions if sources of their operand registers are ahead in the pipeline (and thereby does not start a chain) restrict the amount of instructions that can be reused; hence it does not benefit as much from the 4096-entry large RBs for most benchmarks (except *li*, *povray*, *applu*, and *fpppp*). (We study the sensitivity of this restriction of IR to changes in underlying microarchitecture in Chapter 6).

The amount of reuse captured by scheme S_{v+d} is comparable to that captured by scheme S_v. This shows that restricting the value-comparison to independent instructions and using dependence information for reusing the dependent instructions does not sacrifice the reuse rate appreciably. Using the dependence information for reusing the dependent instructions, facilitates collapsing chains of dependent instructions in a cycle.

In FP benchmarks, on average less amount of reuse is captured as compared to integer and graphics benchmarks (24% in FP versus 48% in integer for the 4096-entry RB with scheme S_v). Also, we observe that for many FP benchmarks increasing the RB size does not improve the reuse rate, indicating that the capacity misses in the RB are very high for these benchmarks and that the increase in the RB size up to 4096 entries is unable to eliminate them.

Finally, it may seem counter-intuitive that even with resurrection, schemes S_n and S_{n+d} do not perform as well as scheme S_v. It would be expected that if scheme S_v is able to reuse an instruction then that instruction should get resurrected (and hence reused) in schemes S_n and

S_{n+d}. The reason for this apparent mismatch is that the resurrection takes place at the commit stage and, hence, does not help if the instructions that would have reused the resurrected entries have already gone past the reuse stage in the pipeline.

4.7.1.2 Speedups

Figures 4.11 and 4.12 show percentage speedups ($(IPC_{withRB}-IPC_{withoutRB})*100/IPC_{withoutRB}$) obtained with the different reuse schemes for varying the RB sizes. The speedups are not as impressive as the percentage of instructions reused, however, they are still significant in many cases; they range from 1% to 7% for a 256-entry RB, from 4% to 12% for a 1024-entry RB, and from 11% to 19% for a 4096-entry RB.

Comparing the percent instruction reuse results in Figures 4.9 and 4.10 with the speedup results, we see the speedup results in general follow the same trend as the reuse results: in cases where more instructions are reused, more speedup is observed. Scheme S_v and S_{v+d} show the highest speedups, specially for the large RBs, with S_v averaging 13% for integer, 12% for graphics, and 10% for FP benchmarks with the 4096-entry RBs. Although S_n and S_{n+d} don't show large speedups on average (with averages less than 6% and 10% respectively), they still show significant speedups for some benchmarks, such as *tomcatv* (8% and 19%), *mgrid* (12% for both), *applu* (10% and 19%), and *povray* (6% and 7%).

4.7.1.3 Reuse Characteristics

To study the reuse characteristics of different instruction types, we divide the instructions into the following broad categories: *loads*, *address calculations*, *control* and *integer*. The category *address calculations* consists of loads and stores for which only the address calculation

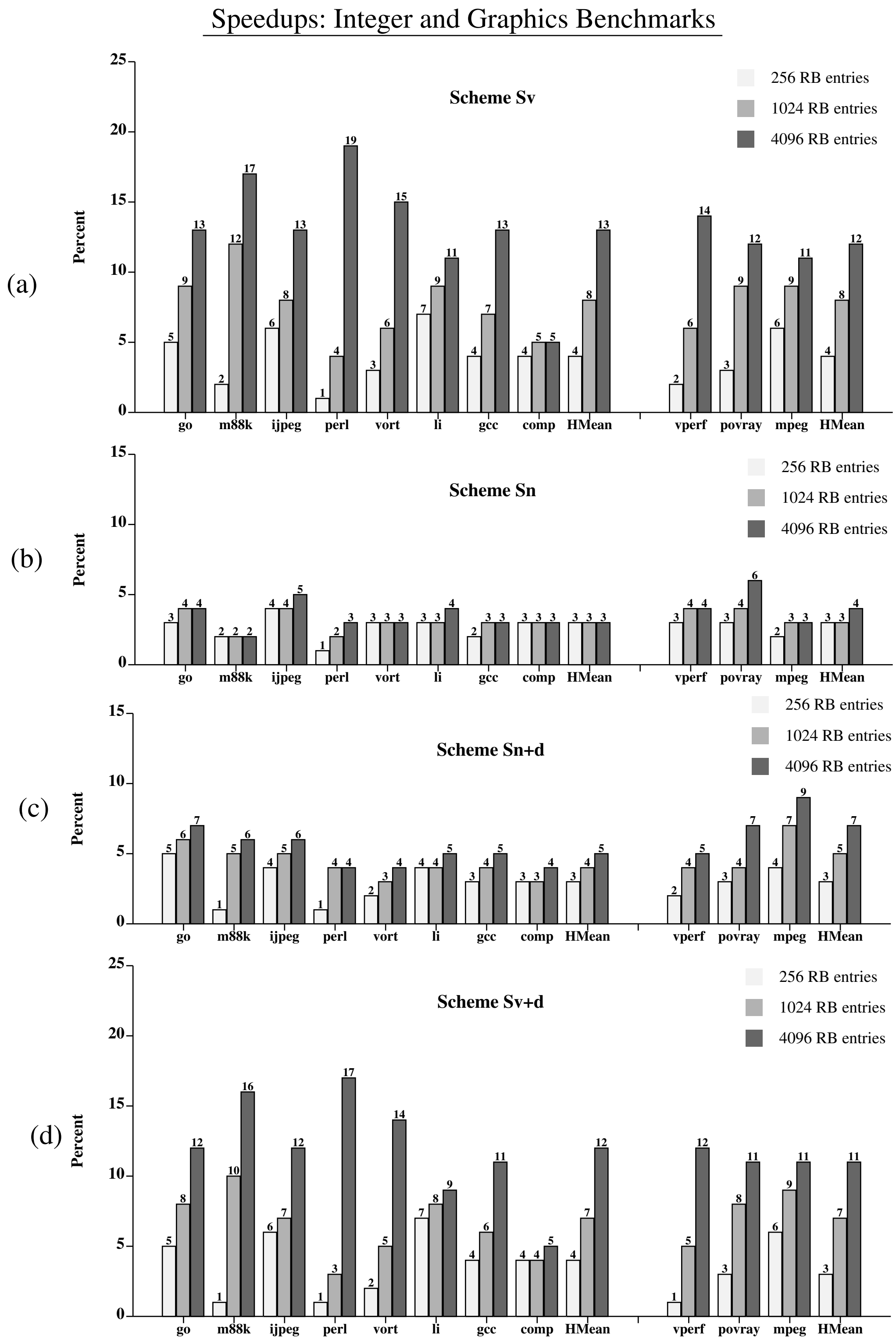


Figure 4.11 Speedups obtained due to instruction reuse. The numbers are presented for RB entries 256, 1024 and 4096. (a) Scheme S_v (b) Scheme S_n (c) Scheme S_{n+d} (d) Scheme S_{v+d}. HMean stands for harmonic mean.

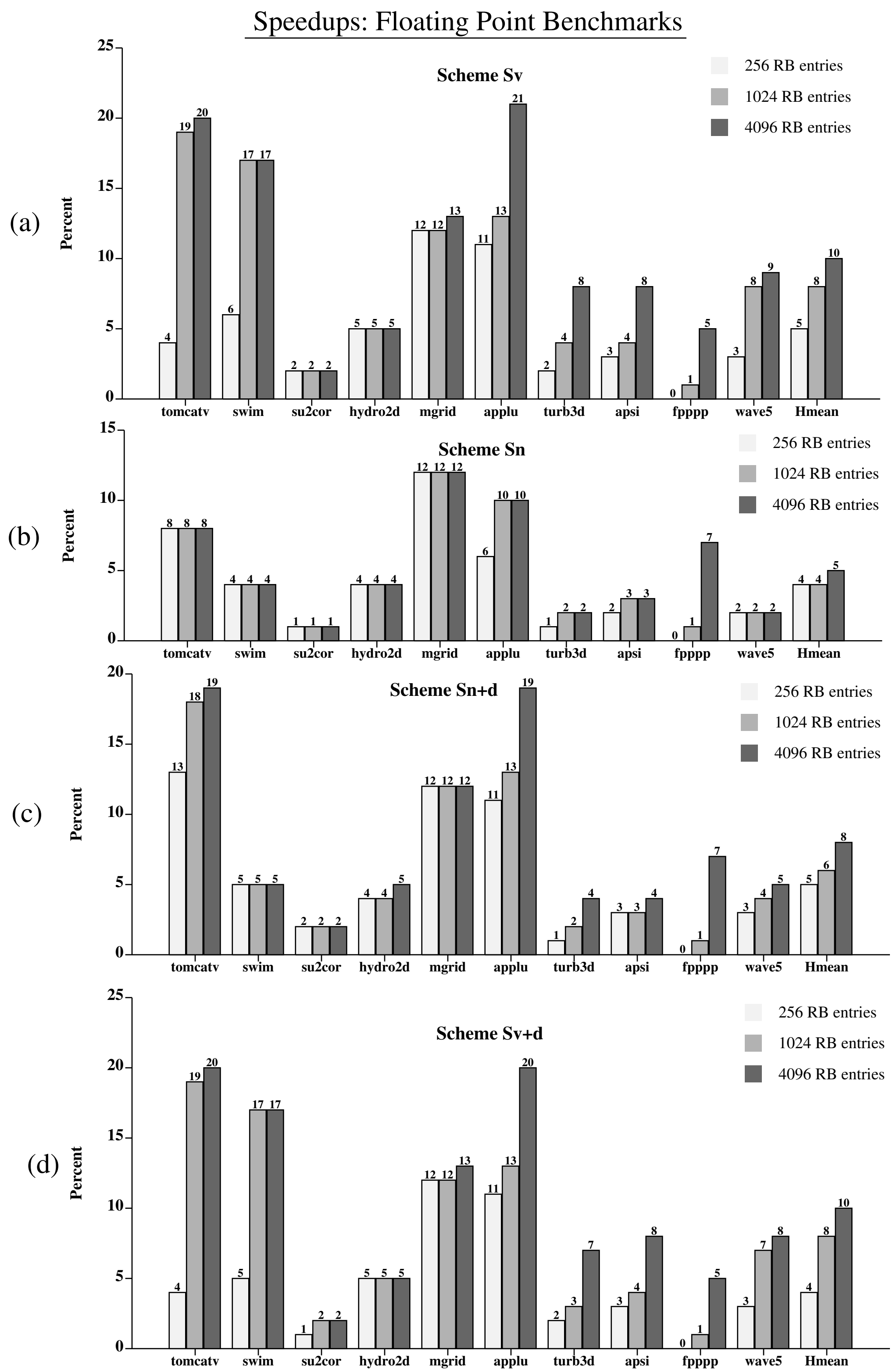


Figure 4.12 Speedups obtained due to instruction reuse for floating-point benchmarks. The numbers are presented for fully-associative RB with entries 256, 1024 and 4096. (a) Scheme S_v (b) Scheme S_n (c) Scheme S_{n+d}. HMean stands for harmonic mean.

part is reused. (As noted earlier, for stores we reuse only the address calculation and not the actual memory operation). The integer instructions are further divided into three subcategories based on the type of operands: *two reg operands, one reg operands* and *immediate*.

Reuse rates of different instruction categories: Table 4.2 shows the percentage of instructions reused from each category using a 4096 entry RB. The numbers are shown for integer,

Instruction Categories		Instruction Reused (%)			
		S_v	S_n	S_{n+d}	S_{v+d}
SpecInt '95					
Loads (value)		31.6	8.1	12.3	30.3
Address Calculations		41.0	16.4	21.5	40.2
Control		45.4	2.4	16.5	42.0
Integer	two reg operands	43.8	15.2	24.8	41.2
	one reg operand	54.2	20.1	34.5	52.2
	immediate	91.2	98.7	98.7	91.0
Floating Point		8.2	0.0	5.3	6.9
SpecFP '95					
Load (value)		27.5	18.8	25.2	28.2
Address Calculations		19.9	11.3	15.1	19.3
Control		44.2	1.2	22.3	41.8
Integer	two reg operands	19.9	5.4	15.8	19.5
	one reg operand	38.9	10.8	31.9	38.7
	immediate	77.1	79.1	79.0	74.3
Floating Point		6.1	0.2	2.1	5.0
Graphics					
Load (value)		36.9	18.7	23.0	35.3
Address Calculations		36.7	16.4	21.6	33.9
Control		39.2	3.1	15.1	35.9
Integer	two reg operands	34.0	10.5	18.5	29.0
	one reg operand	45.6	19.3	30.3	41.9
	immediate	74.3	99.2	99.1	74.1
Floating Point		6.6	0.3	1.5	4.6

Table 4.2 Percent reuse per instruction category for a 4096 entry RB

floating-point, and graphics benchmarks (averaged over the respective benchmark set). Thus the numbers in the table should read as, for example, 43.8% of all integer instructions with two register operands in integer benchmarks are reused with scheme Sv. As expected, most computation involving immediate constants is reused. Likewise, reuse of address calculation is also not very surprising. Somewhat surprising is that a large number of load instructions could be reused (an average of 21.2% for scheme S_v). This reduces the demand for data cache bandwidth, which can possibly be exploited by reducing the number of data cache ports.

Another observation we can make from the results presented in the table is that very few floating point instructions are reused (e.g., even for FP benchmarks only 6% of the FP instructions were reused by the most aggressive reuse scheme). This should not come as a big surprise. There are two reasons for low reusability of FP instructions. First, the FP instructions often operate on large amounts of data: operating on big matrices, or arrays. To capture significant amount of floating point repetition, the RB may need to be able to buffer many instances of the same instructions. For an RB of a limited size this may not be possible, hence the reuse rate for the FP instructions are low. Second, as mentioned in Chapter 3, much of the repetition in the programs is often due to instructions that perform overhead work — like accessing complex data-structure elements, calculating memory addresses, function prologue and epilogue, etc. Floating point instructions are seldom used for performing these overhead work and are mostly used to perform the "actual computation" on the data input to the program. As shown in the last chapter, less repeatability falls on instruction slices originating from the program input data, hence floating point instructions are a less repeatable category to begin with.

Contribution of different instruction categories to total reuse: Figure 4.13 shows the con-

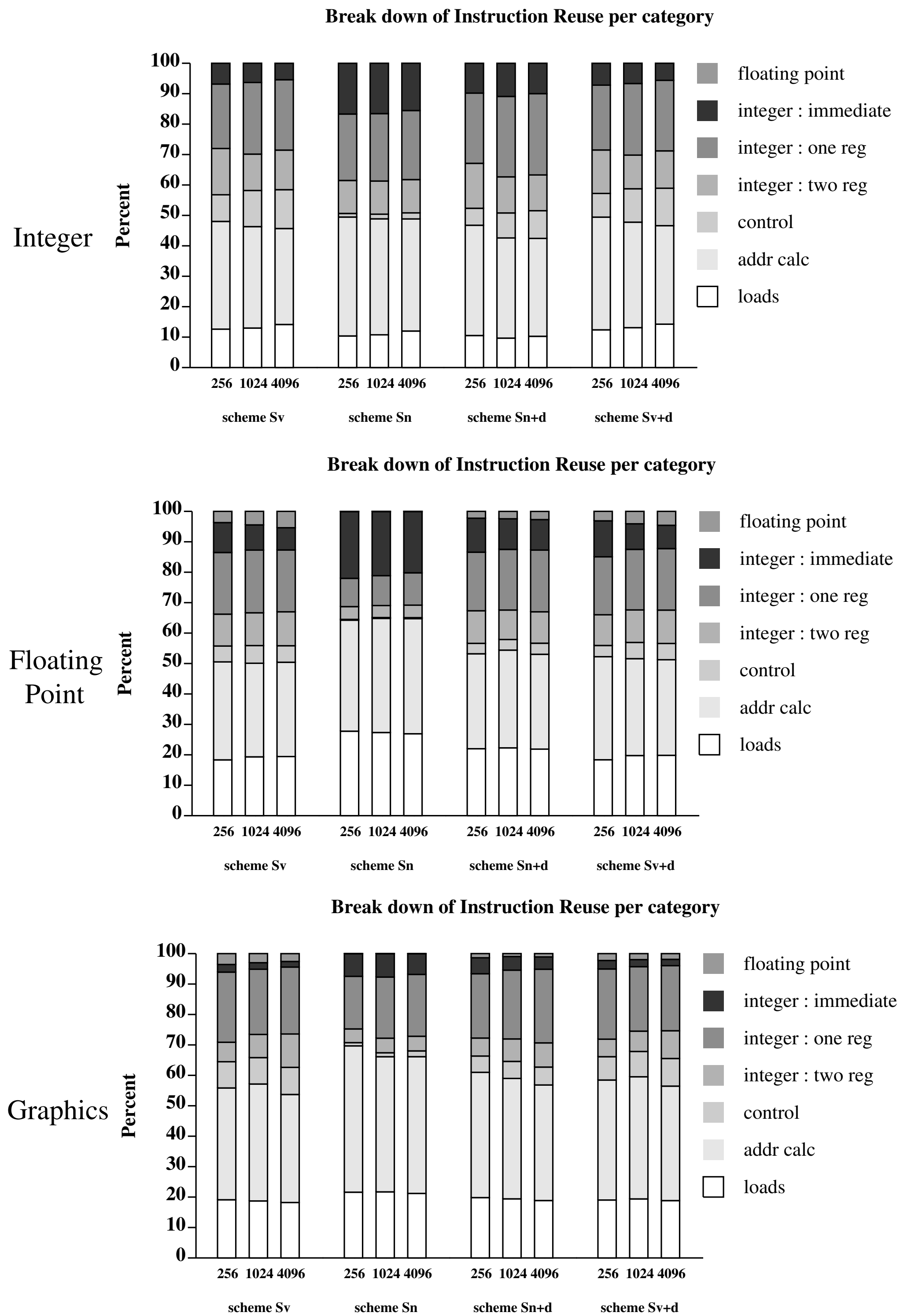


Figure 4.13 Contribution of each instruction category to total reuse. These numbers are average over all benchmarks for a full associative RB. Note that in some bars above certain sub-bars are indiscernible : FP in integer benchmarks and for scheme S_n in other benchmarks; control for scheme S_n in FP-benchmarks.

tribution of each instruction category to the total instruction reuse for 3 different RB sizes for each reuse scheme. The figure shows three plots, one for each benchmark set, and each plot is an average over all programs in that benchmark set. We observe that each instruction category makes a measurable contribution to the total instruction reuse; reuse is not limited to some particular instruction type. However, some categories make more contributions than the others. For example, it is worth noting that almost 40-50% of the reuse comes from the load instructions (about 15%) and address calculations (25-35%). Also, the FP instructions make a very small contribution to the total reuse (most contribution being ~7% in the case of FP benchmarks for scheme S_v with a 4k-entry RB). This would be expected from our previous observation that FP instructions are not as amenable to reuse as integer instructions.

4.7.1.4 Lengths of reused dependence chains

As mentioned several times in this thesis, IR can reuse chains of dependent instructions in a same cycle. In Table 4.3, we show the length distribution and the average lengths of such reused chains. The distribution numbers, which are averages over all benchmarks programs, show percentages of all chains of instructions that are 1-instruction long, 2-instruction long, and so on. The maximum length in this distribution is 4 because we can reuse at the most 4 instructions per cycle, since our base machine is 4-way superscalar. From the table, we see that overall most of the reused chains are 1-instruction long. However, significant amount of 2- and 3-instruction chains are also reused. For example, in scheme S_{n+d} for integer benchmarks, on average, 17.7% of the dependent chains are 2 instruction long, 4.98% are 3 instruction long, and so on. We also see that the average lengths of the dependent chains of instructions that get reused in a cycle to be typically between 1.30 and 1.45.

4.7.1.5 Squash Reuse vs. General Reuse

As we have described in Chapter 1, instructions reuse can occur due to *squash reuse* or *general reuse* (both illustrated in Figure 1.1 and Figure 1.2 of Chapter 1, respectively). In this section we present the relative contribution of these two types of reuse to the total number of instructions reused and the overall performance improvement. First, we describe how we separate out the contribution of the general and squash reuse. To do so, we simulate each benchmark twice. The first simulation is the usual simulation, like that done for other results in this chapter; this gives us the overall reuse rates and speedups. In the second simulation, we do everything exactly as in the first simulation except that we only reuse those instructions that

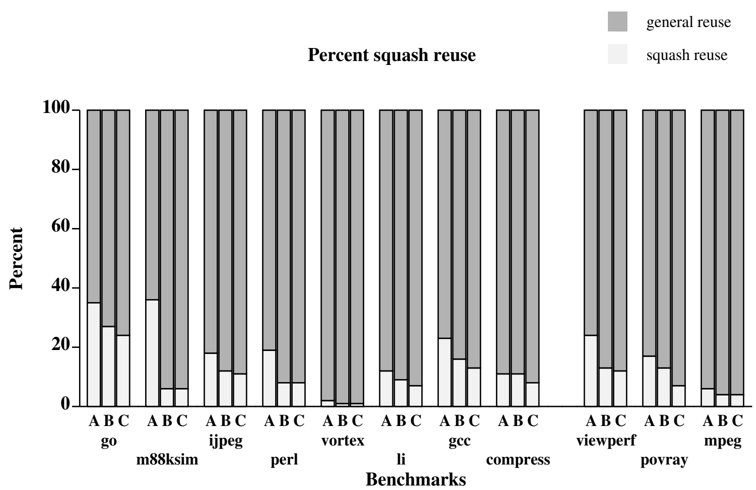
Schemes	Dependent Chain Length Distribution (%)				Average Length
	1	2	3	4	
SpecInt '95					
S_v	70.83	19.96	5.72	3.49	1.43
S_{n+d}	75.66	17.70	4.98	1.65	1.33
S_{v+d}	70.10	21.86	5.29	2.74	1.41
SpecFP '95					
S_v	63.07	30.49	4.64	1.78	1.45
S_{n+d}	62.07	32.78	3.77	1.37	1.45
S_{v+d}	64.38	30.03	4.00	1.59	1.43
Graphics					
S_v	67.40	25.49	4.83	1.27	1.42
S_{n+d}	69.62	26.00	3.30	1.07	1.36
S_{v+d}	68.07	26.35	3.93	1.64	1.39

Table 4.3 Percentage of dependent chains that are of lengths 1, 2, 3, and 4. The number are averages over programs in each benchmark suite. Average chain lengths are also shown. The numbers are not shown for scheme S_n, since it does not reuse dependent chain of instructions. All numbers are for a full-associative 4096-entry RB.

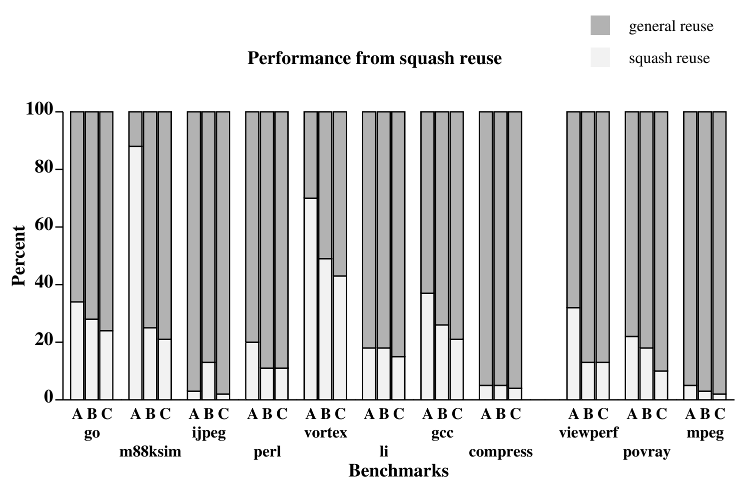
were entered in the RB speculatively and were later squashed. To recognize such instructions in our simulator, at the time of squash recovery, for each executed instruction that we throw away we set a flag in the RB entry of that instruction. Only those RB entries which have this flag set are considered for reused. This flag is reset after first reuse. The reuse rates and speed-ups obtained in this manner give us the contributions of the squash reuse to these two metrics. The remaining portion in the overall reuse rates and overall performance is attributed to general reuse.

In Figure 4.14(a), we present a break down of the number of instructions reused into squash reuse and general reuse. In this figure we show the information for integer and graphics benchmarks for all three RB sizes (256-, 1024-, and 4096-entry) with scheme S_{n+d} (breakdowns for other schemes and for floating point benchmarks are shown in Appendix A). We observe that the relative contribution of squash reuse to total reuse decreases as the RB size increases. For a 256-entry RB, the amount of reuse due to squashes is, in general, within the range of 10-30% (with some exceptions); for a 4096-entry RB this range is typically 5-10%. The squash reuse contributions are more for small RBs because, small RBs are less effective in performing general reuse since for that instructions often need to be buffered for a long period of time. Many squashed instructions, however, are re-encountered in a short while after the squash, and hence can be reused by a small RB. As the size of RB is increased, it become more effective in performing general reuse, and given massive amounts of instruction repeatability present in most programs, the contributions to total reuse of general reuse increases, and accordingly the contribution of squash reuse decreases.

Figure 4.14 (b) separates the performance obtained by squash reuse from that obtained by general reuse. As was the case in Figure 4.14 (a), we observe that the contribution of squash reuse to overall performance decreases with the increase in the RB size. Barring a few exceptions (e.g., *m88ksim* and *vortex*) the typical contribution of squash reuse to total performance improvement is between 5-25%. We also observe that for several benchmarks (e.g., *m88ksim*, *vortex*, and *gcc*) the fraction of the speedup attributed to squash reuse is greater than the contribution of squash reuse to the total number of instructions reused (compare with Figure 4.14 (a)). This suggests that squash reuse is more time critical than general reuse — the squash



(a)



(b)

Figure 4.14 Breakdown of percentage instruction reused (shown in (a)) and performance (shown in (b)) in terms of *general* and *squash* reuse using scheme S_{n+d}. Bar 'A' is for a 256-entry RB, 'B' is for a 1k-entry RB, and 'C' is for a 4k-entry RB.

Schemes	Load invalidations		Non-load invalidations	
	Prevented (% of total invalidation attempts)	***Resurrected*** (% of total successful invalidations)	***Prevented*** (% of total invalidation attempts)	***Resurrected*** (% of total successful invalidations)
SpecInt ‘95				
Scheme S_v	41	29		
Scheme S_{v+d}	41	30		
Scheme S_n	55	42	22	20
Scheme S_{n+d}	44	20	35	44
SpecFP ‘95				
Scheme S_v	41	24		
Scheme S_{v+d}	36	20		
Scheme S_n	44	26	37	41
Scheme S_{n+d}	38	23	56	34
Graphics				
Scheme S_v	36	28		
Scheme S_{v+d}	28	45		
Scheme S_n	36	47	47	37
Scheme S_{n+d}	25	42	19	40

Table 4.4 Invalidation prevention and resurrection rates for different reuse schemes. The numbers are averages over all benchmark programs for 1024-entry RB.

penalty impacts the bottom line more than the latency of an instruction (or a set of instructions), especially in a dynamically scheduled processor.

4.7.1.6 Impact of reuse scheme optimizations

Until now all experiments that we performed used the two reuse optimizations presented in Section 4.5.1: the invalidation prevention and the invalid instruction resurrection. In this section, we show the number of times these optimizations were applied in prior experiments and the amount of impact they had on the percentage instructions reused and speedups gained.

To show how often these optimizations were applied, we present in Table 4.4 the percentage of invalidations that were prevented and the percentage of invalid instructions that were resurrected for different reuse schemes. The numbers presented are the arithmetic averages over the benchmark programs and are for a 1024-entry RB. We see that a significant percentage of invalidations, both load and non-load, were prevented and resurrected. For example, for integer benchmarks, 44% of load invalidations were prevented and 20% of invalid loads were resurrected on average when using scheme S_{n+d}. The corresponding numbers for non-load invalidations were 35% and 44%, respectively. In general, the percentages of invalidation preventions and resurrections were between 20 to 40% (with some exceptions).

To show the impact of these optimizations on the reuse results, we present in Figures 4.15 and 4.16 the average percentage instructions reused and the average speedup gained with and without these optimizations. From these figures, we see that schemes S_n and S_{n+d} are affected more significantly by these optimizations than the other two schemes. For example, for scheme S_{n+d} with a 4096-entry RB, the average reuse decreases from 25% to 18% for integer benchmarks and from 23% to 14% for graphics benchmarks when not using these optimizations (Figure 4.15); the corresponding decrease for scheme S_v is from 48% to 46% and from 40% to 39%, respectively. Similar trends are also seen for the speedup results (Figure 4.16). This difference in impact is expected, since schemes S_n and S_{n+d} are more dependent on invalidations than schemes S_v and S_{v+d}. We also note from the results that without the optimizations scheme S_{n+d} is less likely to benefit from increasing the RB size, since it becomes more encumbered with invalidations.

4.7.1.7 Set Associative RB

Until now we have used fully associative RBs in our experiments. In this section, we

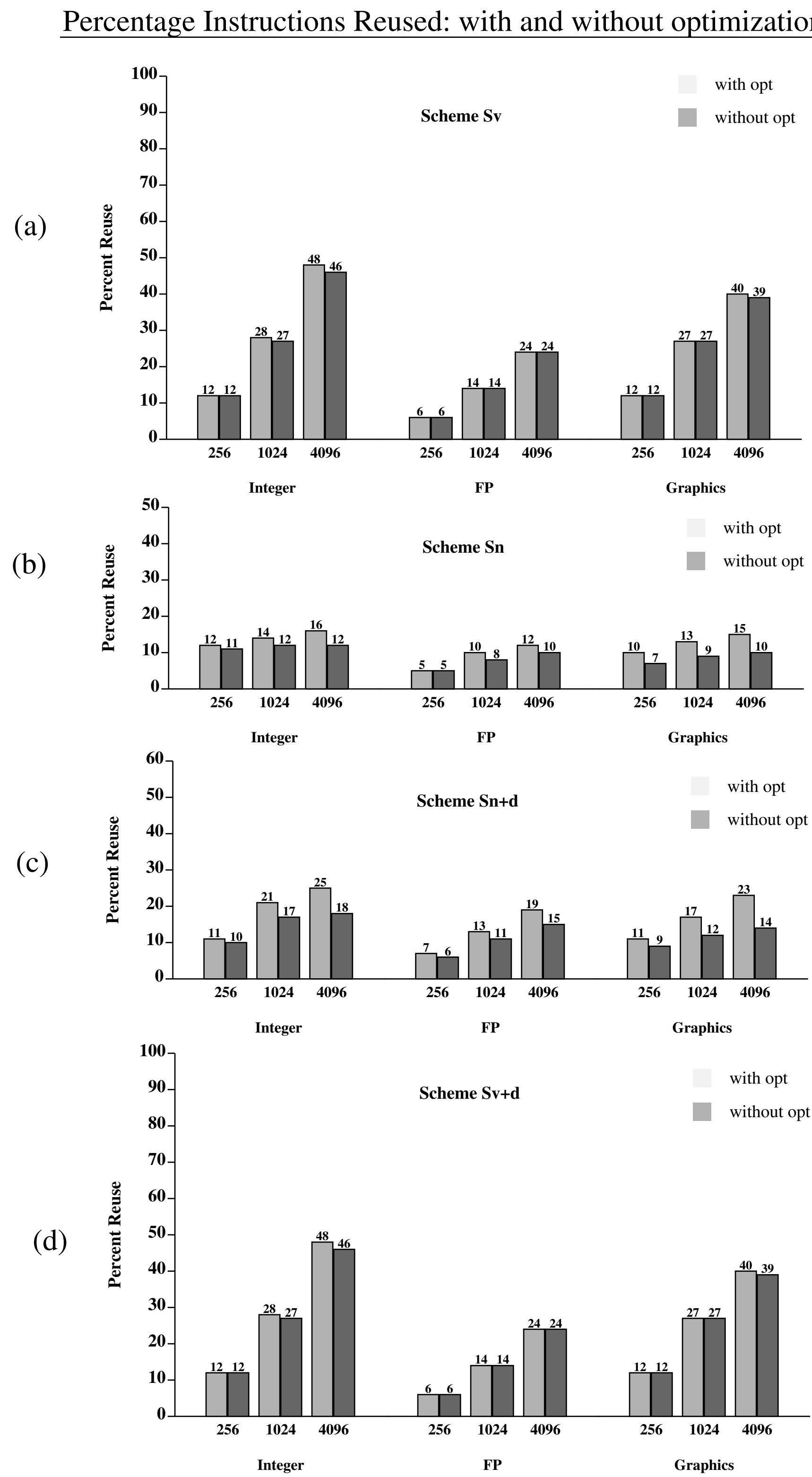


Figure 4.15 Mean percentage reuse rates with and without the various reuse optimizations (discussed in Section 4.5.1) for different reuse schemes and RB sizes.

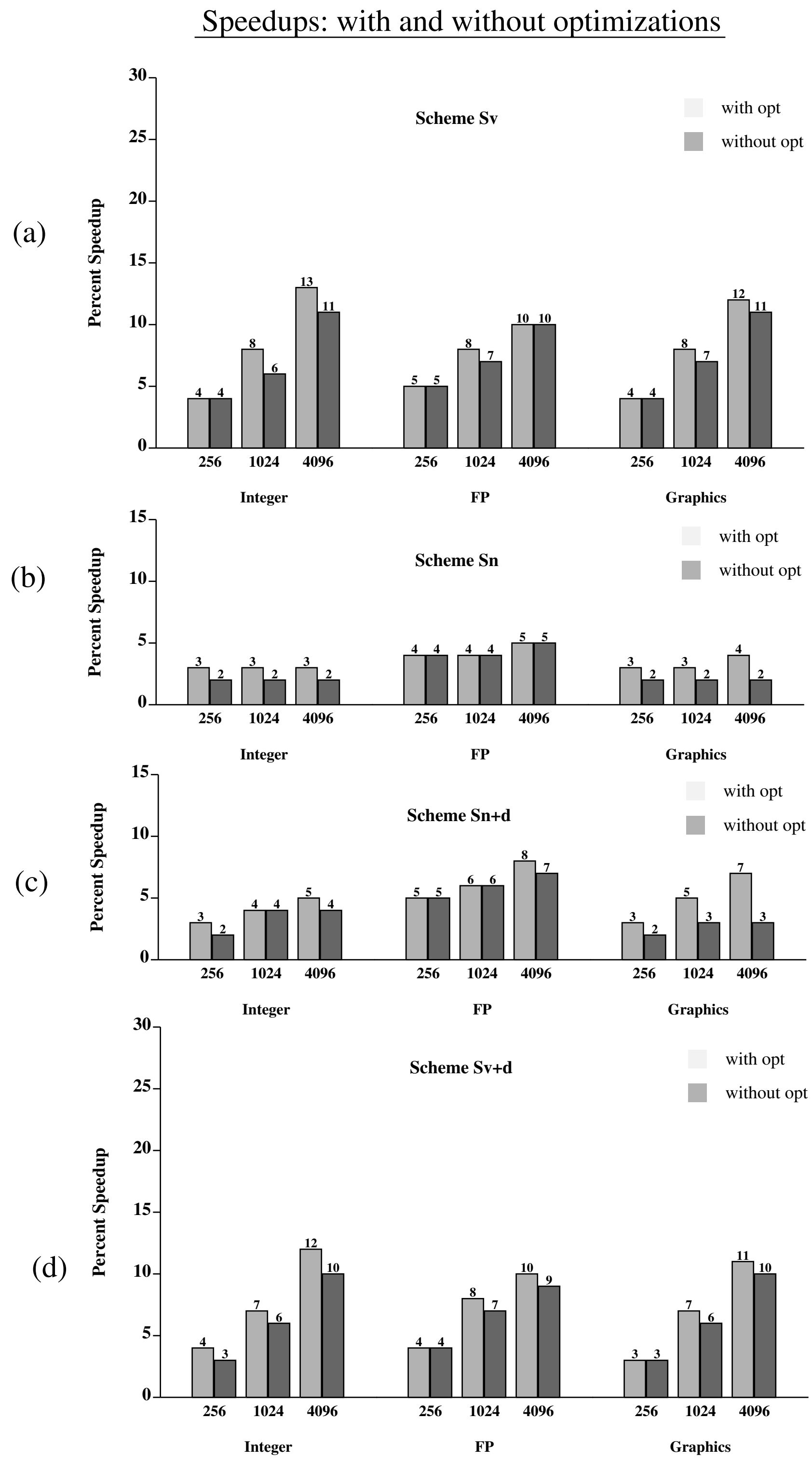


Figure 4.16 Mean speedups with an without various reuse optimizations for different reuse schemes and RB sizes.

briefly show how a set-associative RB compares with a fully-associative RB. A more thorough evaluation of RB associativity and RB size) is conducted in Chapter 5.

Apart from eliminating the conflict misses between different instructions, a fully-associative RB allows buffering several instances per instruction, and hence permits the reuse of instructions that produce several different instances. A set-associative RB, on the other hand,

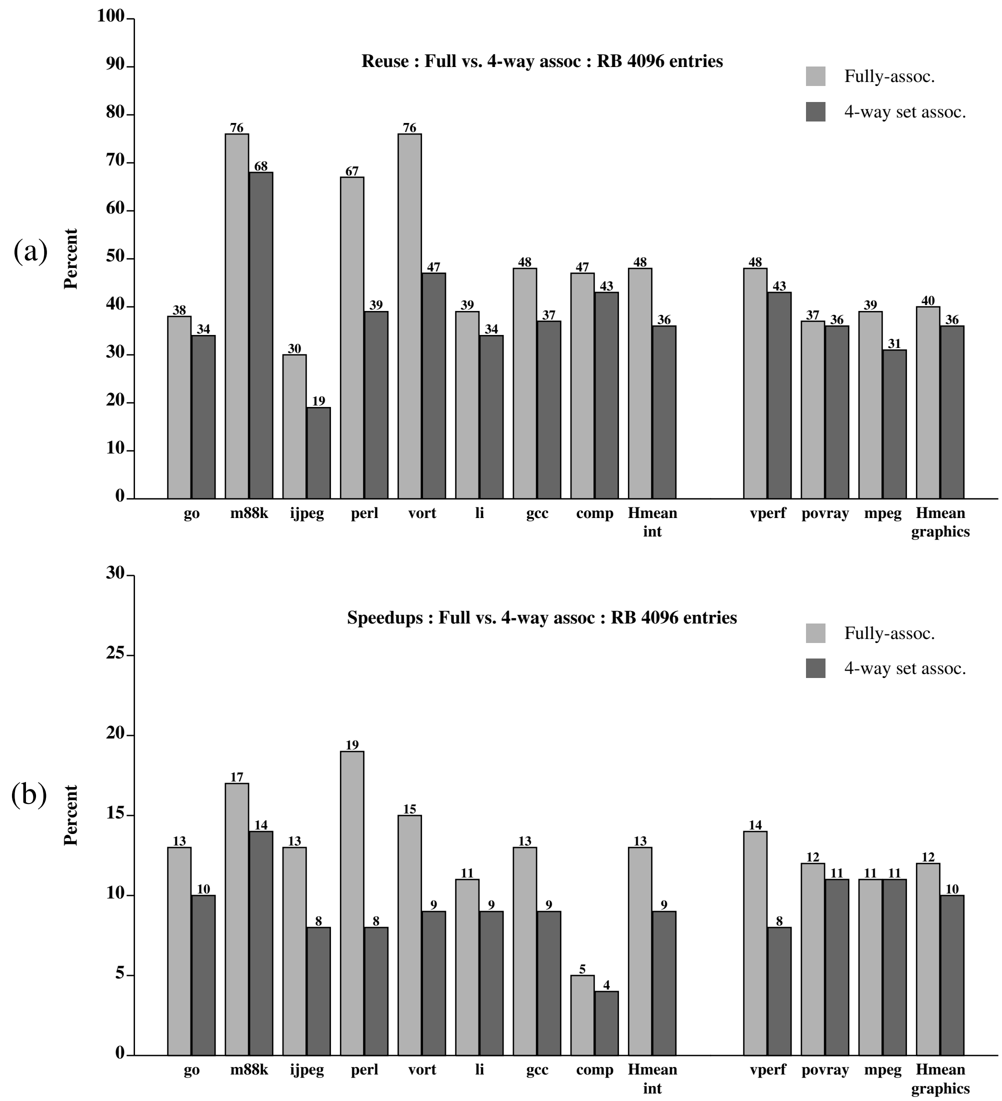


Figure 4.17 Comparison of 4-way set associative RB against fully-associative RB. (a) Percentage instruction reused, (b) Speedups. The results are for RB with 4096 entries using Scheme S_v.

can only buffer as many instances per instructions as its set size (assuming the only PC is used for indexing the RB). Hence, a set-associative RB may be unable to capture the reuse of instructions that produce a large number of different instances.

Keeping this explanation in mind, we see the results in Figure 4.17. In this figure, we present the (a) reuse rates and the (b) speedups for a 4-way set-associative and a fully associative RB with 4096 entries. The results are shown for integer and graphics benchmarks and are obtained using the scheme S_v. We observe that for most benchmarks the reuse rates and the resultant speedups for 4-way associative RBs are comparable to the fully-associative ones (e.g., for *go*, *viewperf*, *povray*, *compress*). But for benchmarks, like *perl*, *vortex*, *ijpeg* and *gcc*, fully associative RBs perform far better than 4-way associative RBs. The reason for this difference is, as explained above, due to the ability of the fully-associative RB to retain many different instances per instruction. We will look at the size and associativity requirements more thoroughly in Chapter 5.

4.8 Related Work and Discussion

Harbison in [23, 22] proposes a stack-oriented architecture, the *Tree Machine*, which uses a hardware mechanism, the *value cache*, for eliminating common sub-expressions and loop invariant expressions. He keeps the result of a computation (called a *phrase*) in the *value cache*. A bit vector, called a *dependency set*, is associated with each result in the value cache to indicate the variables used in computing the result; the bit positions are determined by the address of the variables. When an address is overwritten, all the results in the value cache which have the bit set for that address are invalidated. If a phrase is encountered again, recomputation is avoided by reading the result from the value cache. This approach is similar to our second reuse scheme, scheme S_n. Both perform reuse based on the architectural names of the operands (scheme S_n uses the register specifier, while the value cache uses the memory address). The differences are highlighted later in this section.

Richardson [36, 37] introduces the notion of redundant computation, which is computation that produces the same result repeatedly because it gets the same value for its operands. In this work, the results of floating point operations are stored in a cache, called the *result cache*. The index of the cache is obtained by hashing the operand values. The result cache is accessed in parallel with executing an floating point operation. If the result is found in the result cache then the operation is halted.

Oberman and Flynn [34], propose the use of *division caches* and *reciprocal caches* for capturing the redundancy in the division and square root computation. The division caches are similar to Richardson's result cache, but for divisions only. The reciprocal caches hold the reciprocals of the divisors. They help convert the high latency division operation to a relatively low latency multiply operation. These caches are accessed using the bits from the mantissa of the operands.

There are several differences between our work and the work mentioned above. First, the above techniques are more special purpose. The value cache [22, 23] approach is tailored for an architecture which expresses computation in the form of *parse trees* (Tree Machine). The result caches [36], and the division and reciprocal caches [34] target only floating point operations. Our approach is general purpose in that it does not assume any special architecture, and it captures reuse of any type of instruction (except stores). Second, the techniques referred to above access their respective result buffers (value cache in [23], result cache in [36] and division and reciprocal caches in [34]) by using either the operand address [23] or operand values [36, 34], which are only available later in the pipeline. Thus, the result buffer access is delayed until the execute stage, which restricts the usefulness of these techniques only to instructions which have multi-cycle latency ([36] uses it for floating point instruction, while [34] uses it for floating point divides only). In contrast, the reuse schemes presented in this thesis access the RB using the instruction address, and hence reuse occurs while the instruction is still in the decode stage. This has two advantages: first, even single cycle instructions benefit from reuse; second, the reused instruction need not flow down the pipeline, which frees machine resources for other instructions to use. The third difference is, since other techniques use operand values for indexing in the result buffer, unlike our schemes, they cannot reuse multiple dependent instructions simultaneously (the result of one instruction would be needed to form the index for the dependent instruction)

One of the benefits of instruction reuse is that it collapses true dependencies. Other techniques based on value prediction have been proposed to achieve the same effect [27, 26]. The fundamental difference from our schemes is that these approaches are speculative. The instructions still must execute to generate result for later verification. Our schemes are non-speculative, and the reused result is guaranteed to be correct. For a more elaborate comparison of the similarities and differences of these two techniques, the readers are referred to [45].

4.9 Summary and Conclusions

In this chapter, we introduced and studied the concept of dynamic instruction reuse. We presented four schemes for exploiting the phenomenon. All four schemes buffer the outcome of an instruction in a *reuse buffer* from where future instructions can access it (if the operands match). The schemes differ in the way that they track the reuse status of an instruction: scheme S_v uses operand values, scheme S_n uses operand names, scheme S_{n+d} uses operand names as well as dependence information, and scheme S_{v+d} uses operand values and the dependence information. By dynamically reusing instruction results, we are able to (i) cut down on the resources required to execute the instructions, and (ii) cut down on the time that it takes to know the outcomes of sequences of dependent instructions, *i.e.*, reduce the length of critical paths of computation.

We evaluated the effectiveness of the proposed schemes using 3 different buffer sizes: 256, 1024, and 4096 entries. Significant instruction reuse was found in many cases (e.g., 76% for *vortex* and *m88ksim*, 48% for *gcc*, *viewperf* and *tomcatv*), with as many as 76% instructions reused in two cases. Comparing the four schemes, we see that scheme S_v, being the most aggressive scheme, reused the most instructions (average reuse over SPEC ‘95 integer, SPEC ‘95 floating point, and graphics benchmarks were 48%, 24%, and 40%, respectively, for the 4096-entry RB). Scheme S_n, being the most conservative scheme, reused the least number of instructions (average reuse rates for integer, floating point and graphics benchmarks were 16%, 15%, and 12%, respectively, for 4096-entry RB). Scheme S_{n+d} improved upon scheme S_n by allowing the reuse of the dependent instructions, and consequently performed better than S_n (average reuse for the three benchmark sets were 25%, 19%, and 23%, respectively). Finally, scheme S_{v+d} augmented scheme S_v by reusing dependent instructions in the same cycle only through the dependence information (and not through values). This scheme not only facilitated the reuse of dependent instructions in a value-based scheme, but it also attained the reuse rates close to scheme S_v (averages for the three benchmarks sets were 45%, 23%, and 36%, respectively).

We presented other reuse characteristics, such as, the reusability of different instruction types and the contribution of each instruction type to total reuse. These results showed that most of the instruction categories were amenable to reuse; a significant number of instructions were reused from all the broad categories of instructions considered (e.g., 31% of loads were reused, 44% of 2-register operand integer instructions were reused). We saw that although most instruction categories are amenable to reuse, loads and address calculation contribute the most to the overall reuse (nearly 50% or more of total reuse came from these two categories).

We also measured the resulting speedup in the program execution time. We saw that speedups follow the trend in the reuse rates, *i.e.*, an increase in reuse rates almost always engenders a corresponding increase in speedups over base case. However, in absolute terms the speedups obtained were small in many cases, specially for scheme S_n and S_{n+d} (they are less than 5% in many cases). However, for schemes S_v and S_{v+d} we saw significant improvement in performance with more than 10% speedups in many cases.

Chapter 5

Reuse Buffer Characterization and Management

As may be obvious from the previous chapter, the Reuse Buffer (RB) is the hardware structure central to the instruction reuse (IR) technique. It provides space for preserving the state of instructions and, depending on the reuse scheme, has mechanisms for ensuring the consistency of these instructions when the state of the machine changes.

The RB is a complex structure with its *CAM* (content addressable memory) [49] search logic for selectively invalidating instructions and multiple ports. We would like to keep this structure small so as to keep it implementable. Yet we would also want to achieve high reuse rates. We can cater to both requirements by having a small RB and *managing* it efficiently — *i.e.*, by judiciously deciding which instructions get to reside in the RB so that the reuse rate is maximized. In this chapter, we study the topic of RB management. We present four ways for managing the RB efficiently to improve reuse rate. Three of these policies are enhancements to existing policies (such as LRU) and the fourth is a novel management policy that attempts to tackle the problem of buffer management at a more fundamental level.

Before exploring the management policies for the RB, we attempt to better understand the behavior of the RB itself. For this purpose, we characterize the RB with respect to its three main parameters — size, associativity, and current management policies — showing how the reuse rates change when each of these parameters are varied. We also present the *limit* reuse rates for different RB sizes and associativities to ascertain the best reuse rate we can hope to achieve for different RB configurations. Aside from characterizing the RB, this study — especially the comparison between the limit and the real reuse rates — exposes the inefficient use of the RB, and hence sets the stage for the RB management studies which follow.

This chapter is laid out as follows. In the next section, we present the experimental setup specific to this chapter. Thereafter, in Sections 5.2 and 5.3, we characterize, respectively, the size and associativity of an RB. In Section 5.4, we describe four new RB management policies, presenting the rationale behind using them and discussing their advantages and disadvantages. In Section 5.5, we experimentally evaluate these policies. And, finally, we summarize this chapter and provide conclusions in Section 5.6.

5.1 Experimental Setup

The experiments in this chapter are performed using the timing simulator described in Chapter 2. The processor model used is the same as the one described in Section 4.4, with one exception: in these experiments, we insert instructions in the RB at the commit stage of the pipeline. We do so to study the RB characteristics arising because of the actual program rather than because of speculation. The instruction reuse is implemented using scheme S_v, the most aggressive reuse scheme.

To obtain the maximum reuse rates achievable by different RB sizes, we use a RB management policy similar to the Belady's optimal management policy [7]. In this policy, we use the oracle information about when each instruction is going to be reused in the future to decide which instructions to keep in the RB. However, there is a slight difference between our and Belady's algorithm: the original Belady's algorithm only controlled the replacements from a storage, whereas our algorithm controls both storage replacements and insertions. More specifically, the Belady's algorithm replaces those items from a storage which will be needed farthest in the future, but it always inserts the incoming item in the storage, even if the incoming item will be needed further away in the future than the item that it replaces. In our algorithm, we not only evict those items from the storage which will be needed farthest in the future (like Belady), but we also do not insert an incoming item if it is going to be needed further away in the future than the item that it will replace. Although, intuitively, it appears that our algorithm should be optimal (and the experiments show that it performs better than the original Belady's algorithm), we do not yet have a formal proof for it's optimality. Hence, in spite of our conjecture that this algorithm is optimal, we restrain ourselves from naming it as such in this thesis; instead, we call it a *limit policy.* However, the reuse rates from this policy are taken to be the upper bounds on the reuse rates achievable by different RB sizes.[1]

To keep the simulation requirements manageable, we only use the SPEC'95 integer programs for the studies in this chapter.

1. Even if these reuse rates were not true optimal, they will still be quite high, and since it is inconceivable that these high reuse rates can be attained by any practical policy, they will still serve as a useful upper bound.

5.2 Characterizing RB: Size

In this section, we determine how the reuse rates — *i.e.*, percentages of dynamic instructions reused — vary with the RB size. The results of this characterization are presented in Figure 5.1. The reuse rates are obtained for 4-way associative RBs ranging in size from 256- to 64k-entries (sizes are shown in $\log_2$ of number of RB entries). We show results for three commonly used replacement policies — LRU, FIFO, and Random — and for the limit policy (labelled as *limit:4way*). On this graph, we also include the reuse rates for full-associative RB managed using the limit policy (*limit:full*). This curve is meant to provide the absolute upper bound on the reuse rate for every RB size.

Next, we present the different ways we can interpret the graphs in Figure 5.1 and also present several observation we can make from it.

- First, for most benchmarks, we see that the reuse rate increases steadily with the size of the RB. This result confirms what would be expected given that there is a significant amount of repetition in programs.
- In Chapter 3, we had seen what fraction of a program gets repeated (Table 3.1). Analogously, in Figure 5.1, we show what fraction of a program gets reused. This result is provided by the highest point on the *limit:full* curve, which shows the maximum number of instructions that can be reused in programs. Comparing this result with the results in Table 3.1, we see that most of the repetition present in programs can be reused; the non-speculative (and, hence, conservative) nature of IR does not fundamentally limit the amount of repetition it can capture.
- Further the *limit:full* curve also indicates the minimum number of RB entries required to

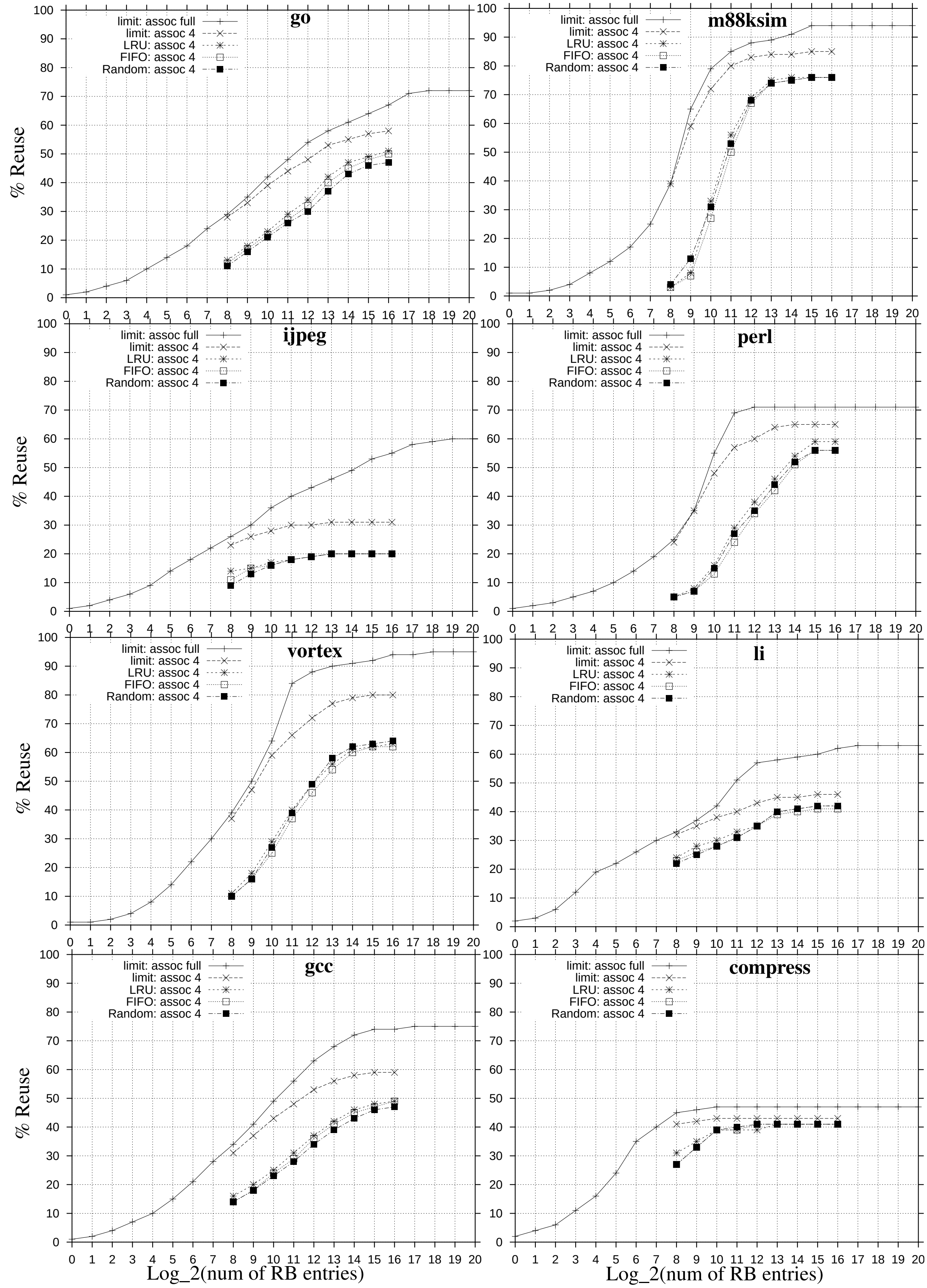


Figure 5.1 Limit and actual (Lru, Fifo, and Random) reuse rates for different RB sizes.

attain a certain level of the reuse rate. For example, to attain a reuse rate of 50% in *vortex* we need an RB with at least 512 entries. We also note from the *limit:full* curve that a significant amount of reuse can be captured with small number of RB entries —e.g., more than 50% of dynamic instructions can be reused for most benchmarks with less than 2K RB entries.

- For most benchmarks (except *ijpeg* and *li)*, the *limit:4way* curve closely follows the *limit:full* curve, suggesting that an associativity of 4 may be sufficient to attain the reuse rates close to the absolute limit. However, for *ijpeg* and *li*, higher associativity may be required. (Associativity is more thoroughly investigated in the next section.)
- We see that there is a significant gap between the limit:4way and the LRU:4way curves. For several benchmarks (e.g., *perl*, *go*, *gcc*), the limit policy achieves the same level of reuse as the LRU with an RB nearly 8 times smaller. The RB size needed by the limit policy is at least 2 times smaller than the LRU for the same level of reuse in all cases. This gap indicates that there is significant room for improving the reuse rate of an RB — or, alternatively, reducing the size of an RB keeping the same reuse rate — by better management of RB space.
- Finally, we note that the reuse rates obtained by LRU, FIFO, and Random replacement policies are comparable, and that FIFO and Random policies are unable to bridge the gap between the limit and LRU policies. The inability of these commonly used policies to solve the problem (we discuss the reasons later in Section 5.4) provides further motivation for investigating better RB management policies.

5.3 Characterizing RB: Associativity

Before we move on to the topic of RB management, we also characterize the RB with respect to its associativity. In an RB, associativity helps in two ways: (i) by allowing multiple instructions to reside in the same set, hence reducing the conflict misses (as in caches), and (ii) by permitting the storage of multiple instances of instructions in the RB, thus enabling the reuse of instructions whose repetitions are interspersed with their other (non-matching) instances (this is unique to IR). In this section, we first determine how interspersed the reuse is in general and thereby ascertain an upper bound on the amount of reuse that can be captured by an RB of a certain degree of associativity. After that, we present the overall effect of RB associativity on the reuse rates.

5.3.1 Effect of storing multiple instances in RB on reuse rates

As mentioned above, an instruction may get repeated after several other of its instances have been encountered. If every dynamic instruction is inserted in the RB, the reuse of such an instruction is possible only when the RB associativity is large enough to store the instruction and all its intermediate instances. The amount of reuse, for example, that a 4-way associative RB can capture is limited by the amount of reuse that exists at a distance of up to 4 instances away. That is, a reuse that occurs after 3 instances *may* be captured by a 4-way RB, while the one that occurs after 7 instances *will* not be. In this section, we try to get a feel for the upper bound on the amount of reuse for different RB associativity — or, alternatively, the least degree of associativity needed to capture a certain amount of reuse.

In Table 5.1, we show what percent of *total reuse* is present at which *reuse distance*. The *total reuse* is defined as the reuse that can be captured using an infinite size (very large) RB (this total reuse is shown by the highest point on the *limit:full* curve in Figure 5.1). The *reuse distance* is defined as the number of instances after which an instruction gets reused — for example, if an instruction is reused after 7 of its other (non-matching) instances are encountered, then the reuse distance is considered to be 8. The greater the reuse distance, the more instances need to be saved to capture the reuse, and, hence, the higher is the required associativity. In the table, we show the reuse distances from 1 to 4096 (at increments of a power of 2) and the fraction of total reuse that exists within that distance. From the table we can make the

Reuse Distances	**% of Total Reuse in Programs**							
	go	**m88k**	**ijpeg**	**perl**	**vort**	**li**	**gcc**	**comp**
1	57	62	35	56	58	49	51	73
2	63	74	37	66	64	55	59	76
4	71	85	41	86	73	61	69	84
8	77	88	50	90	83	70	75	92
16	82	90	52	95	91	78	81	94
32	87	92	61	98	94	84	85	95
64	90	93	69	99	95	86	89	96
128	92	95	74	99	96	87	91	96
256	94	95	78	99	97	88	95	97
512	97	100	86	100	98	88	98	98
1024	98	100	95	100	98	93	99	99
2048	99	100	98	100	99	98	100	100
4096	99	100	99	100	99	99	100	100

Table 5.1 Percentage of total reuse in programs at different reuse distances (see text for definition). The reuse distances are in terms of the number of instances of the same instruction. Thus, 77% of reuse at the distance of 8 in *go* means that 77% of total reuse present in the program is encountered within 8 instances of the same instruction.

following observations:

- For all benchmarks, except *ijpeg* and *li*, more than 50% of total reuse present in programs exists at a distance of 1 — meaning, we can capture more than 50% of available reuse by only buffering the last instance of instructions, *i.e.*, 1-way associativity is sufficient. In general, more than 70% of total reuse exists within the distance of 4, meaning we only need to buffer the last four instances to capture this fraction of reuse. Thus, we note that for most programs we don't need to look beyond a large number of instances for reuse; the majority of reuse exists "near-by". Thus, a high associativity may not be essential for capturing high amounts of reuse.

- For, *ijpeg* and *li*, the reuse distances are longer. For example, to capture 62% of total reuse in *ijpeg* we need to be able capture reuse that occurs up to the distance of 32. Thus, to capture large amounts of reuse, many instances need to be saved for these benchmarks, requiring a highly associative RB. In Section 5.2, we had stated for *ijpeg* and *li*, based on the divergence between the *limit:4way* and *limit:full* curves, that these programs may require a higher associativity; the results here corroborate this statement.

5.3.2 Overall effect of associativity on reuse rates

In this section, we show how the reuse rates vary with associativity. In Figure 5.2, we show the reuse rates for RBs with associativities 1, 2, 4, and 8, and sizes varying from 256- to 64k-entries. The numbers are presented for LRU and limit management policies. We can make the following observations from this figure:

- The associativity becomes important mainly for large buffers. For small buffers, the reuse

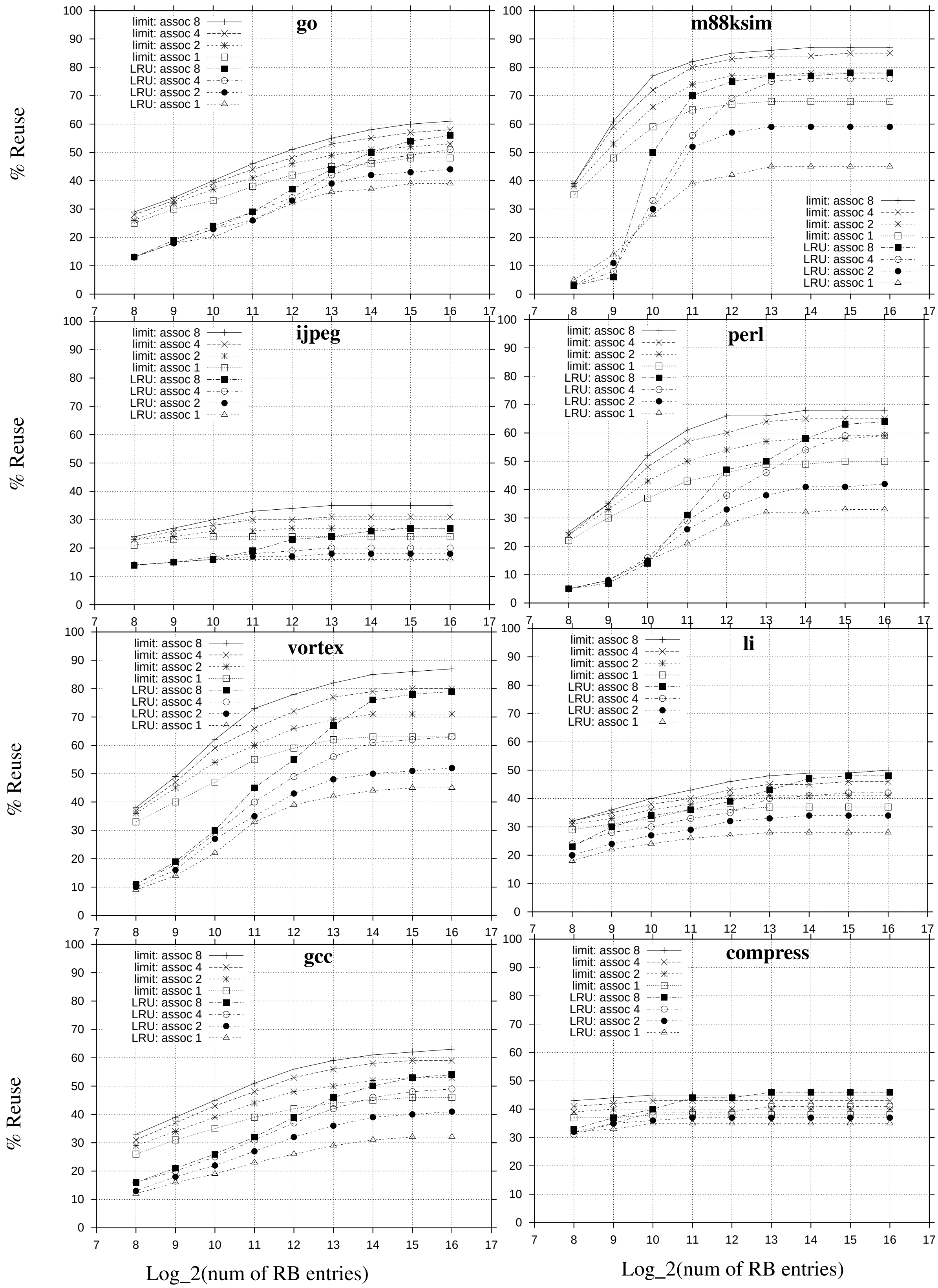


Figure 5.2 Limit and LRU reuse rates for RB assoc. 1, 2, 4, and 8 for different RB sizes.

rates are comparable for all four associativities. The advantages of increasing the associativity are small for small buffers because, generally, the accesses are uniformly distributed over all entries. Thus, increasing the associativity does not reduce the contention. In fact, it may sometimes increase contention — e.g., when a set merges with another set with a high amount of contention after the associativity is increased — causing a degradation in the reuse rate (e.g., in *m88ksim*, for a 512-entry RB).

- Comparing the limit and LRU curves, we notice that higher associativity reduces the gap between the two management policies — e.g., the gap between 8-way limit and LRU curves is, in general, smaller than that between the 4-way curves. However, there is still a significant gap between the two management policies, especially for small RB sizes (e.g., for *m88ksim*, *perl*, *vortex*, and *go*). Thus, increasing the associativity by itself does not solve the problem of less than optimal buffer usage; we need to develop better management policies for that purpose.

5.4 RB Management

Next, we discuss RB management, by which we mean deciding how the space in the RB is used, *i.e.*, which instructions get to reside in the RB. Until now, we have only used regular replacement policies, such as LRU, FIFO, and Random, for managing the RB, without using any special enhancements for improving the reuse rates. As we can see from Figure 5.1, these regular policies perform far below the limit level, and that we need better ways of managing the RB to utilize it more efficiently.

Intuitively, the reason for the sub-limit performance of the regular policies is that often they evict reusable instructions, instead of unreusable instructions, from the RB. Thus, if we are to develop a new management policy to improve RB utilization then this new policy should prevent such non-optimal evictions from taking place. There are two broad approaches: (i) by controlling insertion, *i.e.*, by reducing the traffic into the RB (only inserting instructions that are likely to be reused) and hence reducing the chances of (reusable) instructions in the RB getting evicted; and (ii) by controlling replacement, *i.e.*, by not evicting instructions that are likely to be reused. However, there can be many ways of implementing these two approaches. In this thesis, our purpose is not to perform an exhaustive study of various buffer management strategies. Instead, we wish to focus on and understand the main causes of poor buffer management; therefore, we develop and study a small group of policies that attack what we believe are the main reasons for sub-limit RB usage. By learning about why these policies work (or don't work), we can gain insight into how to design more effective management policies for the RB.

We study four management policies. Two of these are insertion policies, *i.e.*, they identify instructions that are not likely to get reused and don't insert them in RB. The third is a replacement policy, which identifies instructions that are likely to get reused and does not evict them from the RB. The fourth — unlike the other three, which are simple enhancements to the regular management policies — is a novel management policy. It performs both selective insertion and selective replacement, and is designed along the lines of the limit policy. This policy determines the expected distance to reuse for instructions and uses this distance to schedule them in the RB, always giving priority to instructions whose reuse is nearer (like the limit policy).

Next, we describe these policies along with the rationale for using them.

5.4.1 Insertion Policy

5.4.1.1 Filter instructions that do not get reused (FnReused)

As mentioned above, the regular policies perform below limit level because they evict reusable instruction from the RB when they should not have. We have seen that many times these reusable instructions are evicted by instructions that themselves do not get reused. We can prevent such evictions from taking place if we can filter out the unreusable instructions from the insertion stream.

One way of detecting the unreusable instructions is to track their history: if they were not reused in the past they are unlikely to get reused in the future. In the *FnReused (Filter not Reused)* policy, we use such a method to detect unreusable instructions.

In this policy, we employ a table of counters indexed by the PC. Every time an instruction is evicted from the RB without being reused, the counter for this instruction in this table is incremented. If the counter reaches a pre-defined threshold value, then a bit is set in that table entry indicating that the instruction is "unreusable". If, however, the instruction gets reused before the threshold value is reached then the counter is reset back to 0. This table is consulted at the time instructions are inserted in the RB; if an instruction is found to be tagged "unreusable" then it is not inserted in the RB. Note that the counter table is separate from the RB and that its each entry is much smaller than an RB entry. Hence, it can potentially buffer state for a much larger number of instructions than the RB.

Although we have described the policy using a separate table (and we also evaluate it this way later), another place for storing the counters and the reusable/unreusable bits can be the I-cache. The counters and the bits can be maintained per instruction (or per couple of instructions) in the I-cache. The advantage of this design is that we do not have to manage another table structure; the counters and the bits are managed as part of I-cache itself. The disadvantage is that it make the I-cache design more complex.

The threshold value needs to be chosen so that it not only minimizes the number of times potentially reusable instructions are tagged as "unreusable" (which may happen when the threshold value is too small), but it also minimizes the number of times the actually unreusable instructions escape untagged (which may happen when the threshold value too high). The threshold value may also be dictated by the hardware cost of implementing the counters since the counter width is determined by it.

Finally, one issue with this policy is that it does not have a self-correction mechanism — *i.e.*, once it tags an instruction as unreusable, it cannot reset the tag if the instruction becomes reusable (e.g., due to a different program phase). We will see the impact of this limitation in the evaluation section.

5.4.1.2 Filter instructions that are "Not-Ready" (FnReady)

One of the reasons why some instructions don't get reused is that their operands are not available (*i.e.*, ready)[2] *at the time they are checked for reuse*. In this policy, called *FnReady (Filter not Ready)*, we filter out such instructions from the RB insertion stream (and, hence, prevent the possible eviction of reusable instructions from the RB). In terms of coverage, this policy is a subset of the FnReused policy since it only filters one "type" of unreusable instruc-

2. Though actual values are required for the reuse test only in the case of scheme S_v (and, for independent instruction in scheme S_{v+d}), the information whether instruction operands are ready or not is used for determining reuse by other reuse schemes as well; in schemes S_n and S_{n+d} also, instructions do not get reused if their operands are not ready. Thus using readiness of operand values for categorizing reusable and unreusable instructions is not something specific to scheme S_v only.

tions. However, it may be significantly cheaper to implement since, as we describe next, it does not need a table for storing its meta-state. Also, unlike the FnReused policy, this policy is self-correcting, as we will explain later. In the ensuing discussion, we refer to an instruction that does not have its operands ready at the time of reuse as a *not-ready* instruction, and the one that has as a *ready* instruction.

Conceivably, we can implement this policy along the lines of the FnReused policy: that is, track the history of instruction ready/not-ready information in a table; tag instructions that are repeatedly not-ready; and prevent insertion of tagged instructions in the RB. But, it is possible to implement the FnReady policy in a simpler way, without requiring a separate table. The operand ready/not-ready behavior is reasonably stable from one incarnation of an instruction to the next; if an instruction is not-ready in its current incarnation, it will likely be not-ready in the next one. So we do not need to accumulate counters to detect the not-ready instructions. Based on whether an instruction is ready or not-ready in the current incarnation, we can insert or not-insert the instruction in the RB.

This policy is implemented as follows. With each instruction in the instruction window we associate a “ready-at-reuse” bit. We set this bit on two conditions: (i) if the instruction has its operands ready at Register Read stage (the stage in which the reuse test is done), or (ii) if the instruction’s source instructions, ahead in the pipeline have their “ready-at-reuse” bit set (the reason for this condition is explained below). Only instructions with their “ready-at-reuse” bit set are inserted in the RB. As mentioned above, apart from not requiring a separate table, this policy has another advantage over FnReused: it has a natural self-correction mechanism. If a not-ready instruction starts having its operands ready, it will automatically start getting inserted in the RB. This advantage over FnReused arises because, unlike reused/not-reused information, the generation of ready/not-ready information does not depend on whether or not the instruction is in the RB.

The second condition for setting the “ready-at-reuse” bit mentioned above, needs explaning. This condition is used to avoid obstructing the reuse of a dependent sequence of instructions. For most dependent instructions, the operand values become ready because their source instructions are reused (otherwise, the operand value may not be ready at the register-read stage). Without the second condition, a dependent chain of instructions of length ‘n’ will have to get encountered ‘n’ times to get fully inserted in the RB (since it will get inserted one instruction per encounter). This might severely hamper the reuse of such chains. To avoid this, we check in the first pass itself if the source instructions will get entered in the RB (because their “ready-at-reuse” bits are set), and if so we enter the dependent instruction as well (set its “ready-at-reuse” bit).

5.4.2 Replacement Policy

5.4.2.1 Retain Reused Instructions (RR)

Until now we have used a standard LRU policy to do the replacement from the RB. This policy does not consider whether the victim instruction is likely to be reused or not, and hence it often ends up evicting a reusable instruction even when there are unreusable instructions in the RB. Reuse rate can be improved if such evictions can be prevented. In this RR (retain reused) policy, we attempt to do this. We mark instructions in the RB as likely- reusable and likely-unreusable. The instructions marked likely-reusable are then given a privileged status in the RB: they are selected for eviction only when there are no likely-unreusable instructions available for replacement. Although several heuristics are possible for selecting likely-reusable instructions, we employ a simple one in this policy. We consider an instruction to be likely-reusable if it has gotten reused in the past. This is based on our finding that instructions that get reused often do so multiple times.[3] We describe the implementation of this policy next.

With each entry in the RB we attach a small counter. When an instruction is reused, the counter in its entry is set to a pre-defined value, indicating that the instruction is likely-reusable. The replacement algorithm selects the likely-unreusable instructions (those with a zero counter value) for eviction before the reusable instructions. The counter value of a likely-reusable instruction is decremented every time it is picked for replacement, but is not replaced. When the counter value becomes zero, the instruction loses its privileged status and, thereafter, it is chosen for replacement as usual. If, however, the instruction is reused again, its counter value is reset to the pre-defined value. The purpose of using the counter rather than a static tag for marking the instructions as likely-reusable is that it limits the duration of the privileged status and, hence, prevents a reusable instruction from residing in the RB forever (even when it is no longer reusable).

When a likely-reusable instruction is evicted, we have two options: (i) either we can save the information that it is likely-reusable in some another table and initialize the counter appropriately when the instruction is inserted in the RB the next time; or (ii) we do not save any information on eviction, and on the next insert the instructions starts like an ordinary instruction (with counter value 0). While the first option may be more profitable (since it will be able to retain more reusable instructions), the second will entail less hardware cost since it does not require an extra table. In this thesis, we evaluate the second option.

3. We have shown in Table 3.2 that unique repeatable instances get repeated many times on average.

5.4.3 FiF: Farthest in Future Replacement Policy

All policies presented so far are enhancements to existing management policies, such as LRU. Also, they are somewhat ad-hoc in nature, attacking specific aspects of the buffer management problem rather than the whole problem in general. In this section, we look at a new management policy that attempts to solve the RB management problem at a more fundamental level.

Before presenting the new policy, let us see what needs to be done to manage the RB efficiently. The key lies in the criterion for selecting which instructions are inserted in the RB and which are replaced from the RB. Intuitively, the criterion that will result in the best RB utilization is the *likelihood of reuse* — *i.e.*, how likely an instruction is to get reused. The reason being, that's the only piece of information that precisely quantifies the importance of keeping an instruction in RB. Also, asymptotically — with 100% accurate likelihood information — a policy based on likelihood of reuse will perform exactly like the limit policy. This assures us that this line of approach is the right one, in the sense that, it (unlike other policies) has the potential to lead us all the way to the best RB utilization. With this approach, we also know the knob that needs to be tuned for improving performance, namely the likelihood to reuse. More accurate we can make the estimate of likelihood of reuse, the better will this policy be able to utilize the RB. Under this criterion, the RB will be managed as follows: at the time of replacement, an instruction that is least likely to get reused will be evicted; at the time of insertion, an instruction will be inserted in the RB only if it is more likely to get reused than the instruction it will replace.

But, how do we ascertain the likelihood of reuse for instructions? There may be several ways. We describe one such way next, along with the rationale behind it. After that, we describe the *FiF (Farthest in Future)* policy, where we show how this information can be used to perform buffer management.

5.4.3.1 Obtaining "likelihood of reuse" information

We know from the results of Chapter 3 that most instructions in programs get repeated (more than 75% of dynamic instructions get repeated in many cases). One of the main reasons why we cannot reuse all these instructions is because we are unable to retain them in the RB until the time they are needed for reuse. Said another way, these instructions do not get reused because they get evicted from the RB. Thus, we can calculate the likelihood of reuse in terms of the likelihood of eviction — *i.e.*, we can say that an instruction is more likely to get reused if it is less likely to get evicted from the RB, and vice versa. We describe how we calculate the likelihood of eviction next.

The likelihood that an instruction will get evicted before reuse depends on how many other instructions map to the same RB set during the time the instruction is resident in the RB. For example, suppose that an instruction gets inserted in the RB at time t1 and will get reused at time t2. To get reused it needs to reside in RB until time t2. The likelihood that it will get replaced before t2 depends on how many other instructions contend for space in the same RB set between time t1 and t2. If the number of "collisions" — which we define as the number of instructions mapping to the same RB set — during the time interval is small, then the likelihood that the instruction will be evicted will be small. Conversely, if there are many collisions in the time interval, then the likelihood of eviction will be large. In other words, an instruction is more likely to get evicted if it encounters more collisions from other instructions during the time period it needs to be in the RB to get reused.

To summarize, we ascertain the likelihood of reuse for an instruction by estimating the number of collisions it will see during the time period it needs to be present in the RB for getting reused: the more the number of collisions, the more likely the instructions will get evicted, and hence the less likely that they will get reused.

5.4.3.2 Description of the policy

The FiF policy uses the likelihood of reuse information, calculated in terms of the number of collisions (as mentioned above), to perform buffer management. The number of collisions an instruction is likely to experience before it gets reused can also be interpreted as its *distance* to reuse — the greater the number of collisions, the “further away” is the reuse, and vice versa. It is this interpretation that gives the FiF policy its name. Based on this interpretation, we can describe the working of the FiF policy as follows. We try to keep the instructions which have the shortest distances in the RB: choosing the instructions with the largest distance to reuse for replacements, and inserting new instructions in the RB only if their distance to reuse is smaller than those of the instructions they will replace (how distance is maintained in RB is described shortly).

Next, we describe the various hardware structures needed to implement this policy. Then we describe how we calculate the distances. Lastly, we describe the policy operation in detail.

Hardware structures: We need the following hardware structures for the FiF policy:

- A *collision counter* per RB set, which counts the number of collisions to the set.
- A PC-indexed *Distance Table (DT)*, which is used for calculating and storing distances. An entry in this table is shown below in Figure 5.3. It consists of three fields: last encoun-

last encounter count (LEC)	distance (d)	confidence counter

Figure 5.3 An entry in the Distance Table (DT).

ter count (LEC), distance, and confidence counter. As we will describe later, we calculate the distance between two instances by taking the difference of the collision counter values at those two instances; the LEC field stores the first collision counter value. *It is used for calculating the distances for the unreused instructions.* The confidence counter is used for lending confidence to the distance value stored in the distance (d) field.

- Each entry in the RB is augmented with two additional fields: (i) the last encounter count (LEC) field and (ii) the current distance to reuse (CDR) field. The LEC field, like the LEC field in the DT, stores the first collision counter value. However, unlike the other field, *this value is used for calculating the distances for the reused instances.* The CDR field maintains the current distance to reuse for the instruction resident in the entry — *i.e.*, the remaining number of collisions it is expected to experience before reuse. On every collision to a set, the CDR value of every RB entry in that set is decremented by 1.

Calculating the distances: Next, we present how we calculate the distances — the process that is the heart of this policy. We present the whole process of distance calculation in Figure 5.4 using a pseudocode. For the purposes of clear exposition, we have divided the process into two parts: one part deals with the case when an instruction is first encountered and the other deals with the case when the instruction is re-encountered. The second case is again divided into two parts: one for the case when the instruction is reused and other when it is not.

Instruction First Encountered

```
RBset [collision counter] ++
Reserve entry in DT
DT [distance] <— 0
DT [LEC] <— RBset [collision counter]
Attempt insertion in RB
if yes
   RBent [LEC] <— RBset [collision counter]
   RBent [CDR] <— DT [distance]
```

Instruction Re-encountered

<u>If Reused</u>

```
RBent [CDR]      <— RBset [collision counter] - RBent [LEC]
RBent [LEC] <— RBset [collision counter]
DT[distance] <— RBent [CDR]
DT[LEC] <— RBset [collision counter]
```

<u>If Not Reused</u>

```
RBset [collision counter] ++
DT [distance] <— DT [distance] + (RBset [collision counter] - DT [LEC])
                                               (see text for explanation)
DT [LEC] <— RBset [collision counter]

Attempt insertion in RB
if yes
   RBent [LEC] <— RBset [collision counter]
   RBent [CDR] <— DT [distance]
```

RBent: RB entry; **RBset**: RB set; **DT**: Distance Table; **DT [LEC]**: LEC field in DT

Figure 5.4 Pseudocode depicting the distance-calculation process in the FiF policy.

We describe the process in this order. The collision counter for an RB set is incremented on every collision to the set. We define collision to a set as an instruction insertion attempt made to the set, successful or otherwise.

When an instruction is first encountered, we create an entry for it in the DT and store the current value of the collision counter from its RB set in the LEC field. The distance field in the DT is set to 0. If this instruction gets inserted in the RB, the value of the collision counter is also stored in the LEC field in the RB entry. The CDR field is set to the distance value from the DT.

The distance value for an instruction is calculated when it gets re-encountered. But, how exactly we calculate the distance value depends on whether the instruction is reused or not. We describe the two processes separately. If an instruction is reused when re-encountered, we calculate its distance value by subtracting the current value of its RB set collision counter with the value in the LEC field of the *RB entry*. This gives us the exact distance value between the original instruction instance and its reuse, and this is the reuse distance that we want in this policy. This distance value is then stored in the DT and in the CDR field of the RB entry. If a different distance value already exists in the DT entry then it is replaced if the confidence count is less than some threshold (not shown in Figure 5.4); otherwise, we decrement the confidence count. If, on the other hand, the prior distance value is same as the current one then we increment the confidence value. The current collision counter value is also stored in the LEC fields in both the RB entry and the DT.

On the other hand, if the instruction is not reused when re-encountered, we calculate its distance value as follows. Since in this case we do not have the exact LEC value for this instruction (because the instruction is not in the RB), we use the LEC value from the DT. This gives us the LEC value for the last instance of the same static instance. We subtract this LEC value from the current value of the instruction's RB set collision counter, and *cumulate with the previous distance value for the instruction*. We explain the reason for this shortly. The current value of the collision counter and the distance value are stored in the DT. If the instruction gets inserted in the RB, these values are also stored in the LEC and CDR fields in the RB entry.

Now, we explain the reason for cumulating the distance values for unreused instructions. As mentioned earlier, the distance value that we really desire is the one between the two recurrences of the same instruction instance — *i.e.*, between an instance and its repetition. For instructions that don't get reused, we cannot calculate this distance because the collision counter value of the previous instance is not known (since that instance is not in the RB). Thus, we need to come up with a way to approximate the distance values for unreused instructions. One option is to use the distances between the two consecutive occurrences of the *static* instruction. In the DT, we store the collision counter for the last occurrences of static instructions. We can calculate the distances between the two consecutive occurrences of a static instruction by simply subtracting the LEC value in the DT with the current collision counter value (RBset [collision counter] - DT [LEC], as shown in Figure 5.4). But, the distance between two consecutive occurrences of static instructions can be significantly smaller than the distance between an instance and its repetition; thus, using that as approximation for distance may give an unduly high priority to unreused instructions. We would like to give such instructions low priority, *i.e.*, high distance values. Thus, instead of using the distances between the consecutive occurrences of static instructions per se, we cumulate these distance values — *i.e.*, we add the currently calculated distance to the previous value present in the DT.

This way, instructions that are not reused repeatedly get their distance values increased progressively, and thereby, become of lower and lower priority, which makes it harder for them to secure a place in the RB (which should be the case). Cumulating the distance values is also consistent with the philosophy of the algorithm: if an instruction is not reused, and the previous and the current distance values are 'd' and 'd1', respectively, then we know that the repeating instance that we are trying to calculate the distance from existed at least 'd + d1' distance behind. Hence, in the absence of better knowledge the distance value for the instruction must be at least that much.

Policy Operations: Once the distance is calculated, the policy works as follows. When an instruction is to be inserted in the RB, we read its distance value (d) from the DT. This incoming instruction is not inserted in RB if the instructions present in its RB set have distances (in their CDR field) smaller than 'd' (which means they are more likely to get reused than the incoming instruction). However, if the incoming instruction is found qualified for insertion, then the instruction in the RB set that has the largest CDR value is chosen for replacement (since this is the least likely reusable instruction of all).

At the time of insertion, the distance 'd' is stored in the CDR field of the RB entry. This value is decremented on every collision to that set. Decreasing the CDR value — which signifies coming close to the time of reuse — increases the importance of instructions and makes them more difficult to replace (as should be the case, intuitively). However, if the CDR value gets decremented all the way to 0 and the instruction is not reused, then the instruction loses its importance and, thereafter, is considered for replacement ahead of other instructions in the set. If, however, the instruction gets reuse, then its CDR value is replenished with the current distance value from the DT.

5.5 Evaluation of Management Policies

In this section, we evaluate the management policies discussed in the previous section. These policies have parameters that control their behavior: e.g., the *threshold value* in FnReused, the *counter value* in RR, the table sizes of FnReused and FiF. Ideally, we would like to vary all the parameters and perform a thorough evaluation of these policies. However, that would be digressing from the main focus of the thesis which is the instruction reuse technique and not buffer management. For this purpose, we select the parameter values as follows. For the threshold and counter values in FnReused and RR, respectively, we conducted short simulation studies to determine with what parameter values the policies performed better, in general, and selected those values as the parameter values for the rest of our simulations. The tables used in policies FnReused and FiF are conflict-free — *i.e.*, every static instruction gets a separate table entry. We do so to evaluate the algorithm of the policies, independent of the implementation effects. The configurations that we simulate are shown in Table 5.2. The threshold

Policies	Configuration
FnReused	Threshold = 16; conflict-free table.
RR	Counter value = 8
FiF	conflict-free DT

Table 5.2 Configurations of different policies that are studied.

value selected for the FnReused policy (by the above the method) is 16, which means that an instruction has to get evicted from the RB without getting reused 16 consecutive times to be classified as unreusable. The counter value selected for the RR policy is 8, which means that a reused instruction has 8 “lives” in the RB.

We present the following results in this section. First, we show the amount of reduction achieved by these policies in the number of instructions that get inserted in the RB or the number of likely-reusable instructions that get evicted from the RB. These results give us a direct measure of how successful the new policies are at selective insertion or eviction. After this, we present the stability of the distance values in programs, which is an important factor that determines the effectiveness of the FiF policy. Finally, we show the overall impact of these policies on the reuse rates and bottomline performance (IPC).

5.5.1 Direct measures of policy operation

In Figure 5.5, we show the effectiveness of the three policies — FnReused, FnReady, and FiF — which perform selective insertion, in cutting down the number of instructions inserted in the RB. Thus, 80% on this figure means that 80% fewer instructions were inserted in the RB than in the case of the LRU policy. The numbers are shown for a 4096-entry, 4-way asso-

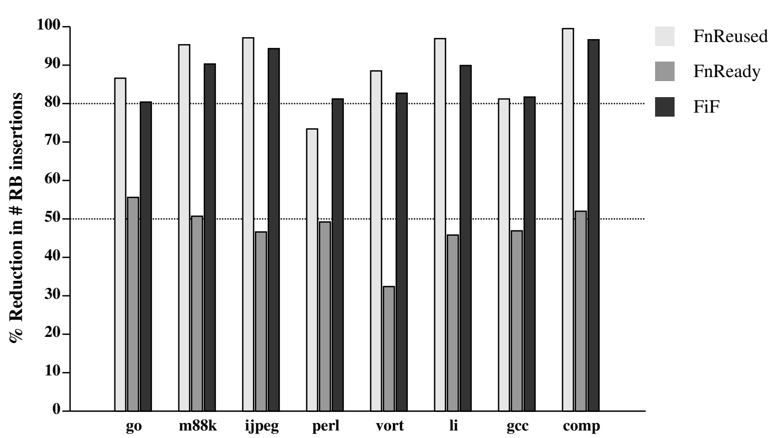


Figure 5.5 Reduction in the number of RB insertions due to FnReused, FnReady, and FiF policies. Note that in this graph higher numbers mean more reduction.

ciative RB (which is representative of the numbers for other RB sizes).

We see that there is a significant reduction overall in the number of instruction inserted in the RB. Both FnReused and FiF cut down the insertion traffic by more than 80% in all cases (except for perl where FnReused cuts down by 72%). In general, FnReused filters slightly more number of instructions than FiF. On the other hand, the reduction caused by the FnReady policy in comparison is much smaller (as would be expected from the discussion in Section 5.4.1.2), but it is still very significant, being close to 50% in most cases. However, it is understood that these reductions may not necessarily translate into improvement in reuse rates or into speedups. Those numbers we will see shortly in Section 5.5.3 and 5.5.4.

One particular interpretation of these results may be interesting. As we shall see later, for several benchmarks, the impact of these policies on the reuse rates and speedups results are small — *i.e.*, the values for these metrics are not much different with and without the new policies. In that light, the results in Figure 5.5 show that we can obtain the same amount of reuse rates and performance with much less RB activity — between 50% to 80% less activity. Since the RB is a large structure, this reduction in activity may result in significant decrease in power consumption. However, power saving is not a topic of consideration in this thesis, and, hence, we do not follow this line of investigation any further.

Next, we address the policy RR. RR, as described earlier, attempts to prevent the eviction from the RB of the likely reusable instructions, which it defines as instructions that have been reused in the past. In Figure 5.6, we show the percentage by which RR is able to reduce the eviction of the likely-reusable instructions (the percentages are over the LRU case). We show the numbers for three RB sizes: 256-, 4k-, and 64k-entries. As we can see from the figure, the

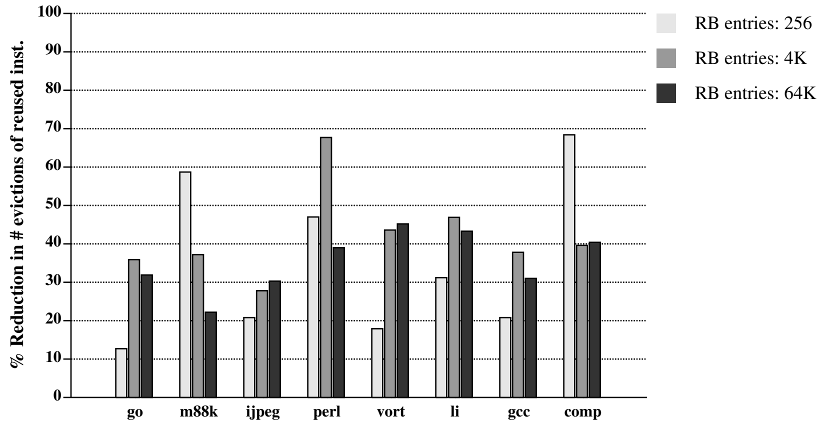


Figure 5.6 Reduction in the number of reused instructions due to the RR policy. Note that in this graph larger numbers mean more reduction.

amount of reduction varies widely with the RB size and even the trend is not consistent. An increase in RB size decreases the contention in the RB; this can both decrease or increase the percentage reduction in eviction. The contention can decrease because with less contention fewer reused instructions get evicted in the first place. The decrease in the contention can also increase the impact of RR because while earlier the policy was unable to retain instruction in RB due to high contention, it is able to do so now that the contention has reduced. In any case, we see that the reduction in reused instruction eviction is significant in several cases. Overall, RR is able to cut down the eviction of likely-reusable instructions by 30-40% (with some exceptions) for large RBs (4k- and 64k-entries) and it is able to do so by 10-30% (with some exceptions) for the small RB.

5.5.2 Stability of distances in FiF

Before we present the reuse rates and speedups results, we briefly study an important property of programs — the stability of the distances between instruction instances. The effectiveness of the FiF policy is closely tied to the accuracy of the distance estimates. And, the accuracy of distance estimates, depend not only on the distance-calculating mechanisms but also on the inherent nature of the programs — i.e, whether the instructions actually do recur at stable and, hence, predictable distances. In this section, we measure this inherent stability of distances in programs.

One measure of stability is how distances differ between the two consecutive repetition of an instruction instance. We explain this with an illustration. Suppose I_1 is a dynamic instance of the static instruction I. Also, suppose, I_1 has three repetition I^l_1, I^l_2, I^l_3 , which occur in the program as shown in Figure 5.7. Let the distance between I^l_2 and I^l_1 be d_1 and between I^l_3 and I^l_2 be d_2. Then the stability can be measured in terms of how d_2 is different from d_1: if d_2 is the same as or close to d_1, we can consider the distances as stable (predictable from the previous distance), otherwise not.[4] In Figure 5.8, we present this measure of stability: the bar labelled “exact” shows the percentage of all distances that are exactly same as the last one, *i.e.*, $d_2 = d_1$; and those labelled ±1, ±4, ±8, and ±16, show the percentage of distances that differ from the previous one by the amounts 1, 4, 8, and 16, respectively. The distances are measured as described in Section 5.4.3.2 using a 256-set RB. We employed a large buffer, which can store up to 2000 instances for every static instruction,[5] to remember the previous distances

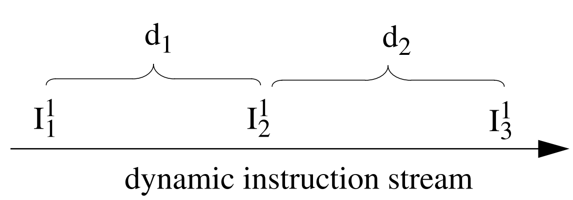


Figure 5.7 Illustration of the distances that are compared in Figure 5.8.

4. There can be other more relaxed definition of stability — we can track non-consecutive distances. However, here, since we make our prediction based on the last distance, we base our measure of stability on the sameness of consecutive distances.
5. This is the same as the buffer used in Chapter 3 for determining the amount of repetition.

for dynamic instances.

In Figure 5.8, we see that for 5 benchmarks (*m88ksim*, *perl*, *vortex*, *li*, and *compress*), the distances are fairly stable: between 42-77% of the distances are exactly the same as the last one — meaning they are predictable based on the last distance. Also, for these benchmarks, more than 70% of the distances differ from the last one by less than ±4. Although they are not exact predictions, the distances which are in error by a small amount, may still cause the policy to make the right management decision.

For the other three benchmarks (*go*, *ijpeg*, and *gcc*) the distances are not as stable. Very few times the distances are the same as the last one (e.g., only 18% for *go* and 28% for *gcc*), while a significant percentage of the distances differ from the last one by more than ±16 (e.g., more than 50% for *go*, 40% for *ijpeg*, 35% for *gcc*).

Thus, we see that stability of distances may vary from benchmark to benchmark. Also, based on the above results, we expect that the FiF policy will be more effective for the first

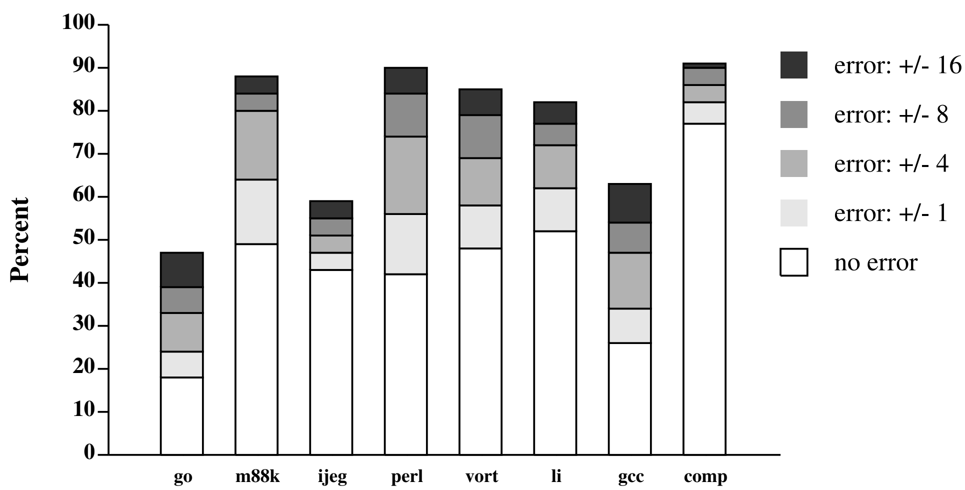


Figure 5.8 Stability of distances in programs. "exact" stands for perfect estimation: the actual distance turned out to be equal to that predicted. "diff +/- 1", etc., are cases when the predicted distance was off by 1, etc., from the actual distance. The space above the bars denotes the percent of times when the distances were in error by more that +/- 16.

five benchmarks than for the next three (in fact, as we will see later, FiF causes degradation in reuse rates for some RB sizes for the next three benchmarks).

5.5.3 Reuse Rates

We show the reuse rates for the new, limit, and LRU policies in Figure 5.9. The reuse rates are shown for RB sizes ranging from 256 to 64k entries (shown on the x-axis in $\log_2$). Overall, we

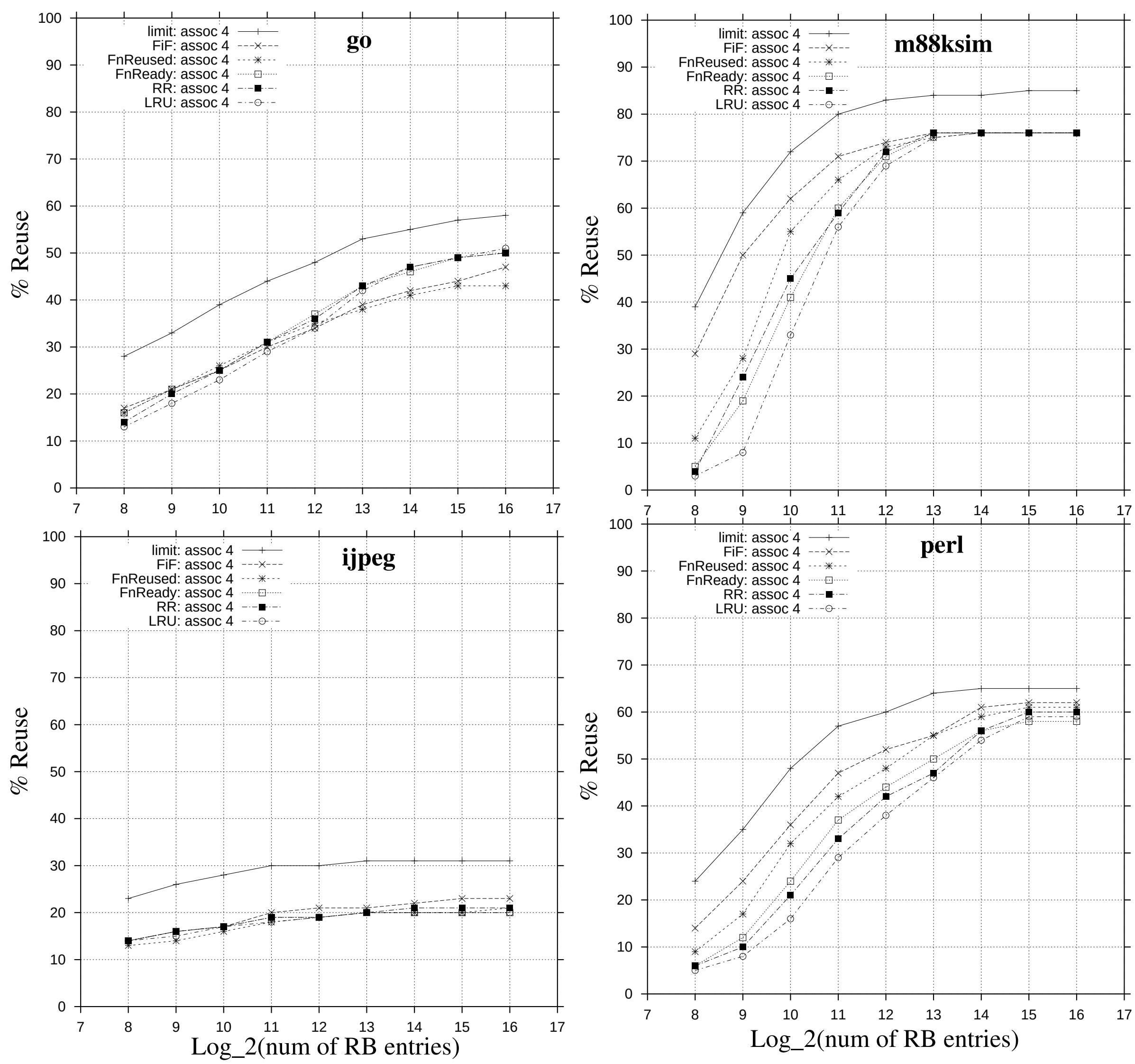


Figure 5.9 Reuse rates obtained with the 4 new policies. The reuse rates for the limit and LRU policy are also plotted for comparison. (This figure continues on the next page).

see that the results are mixed: the new polices perform very well for some benchmarks (e.g., *m88ksim*, *perl*, and *vortex*) but poorly for others (e.g., *go*, *ijpeg*, and *gcc*). Although, for *li* and *compress* they don't improve the reuse rate significantly, in absolute terms, they (specially FiF) are successful in nearly eliminating the gap between the LRU and the limit (which is small to begin with). Next, we interpret the results for individual schemes.

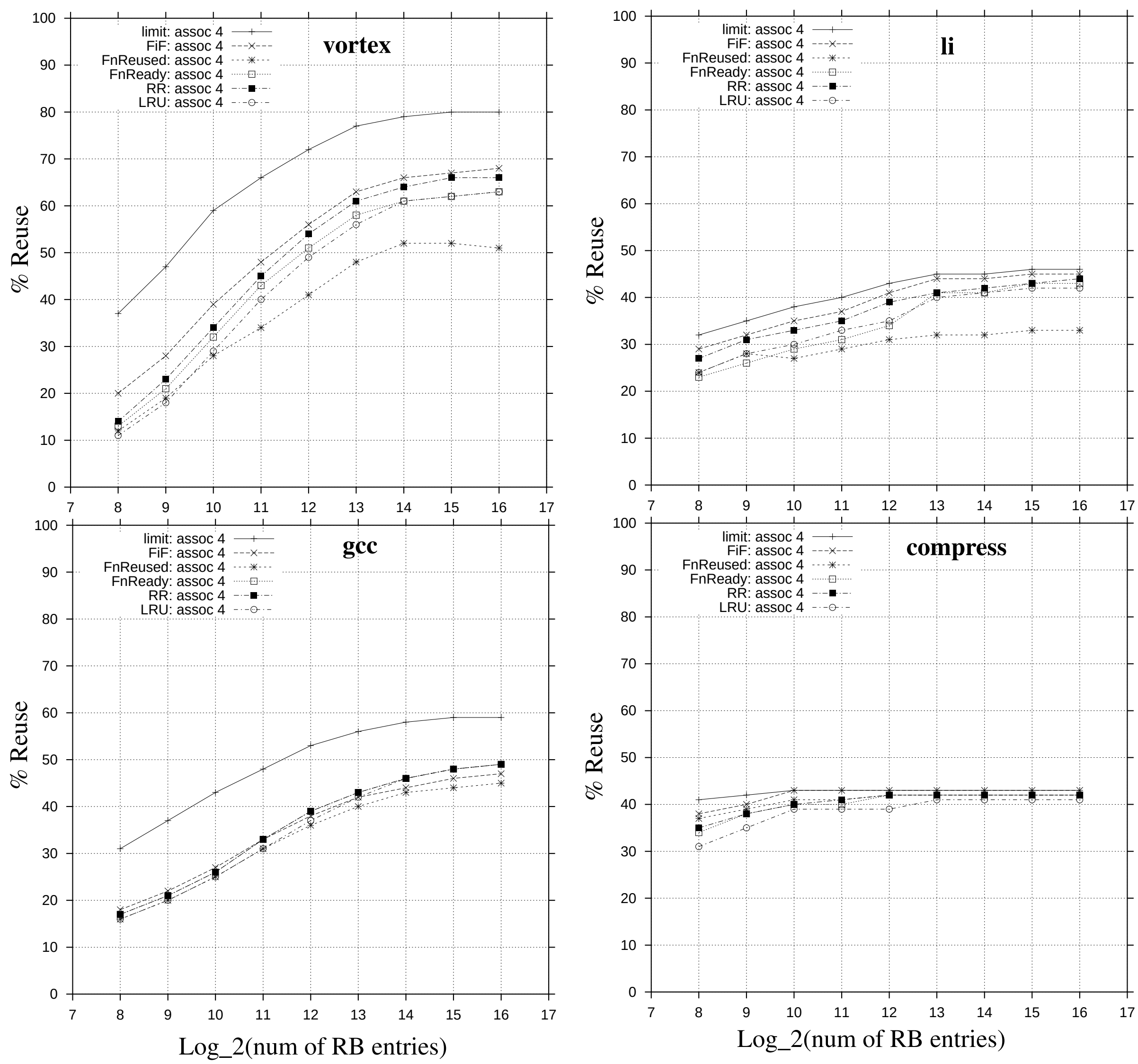


Figure 5.9 (contd.) Reuse rates obtained with the 4 new policies. The reuse rates for the limit and LRU policy are also plotted for comparison.

FiF: This policy performs better than other policies for almost every benchmark. This is especially true for small RB sizes (256-2k entries). For example, in *m88ksim* the reuse rate improves from 9% (LRU) to 50% (FiF) for the 512-entry RB; similarly, for *perl* the reuse rate improve from 16%(LRU) to 36% (FiF) for the 1024-entry RB; improvements in the case of *vortex* are also significant, being about 10%-points for various RB sizes. Seen another way, this result means that a particular reuse rate can be obtained with a much smaller — up to two to four times smaller — RB with FiF than LRU. For *go*, *ijpeg*, and *gcc*, like other policies, the improvement in the reuse rates with FiF are negligible. For *li* and *compress*, on the other hand, though the improvements in reuse rates are small, FiF policy nearly eliminates the gap between the LRU and the limit policy.

FnReused (filter not reused instructions): This policy performs well for a couple of benchmarks (m88ksim and perl), but not so well for others. In fact, for *vortex* and *li*, it actually degrades the reuse rates below the LRU level. For, *m88ksim* and *perl*, cases where it performs well, it improves the reuse rates significantly: e.g., from 9%(LRU) to 29% for 512-entry RB in *m88ksim*; from 16%(LRU) to 31% for 1024-entry RB for *perl* (performing close to FiF). The degradation in reuse rate occurs because, as mentioned in its description in Section 5.4.1, it does not have a mechanism to correct itself when it goes wrong. After it starts filtering an instruction from the RB, it does not have a way of knowing when the instructions becomes reusable. Thus, if the instruction becomes reusable later, it would still filter it from the RB and hence miss reusing it. This causes it to degrade reuse rates for *vortex* and *li*.

FnReady (filter not ready instructions): This policy, in general, improves the reuse rates by a small amount, but this amount is significant considering its simplicity and possible ease of implementation. For example, improvement in the reuse rates for RB sizes between 512- and 4096-entries are in the range of 1-10% points for *m88ksim*, around 5% points for *perl*, and around 1% point for *vortex*. Except for *li*, where it slightly degrades the reuse rate below LRU, this policy always improves reuse rates over LRU.

RR (retain reused): This policy, also, improves reuse rate by a small amount. The improvements are around 2-3% points for *perl* and *vortex*, and between 1-12% points for *m88ksim* for RB sizes ranging between 512- and 4096-entries. This policy may also be inexpensive to implement, relative to FiF and FnReused, since it does not require a separate table for storing the state. Given that, the reuse rate improvements achieved by this policy may be note-worthy.

In general, we note that policies FnReused, FnReady, and RR perform worse than FiF. This can be attributed to the fact that they are all "special-purpose" policies; *i.e.*, they optimize certain aspects of the buffer management problem and don't attack in a general way. These "special-purpose" policies work only when the aspect they optimize happens to be the main cause of poor buffer management. For example, FnReady will improve reuse rates when many reusable instructions get evicted from the RB by not-ready instructions (hence, filtering these not-ready instructions will likely allow the reusable instructions to get reused); Similarly, FnReused will improve reuse rates when many reusable instructions are evicted from the RB by instructions that are persistently not reused; RR will improve reuse rates when instructions that are reused once have the propensity to get reused repeatedly. In places where these aspects are not prominent, the "special-purpose" policies have a limited impact on the reuse rate (they may, in fact, degrade the reuse rate when the limited improvements they achieve is not sufficient to offset the effects of the occasional sub-optimal decisions they make). FiF, on the other hand, attacks the buffer management problem in a general way and, hence, is able to adapt better to the changing RB access patterns and, consequently, able to perform well for most benchmarks.

5.5.4 Performance

In Figure 5.10, we present the speedups obtained with IR (over the machine without IR) using LRU and FiF policies. The speedups are calculated as follows: $((IPC_{IR}/IPC_{w/oIR})$ - 1)*100. The experiments were run with 4-way associative RBs, ranging in size from 256- to 64k-entry. We make the following observations from this figure:

- For both policies, speedups follow the reuse rates (shown in Figure 5.9) closely: an increase in the reuse rate in most cases entails an increase in the amount of speedup. Since these trends also exist for other policies, for the sake of clarity, we do not include their speedup graphs in this figure; their relative position can be inferred from their reuse rates.
- Comparing the two policies, we see that differences in their reuse rates are also, in most cases, reflected in differences in their speedup numbers. If FiF policy reuses more instructions than LRU, it also shows higher speedups than LRU, and vice versa. An exception is *vortex*; despite improvement in the reuse rate by FiF, there is no (or very small) improvement in the speedups. This shows that the additional instructions reused by the FiF policy are not executional bottlenecks (a closer look at the benchmark showed that many were, for example, branches that get predicted correctly); hence, their reuse does not impact the bottomline performance.
- Again, as would be expected from the reuse result, in most cases, the difference in speedups is small, about 1%-point in many cases. But, in cases of *m88ksim* and *perl*, we notice

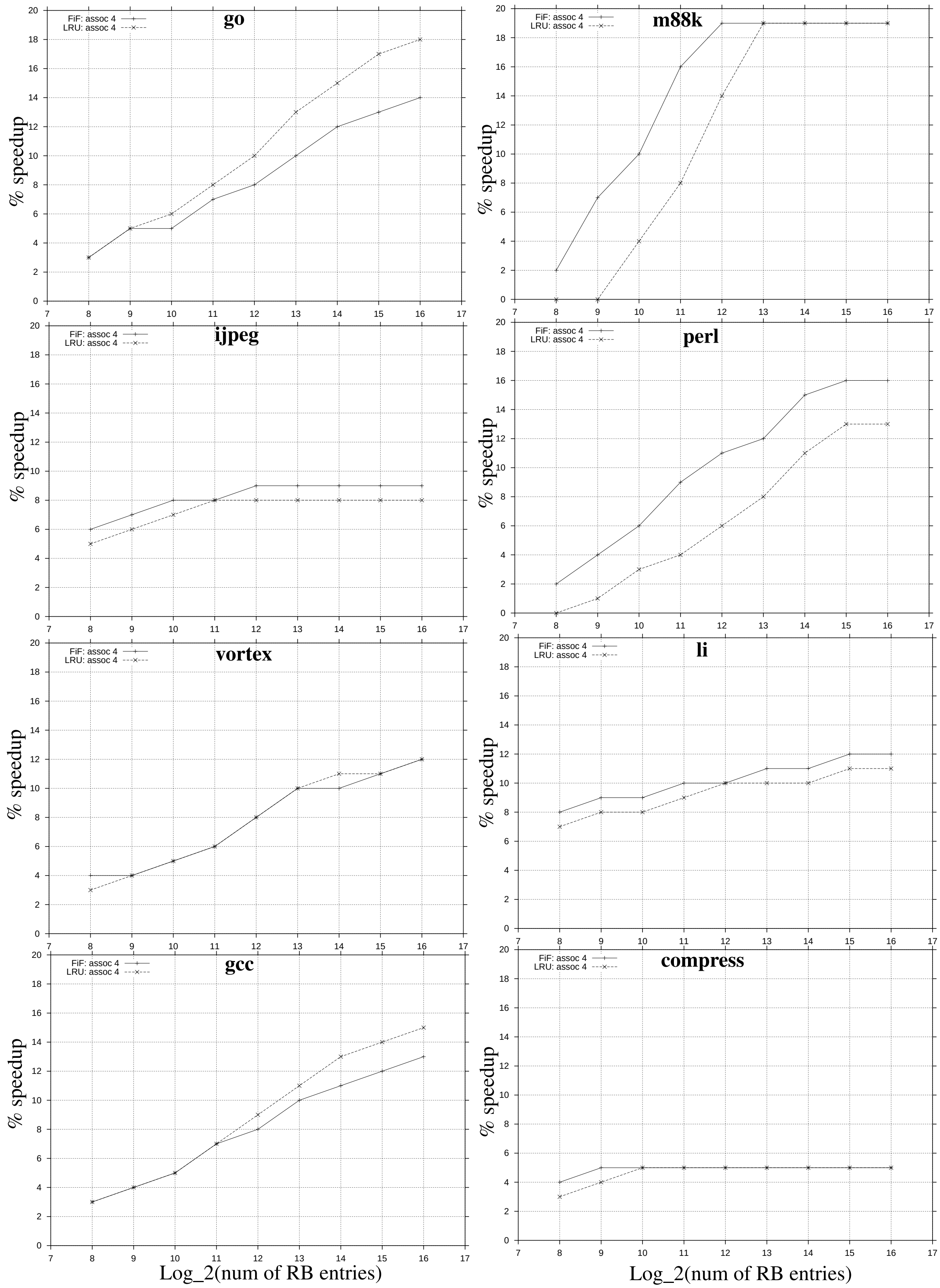


Figure 5.10 Speedups over the base case (no IR) of IR using RB with LRU and FiF policy.

significant improvement in speedups: e.g., for *m88ksim* speedups increase from 8% (LRU) to 16% (FiF) for 2k-entry RB; and for *perl,* speedups increase from 4% (LRU) to 9% (FiF).

5.6 Summary and Conclusions

In this chapter, we study the RB — the main hardware structure used in IR — in more detail. We present how the reuse rates vary with RB size for different conventional replacement policies, such as LRU, FIFO, and Random. We also present the limit reuse rates for each RB size, giving us an upper bound on the amount of reuse we can capture per RB size. We see a significant difference between the limit and convention policies' reuse rates, suggesting that the RB may be very inefficiently utilized.

We also characterize the RB with respect to its associativity. We show how many instances later instructions normally get reused. This gives us a lower bound on the amount of associativity needed to capture a certain amount of reuse. We see from the results that, for most benchmarks, a significant percentage of total reuse may be captured with a small degree of associativity, since most — more than 70% — of the instructions get reused within their next 4 instances. We also show how the reuse rates vary with associativity, concluding that higher associativity is more important for large RBs than for small ones. We also observe, that even with large associativities, a significant gap remains between the limit and conventional policies and that increasing the associativity does not improve RB utilization.

We study four RB management policies to improve RB utilization: FnReused, FnReady, RR, and FiF. The first two policies perform selective insertion in the RB, filtering out instructions that are not likely to be reused. The RR policy performs selective eviction, retaining instructions in the RB that are likely to be reused. The FiF policy is a novel management policy which chooses instructions for keeping in the RB based on their likelihood of reuse. This policy calculates the likelihood of reuse for instructions in terms of their chances of getting evicted from RB before being reused — more the chances of getting evicted, less the likelihood of reuse.

The results for the management policies vary with benchmarks. For some benchmarks (*m88ksim*, *perl*, and *vortex*), we see a significant improvement in the reuse rate with the new policies over LRU (e.g., in *m88ksim* the reuse rate improves from 9% to 50% for FiF policy for 512-entry RB). Interpreting the results another way, for these benchmarks, the same amount of reuse can be captured with an RB up to 2-4 times smaller using the new policies. However, for benchmarks such as *go*, *ijpeg*, and *gcc*, we see a negligible improvement or a degradation in the reuse rate with the new policies. We also show the speedups attained by IR with LRU and FiF policies and note that in most cases the improvements in performance closely follow the improvements in the reuse rate (except for *vortex*).

This work can be further extended in several possible directions. Other methods of buffer management that can be explored. In this work, we gave equal importance to every instruction. It is conceivable to have a policy that treats different instruction-types differently, giving less preference to instruction-types that are less likely to get reused. Similarly, we can think of performing the buffer management with profitability of reuse as a criterion in which only those instruction sequences that will be profitable to reuse (cause performance improvement) are kept in the RB. It is also possible to have an hybrid of different policies: for example, the FnReady policy can be used with other policies, or the FnReused and RR policy can be used together (since they complement each other well). Lastly, we note that the FiF policy is a general buffer management policy: it can also be used for managing other forms of memory storage, such as caches. In this thesis, we evaluated it for IR; it will be interesting to see how well it performs in managing other forms of storage structures.

Chapter 6

Sensitivity Analysis

In previous chapters, we evaluated the concept of IR in the context of one particular processor pipeline. It is reasonable to expect that the IR may perform differently for other types of pipelines. The task of this chapter is to develop a sense of how might the IR performance change when the pipeline configuration is varied. For this purpose, in this chapter, we select a few key parameters of pipeline configuration and vary their values, and study how sensitive the IR results are to these variations.

We investigate six important pipeline parameters in this chapter: (i) instruction window size, (ii) pipeline width, (iii) pipeline length, (iv) branch prediction accuracy, (v) memory latency, and (vi) reuse latency. These parameters were selected because they are likely to be different for different processors and are also likely to impact the performance of IR for reasons explained later in the chapter.

The rest of this chapter is laid out as follows. In the next section, we present the experimental setup used in this study. In Section 6.2, we describe at a high level why IR results may vary with the above mentioned parameters. In Sections 6.3-6.8, we analyze the sensitivity of IR to the above mentioned parameters. In each section, we discuss why and how the reuse rates and the reuse performance may vary by varying the parameter studied in that section. We finish each section by presenting and discussing the simulation results. Finally, in Section 6.9, we summarize this chapter and provide conclusions.

6.1 Experimental Setup

The base machine, over which the speedup numbers are measured, is the same as the one described in Chapter 2, except for the parameters that we vary for different experiments. Thus, for example, in window-size experiments, the base machine will have all its parameters as described in Chapter 2 except for window size, which will be the same as the value used in the experiment (32, 64 or 128). IR is implemented using scheme S_v and a 4-way associative RB with 1024 entries managed with the LRU policy (except in reuse latency experiments where we vary the RB sizes from 256- to 16k-entries.). In cases where it is possible that the impacts on IR may be different for other reuse schemes (such as S_n and S_{n+d}), we point this out during discussion and suggest the likely difference.

We evaluate the impact on IR in terms of changes in two metrics: (i) the *reuse rate* — *i.e.*, the percentage of instructions reused; and (ii) the *reuse performance* — *i.e.*, percentage speedup over the (appropriate) base machine IPC. In several places in this chapter, we refer to two metrics together as the *reuse results*.

6.2 Causes for Sensitivity of IR

Before we can analyze how the reuse results may vary with individual processor parameters, we need to understand what causes the reuse results to be sensitive in the first place. That is, why may the reuse rates and reuse performance change when the underlying processor is changed? What is it in the way we reuse instructions or in the way reuse improves performance that may change when the processor is changed? In this section, we highlight some such aspects of IR. The discussion here will help us better follow the effects of the individual parameters presented in the later sections.

First, we consider the reuse rates. For the instructions to get reused (with scheme S_v) there is a strict requirement — in the way we implement IR in this thesis — that the operand values of the instructions must be ready at the reuse stage.[1] This is *the* main reason why reuse rates may get affected by changes in the underlying processor. Changing the processor parameters may change — move backward or forward — the ready time of operands, impacting the reuse rates accordingly: the reuse rates may improve if the operands become ready sooner (*i.e.*, the ready time is moved backward); they may decrease, otherwise. In later sections, we will see how exactly each parameter influences the ready time of operands.

Next, we consider the reuse performance. Since reuse can improve performance because of several reasons, its performance can get impacted in several ways by changing the underlying processor. To understand how the overall reuse performance may change, we need to understand how its different components may get impacted. We discuss the impact on different components below.

- An important reason why reuse improves performance is that it collapses data dependences by reusing a dependent chain of instructions in a single cycle. How much performance is gained because of this component depends on two things: (i) the length of the dependent chains reused — longer chains means more data dependences are collapsed, and, hence, means more benefit; (ii) the importance of dataflow latencies in the total exe-

1. For reuse schemes S_n and S_{n+d}, the requirements for reuse are more stringent, as mentioned in Chapter 4

cution time of programs — if dataflow latencies are important, then collapsing them will mean more benefit. The reuse performance due to this component will get impacted by any change in the processor that affects either of the above two factors. If the length of dependent chains that can be reused in a cycle changes (e.g., due to changes in machine width) or if the importance of dataflow latencies changes (e.g., due to changes in branch prediction accuracy), then the reuse performance may get affected.

- Another important source of performance is squash reuse, *i.e.*, reusing work that had been discarded because of misprediction squashes. Since squash reuse salvages useful work from the work that was performed on the mispredicted control path, it reduces the penalty of misprediction. This performance benefit not only depends on the amount of squashed work that can be reused (more a function of the reuse technique), but also on the amount of work that was thrown away in the first place (a function of the underlying processor). If the amount of work that gets thrown away changes due to changes in the processor (e.g., improvement in branch prediction accuracy), then the amount of benefit derived from squash reuse may also change accordingly.
- IR also improves performance because it generates instruction results early and, thereby, allows instructions dependent on these results to execute sooner than they would have done otherwise. The amount of benefit derived from this component depends on two factors: (i) how much earlier than the execution results do the reuse results become available — the earlier they do, more may be the benefit from reuse; (ii) how much sooner the dependent instructions actually execute using the reused value compared to using the values generated through regular execution. Thus, any changes in the processor that affect either of these factors may affect the reuse performance. If, for example, the reuse results

do not get generated much earlier than the results from regular execution (for instance, because of high reuse latency), then the reuse performance will be decreased. Likewise, if the dependent instructions get delayed (e.g., because of a pipeline stall), they may not be able to execute any sooner than the base case, diminishing the advantage due to reuse.

- Since the reused instructions do not get executed, IR frees up execution bandwidth. This is another reason why IR may improve performance, especially for machines where the execution bandwidth is not enough to exploit all the available parallelism in the window. For such processors, the execution bandwidth freed because of reuse can be used to execute other ready instructions in the window, hence, improving performance. Obviously, this component of reuse performance will be sensitive to changes in the execution bandwidth of the underlying processor: if the change is such that the execution bandwidth is decreased (e.g., pipeline width is reduced), then the potential benefit of reuse will increase, and vice versa.

The overall impact on the reuse performance of the processor changes will depend on how these individual effects interact. We discuss this further for individual processor parameters in subsequent sections.

6.3 Instruction Window Size

The size of an *instruction window* — which in our case is the same as the size of the RUU — defines the number of instructions that can be in flight at any given time in a processor. In all our previous simulations, we have used a window of 64 instructions. In this section, we investigate the impact of changing the window size on the reuse rate and the reuse performance.

Before presenting the results, let us qualitatively discuss possible impacts changes in window size may have on the reuse rates and reuse performance.

6.3.1 Possible impacts on reuse rates

Increasing the window size may decrease the number of instructions reused because more instructions may find their operands *not* ready at the reuse stage. We explain this with an example. Consider a machine with a small instruction window. The front-end of this machine may often stall because the window gets full. These stalls will provide time for instructions in the window to execute and prepare operand values for the not-yet-fetched dependent instructions. When the stall clears and instructions are fetched, the new instructions may find their operands ready when they reach the reuse stage and, hence, may get reused. A machine with a large window may stall less often because of a full window. This may permit the dependent instructions to arrive at the reuse stage too soon — before their source instructions have executed and, hence, before their operand values have become ready — and cause these instructions to not get reused.

6.3.2 Possible impact on reuse performance

An increase in window size may both increase or decrease the reuse performance. Since the reuse performance is correlated with the reuse rate, it may decrease because the reuse rate may decrease with an increase in window size (as discussed above). However, it may increase reuse performance by increasing the number of *useful* reuses — *i.e.*, reuses that actually cause the dependent instructions to execute earlier. A machine with a small window may frequently stall due to unavailability of window entries, rendering many reuses useless. For example, this situation may happen when an instruction is reused but the instructions dependent on its results are stalled. By the time these dependent instructions enter the pipeline, their operand values could become ready through normal execution; hence, they may not execute any sooner with reuse than in the base case. However, in a machine with a large instruction window, the dependent instructions may be able to get into the window and may be able to execute sooner than in the base case (hence, improving performance) by taking advantage of the source instruction reuse.

6.3.3 Results

In Figure 6.1, we show the impact of varying the instruction window size on reuse rate and reuse performance (Figures 6.1 (a) and (b), respectively). Three window sizes are studied: 32, 64, and 128 instructions. The rest of the processor parameters in the 32- and 64-instruction window experiments are the same as those presented in Chapter 2. But, for the 128-instruction window experiments, we increase the number of unresolved branches allowed in the processor from 16 to 32 to prevent this constraint from implicitly limiting the effective window size. The reuse rates presented are in terms of percentage of all committed dynamic instructions, and the speedups are in terms of improvements over the IPC of the base case (without IR). Here we also point out that each window size has a separate base case, and speedups for a particular window size are calculated over its base case.

From Figure 6.1, we can observe that, in general, varying the window size has negligible impact on reuse rate and reuse performance. For most benchmarks, the reuse rates for all three sizes are the same[2] (Figure 6.1(a)). For *ijpeg* and *vortex*, we see a small decrease in the reuse

2. Within the degree of precision — until the second decimal place — that we plot in our graphs.

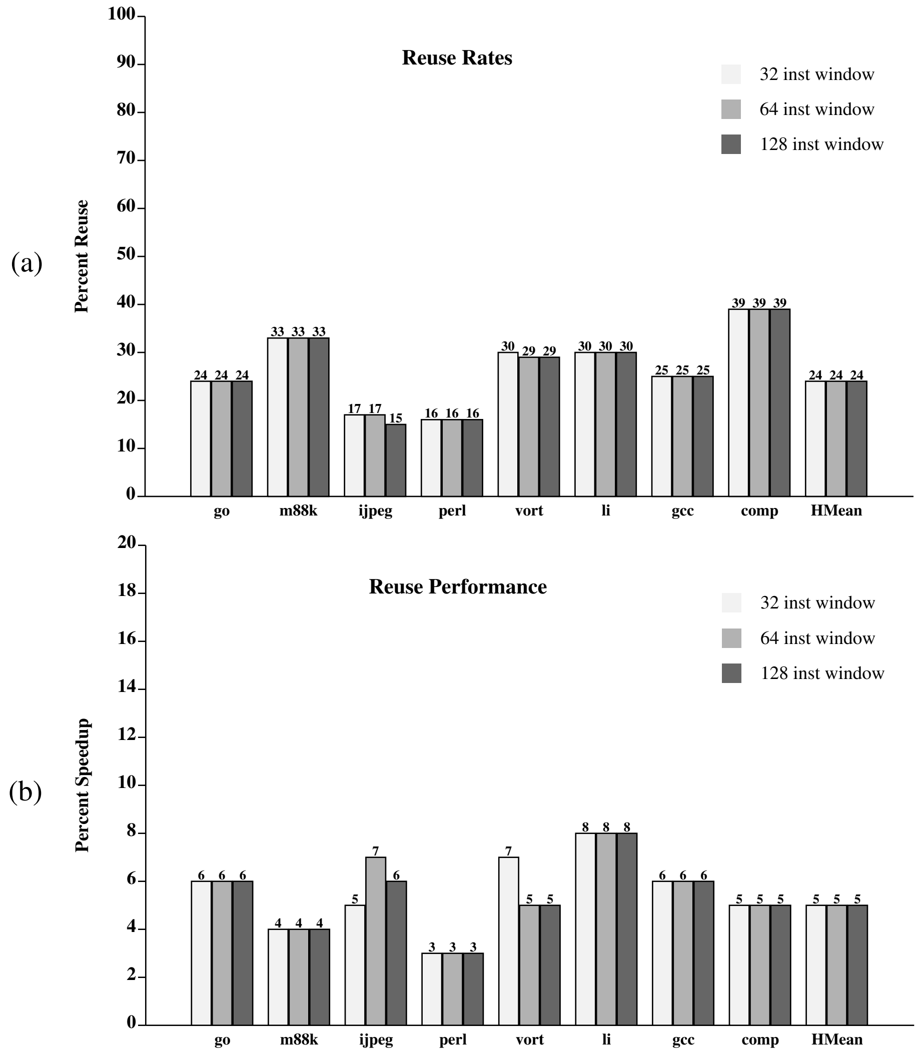


Figure 6.1 Impact of varying window sizes on (a) reuse rates (b) reuse performance.

rate for larger window sizes, which, as we have discussed earlier, occurs because some instructions enter the pipeline before their operands become ready and hence don't get reused.

In Figure 6.1(b), we see that, except for *ijpeg* and *vortex*, the amount of speedup attained by IR for all three window sizes over their respective base cases is the same for all benchmarks. For *ijpeg*, the performance increases slightly (2%-point) from window size 32 to 64 and then decreases (1%-point) for 128. The initial increase can be attributed to the fact that window size 64 converts many reuses that may be useless with window size 32 to useful ones;

the latter decrease can be attributed to the corresponding decrease in the reuse rate. For *vortex*, the reuse performance decreases from 7% to 5% when going from window size 32 to window sizes 64 and 128. Again, this decrease can be attributed to the corresponding decrease in the reuse rate.

Overall, we see that the reuse rates and the reuse performance obtained using scheme S_v are not very sensitive to the variation in the instruction window size.[3]

6.4 Pipeline Width

By pipeline width we mean the number of instructions the pipeline can fetch, decode, issue, execute, and commit, in a cycle. In all our previous simulations, we use a pipeline of width 4. In this section, we study the impact of varying pipeline width on reuse rates and reuse performance. Before presenting the results, we present a qualitative discussion on how we might expect the reuse results to be affected by the changes in pipeline width.

6.4.1 Possible impact on reuse rates

Increasing pipeline width can either increase or decrease reuse rates. The decrease may take place because as the pipeline becomes wider, more instructions may arrive at the reuse stage before their operand values are available and, hence, may not get reused. We illustrate this scenario in Figure 6.2. In this figure, we show the flow of an instruction stream — I1, I2,

3. As described in Section 4.5.2, for schemes S_n and S_{n+d}, the reuse constraints are more stringent than for scheme S_v. For example, with scheme S_{n+d}, an instruction cannot be reused if any of its unreused source instructions exist ahead in the pipeline (executed or not). The amount of reuse missed due to this constraint increases rapidly with window size because instructions may remain in the window longer, obstructing the reuse of dependent instructions. Hence, the reuse performance may be sensitive to instruction window size.

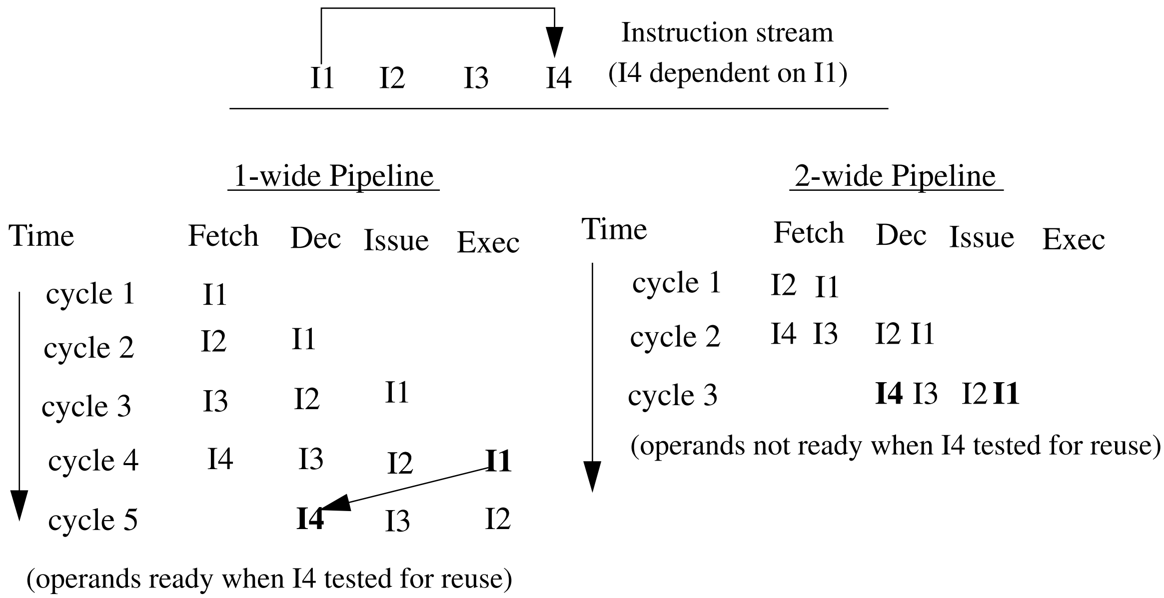


Figure 6.2 Example of how some instructions may not get reused when pipeline wide is increased. Here, when the pipeline width is increased from 1 to 2, I4 does not get reused. (In this pipeline we assume that the reuse test is performed in the decode stage).

I3, and I4, where I4 is dependent on I1 — through 1- and 2-wide pipelines. (For ease of explanation, we do not show the read register stage in the pipeline and assume that the reuse test is performed in the decode stage itself). We see that in the 1-wide machine, the source (I1) and the dependent (I4) instructions are 3-cycles apart. By the time I4 is tested for reuse (cycle 5), its operands are ready (since I1 has already executed), allowing it to get reused. But, in the 2-wide machine, due to a higher fetch and decode bandwidth, I4 arrives at the reuse stage (same as the decode stage in our example) before I1 has executed, and hence it does not get reused.

By increasing the width of a machine we not only increase the front-end bandwidth (which may hurt the reuse rate, as described above), but we also increase the execution bandwidth which may improve the reuse rate, as we describe next. With more execution bandwidth, the processor may be able to execute instructions sooner and, hence, may be able to prepare the operand values in time for the reuse of the dependent instructions. For example, consider the following scenario. Suppose that because of a few long-latency operations (e.g., cache misses), the instruction window fills up and the machine is stalled. Also suppose that most instructions in the window are dependent on these long-latency operations. After these operations complete, the stall is cleared, the dependent instructions start executing, and the new instructions start entering the window. These new instructions will not get reused if their source instructions are stuck in the backlog ahead. How rapidly this backlog gets cleared depends on the execution bandwidth of the machine: the greater the bandwidth, the faster the backlog will get cleared, and, hence, the smaller will be the number of instructions that will not get reused because of this backlog. For this reason, as the machine width is increased, it develops the potential to reuse more instructions.

6.4.2 Possible impact on reuse performance

Like the reuse rate, the reuse performance may also either improve or degrade with an increase in the pipeline width. The change in the reuse performance may take place simply in correspondence to the variation in the reuse rate: if the reuse rate increases with the pipeline width, the reuse performance may also improve, and vice versa. However, there are other reasons why changing the pipeline width may impact reuse performance, and we describe them next.

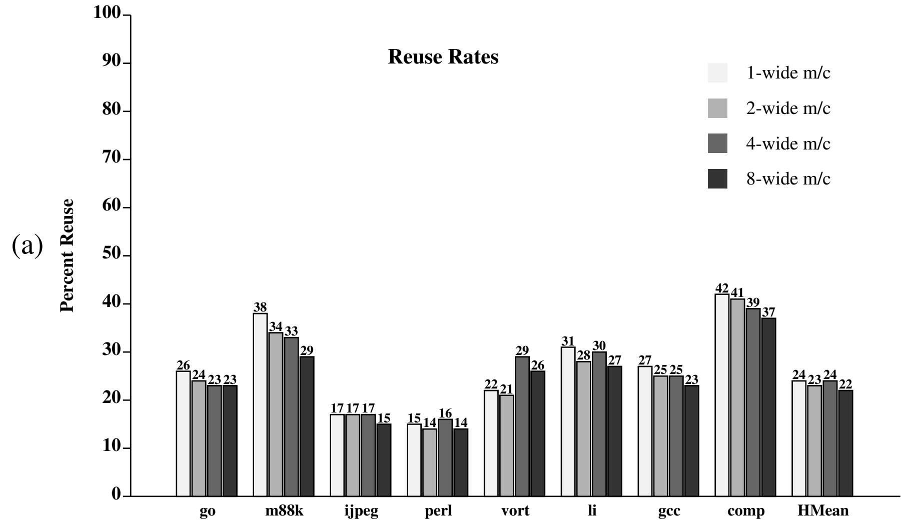
IR may improve the performance of a narrow machine more than that of a wide machine. A low execution bandwidth in a narrow machine may not be enough to exploit all the ILP present in programs — i.e, execution bandwidth may be a bottleneck for these machines. By reusing instructions we can free up execution bandwidth that can be used to execute other ready instructions, thereby, improving performance. Since the execution bandwidth is not likely to be a bottleneck for wide machines, this advantage of reuse may decrease in importance as machines are made wider, possibly decreasing its impact on performance.

Increasing the pipeline width may also improve reuse performance because it facilitates the reuse of longer chains of dependent instructions in the same cycle. In this thesis, we reuse a chain of instructions in the same cycle if all the instructions are present in the reuse stage in the same cycle. When the width of the machine is increased, the number of instructions present in the reuse stage in any cycle also increases, thereby, increasing the chances of reusing longer chains of instructions. Reusing longer chains of instructions in a cycle will impact the performance more. Hence, increasing the machine width may improve reuse performance.

6.4.3 Results

In Figure 6.3, we show the impact of varying the machine width on reuse rates (figure a) and reuse performance (figure b). The results are shown for 4 machine widths, 1-, 2-, 4-, and 8-way.

Although, on average, we see that reuse rates are to a large extent insensitive to machine widths, for individual benchmarks we see interesting variations due to the interplay of various impacts of making the machine wider, as discussed in the previous section. For some benchmarks (*m88ksim*, *go*, and *compress*) we see a steady decrease in reuse rates (although slight) as the machine is made wider. This happens because increasing number of instructions arrive at the reuse stage before their operands are ready for the reasons discussed earlier. For some benchmarks, e.g., *vortex*, *perl*, and *li*, we seen an increase in the reuse rate with the increase in



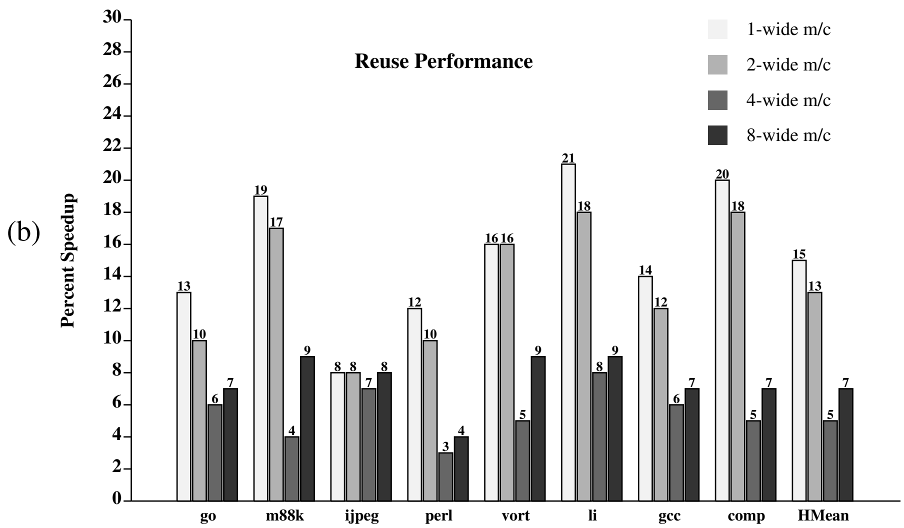


Figure 6.3 Impact of varying the machine width on the (a) reuse rates and the (b) reuse performance.

width from 2-way to 4-way machine. This happens because of the favorable effects of increasing the execution bandwidth, as discussed earlier.

In Figure 6.3 (b), we show the impact of varying the machine width on reuse performance. We see that the reuse performance is extremely sensitive to machine width: with the performance improvement being the most for the narrow machines, e.g., 1-way or 2-way wide. As we have discussed earlier, this can be attributed to the fact that for narrow machines the execution bandwidth is a bottleneck, and reuse frees up execution bandwidth that can be used by other ready instructions. For 1-way and 2-way machines, the average speedups attained with reuse are 15% and 13%, respectively; for some benchmarks, e.g., *li* and *compress*, the speedups are as high as 20%. We see a significant drop in speedups when the machine width is increased to 4-way because for 4-way machines the execution bandwidth is less of a bottleneck. However, we, again, see an improvement in speedups for 8-way machine, which can be attributed to reusing longer chains of instructions per cycle, as described earlier.

6.5 Pipeline Length

In our earlier experiments, we used a 6-stage pipeline, as shown in Figure 2.1. A pipeline of a different length may impact reuse rate and reuse performance differently. In this section, we study the impact of varying the pipeline length on reuse results.

With respect to the reuse stage, the pipeline (Figure 6.4 (a)) can change in two ways —

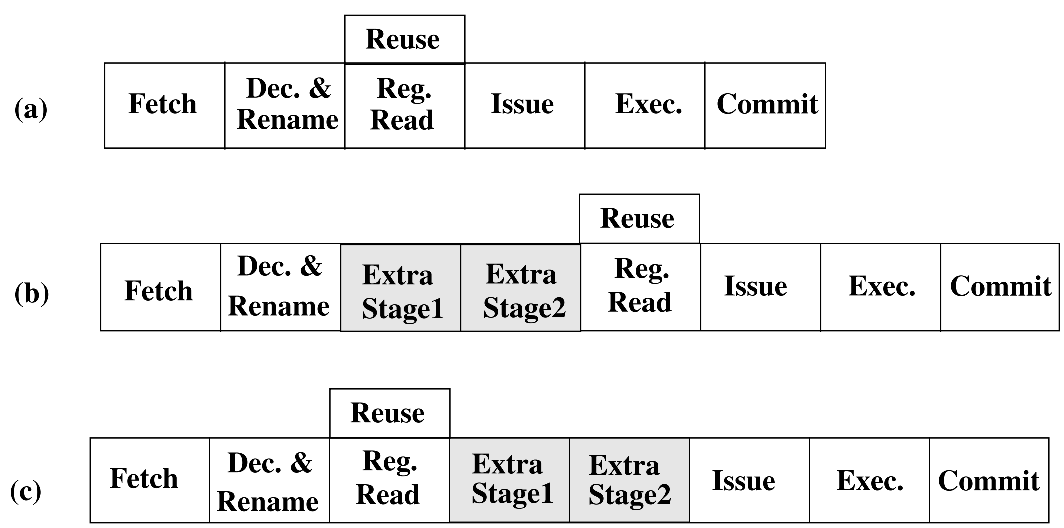


Figure 6.4 The default pipeline (a) and the pipelines with extra stages (b and c). In (b) the extra stages are before Reuse stage (same as Register Read stage), and in (c) the extra stages are after Reuse Stage.

either there are extra stages before it (Figure 6.4 (b)) or there are extra stages after it (Figure 6.4 (c)). As we will discuss next, the reuse results may get impacted differently depending on which is the case. In this section, we study the impact of the latter two pipelines on reuse results. But, before we present the simulation results, we qualitatively discuss how the two pipelines shown in Figures 6.4 (b) and (c) may affect the reuse results.

6.5.1 Possible impact on reuse rates

We discuss the effect of adding extra stages before and after the reuse stage separately. The addition of extra stages before the reuse stage should not affect the reuse rate, provided they are non-stalling. This is because they do not insert any delay between the source and the dependent instructions and, hence, do not obstruct any reuse. However, if these stages are stalling, then they may affect the reuse rate in the same way as any change that could cause the front-end of the machine to stall (e.g., small-window size, which we have already discussed in Section 6.3). However, non-stalling stages before the reuse stage may affect reuse performance in other ways, which we will describe shortly, in the following section.

If the extra stages are added after the reuse stage (Figure 6.4 (c)), then they may lower the reuse rate. This is because these extra stages add latency between the execute and the reuse stage and, thereby, increase the number of times the operand values will not be ready for instructions at the reuse stage (because the source instructions have not yet executed).

6.5.2 Possible impact on reuse performance

The performance benefit due to reuse is, in part, also dependent on the fraction of the pipeline that is skipped by the reused instructions: if a big fraction of the pipeline is skipped then the impact on performance is more, since the big skip cuts down a greater part of the latency through the pipeline; however, if a small fraction is skipped then the performance benefit is small. The benefit of skipping a part of pipeline is realized when the latency of the pipeline is exposed, such as at the time of branch misprediction squash. The fraction of the pipeline skipped is different depending on whether the extra stages are before or after the reuse stage Hence, the performance impact of reuse in the two cases is likely to be different.

Adding the stages before the reuse stage elongates the pipeline but keeps the number of stages skipped by the reused instruction the same (Figure 6.4 (b)). Hence, the proportion of the pipeline skipped by the reused instruction decreases, thereby, possibly decreasing the impact of reuse on performance.

Adding the stages after the reuse stage (Figure 6.4 (c)) has two opposite effects. On the positive side, it increases the number of stages and the proportion of the pipeline skipped by the reused instructions. In such a pipeline, reuse may have greater impact on performance. However, on the negative side, adding extra stages after the reuse stage may lower the reuse rate, as we have discussed earlier, and, hence, may cause the reuse performance to degrade.

6.5.3 Results

In Figure 6.5, we show how the reuse results vary when the pipeline length is varied. The reuse rates and reuse performance are shown in Figures 6.5 (a) and (b), respectively. The pipeline length is varied by adding 1 or 2 extra stages before or after the reuse stage, as shown in Figures 6.4 (b) and (c). These extra stages are non-stalling; *i.e.*, they don't generate pipeline stalls by themselves.

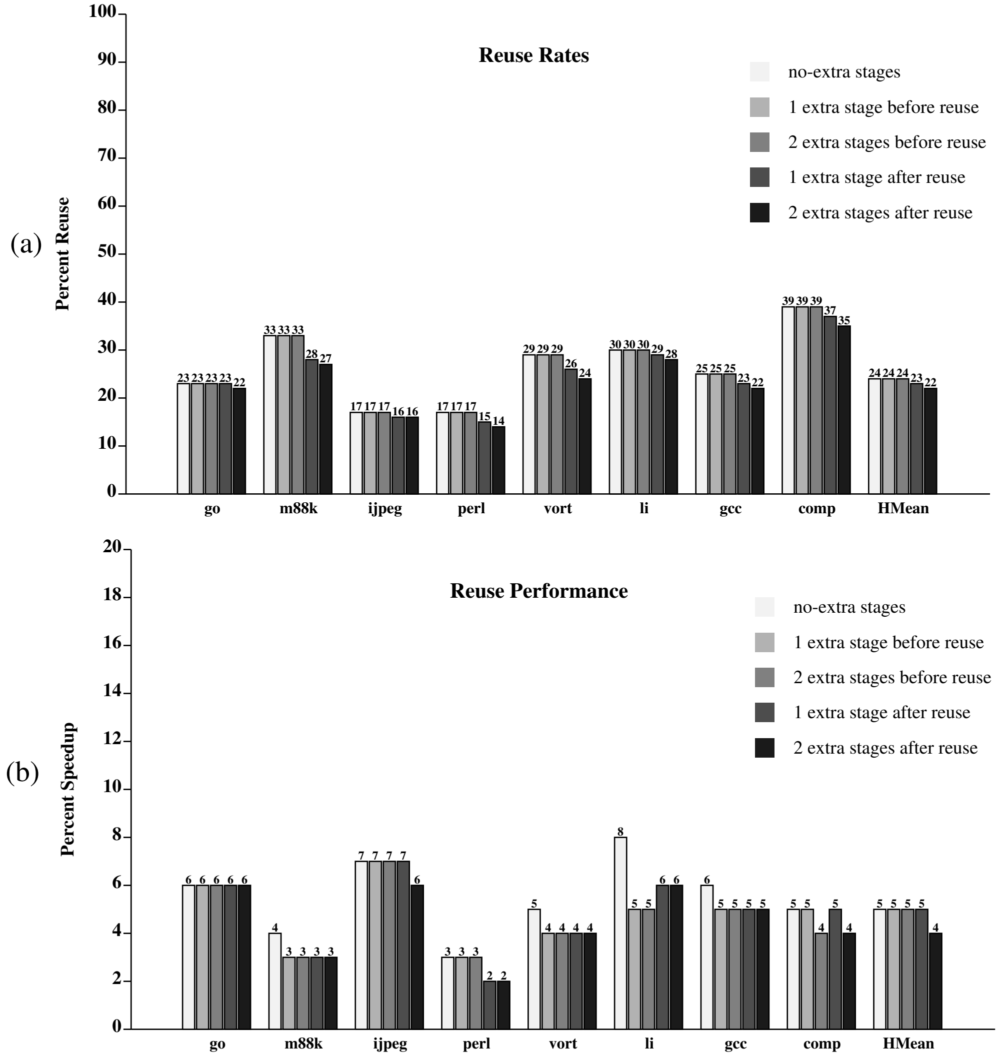


Figure 6.5 Impact of varying the pipeline length on (a) reuse rates and (b) reuse performance.

Adding the extra stages before the reuse stage has no impact on the reuse rates (Figure 6.5 (a)), as would be expected since these stages are non-stalling. Adding them after the reuse stage, on the other hand, decreases the reuse rates for all benchmarks (although by a small amount). This is because, as discussed earlier, due to this extra delay before the execution stage, more instructions don't execute soon enough to enable the reuse of their dependent instructions. However, the decrease in the reuse rate is small on average (1 to 2%-point), with the maximum decrease being 5%-points (from 33% to 28%) in the case of *m88ksim*.

In Figure 6.5 (b), we see that the variation in the pipeline length can impact the reuse performance differently for different benchmarks. For some benchmarks (e.g., *li*, and *compress*), adding stages before the reuse stage degrades the reuse performance more than adding them after the reuse stage; in others (e.g., *perl*, and *ijpeg*) the opposite is true. This variation can be attributed to the relative importance in these benchmarks of the various factors that affect reuse performance (e.g., skipping larger fraction of the pipeline, decrease in reuse rate). However, on average, the changes in reuse performance are small, suggesting that, overall, the reuse results are relatively insensitive to small changes in the pipeline length.

6.6 Branch Prediction

In all the previous experiments, we used a *gshare* predictor for branches (Table 2.1). Although we attain reasonably high prediction accuracy, gshare is not the most aggressive predictor available today; other more accurate predictors, such as hybrid predictors [50], have been developed and are commonly used today, both in the research community and in industry. In this section, we study how the reuse results might change if more accurate branch predictors are used in the underlying processor. We, first, qualitatively discuss the impact of improving the branch prediction rate on the reuse results.

6.6.1 Possible impact on reuse rates

Increase in the branch prediction rate may decrease the reuse rate because of two reasons. First, with high branch prediction rate, the pipeline will experience fewer squashes — and, hence execution will get delayed less often due to branch misprediction. As we have discussed before in this chapter (in relation to window size and pipeline width), delays in the pipeline sometimes help in reusing more instructions because they provide time for the source instructions to execute and produce the results that enable the reuse of the (delayed) dependent instructions. Since improvements in branch prediction can reduce delays in the pipeline, it may reduce the number of instructions reused. Said another way, improving the branch prediction accuracy increases the effective fetch rate, which may cause instructions to arrive at the reuse stage before their operand values are ready and, hence, may cause them to not get reused.

Second, improving the prediction rate may reduce the squash reuse component of the total reuse, because it reduces the number of misprediction squashes. This *may* result in an overall decrease in the reuse rate. (Sometimes, a reduction in squash reuse does not reduce the overall reuse rate because the instructions that would have gotten reused because of squashes get reused as part of the general reuse in the absence of squashes.)

6.6.2 Possible impact on reuse performance

Increase in branch prediction rate may both improve or hurt reuse performance. It may hurt the reuse performance for two reasons. Firstly, since fewer instructions may get reused when the branch prediction rate is improved, the reuse performance may decrease accordingly. Secondly, since squash reuse makes a significant contribution to performance improvement (as we can see from Figure 4.14), its reduction may reduce overall performance.

However, improving branch prediction may also improve reuse performance in two ways. Firstly, improving branch prediction streamlines the control flow and, hence, makes the dataflow latencies more critical to the overall execution time. Since instruction reuse reduces the dataflow latency (e.g., by collapsing dependent instructions), its performance impact may increase with the increase in branch prediction rate. Said another way, an increase in branch prediction accuracy will allow more and more reuses that were earlier rendered useless because of misprediction latencies to be useful reuses.[4] Therefore, reuse may have a higher impact on performance with improved branch prediction accuracy. Secondly, the reuse performance may improve because as the branch prediction improves, the machine can fetch deeper down the correct path and fill the instruction window with "legal" instructions. With more "legal" instructions present in the window, the available execution bandwidth may not be enough to exploit all the useful work present in the window — especially in the case of a narrow machine. Since reusing instructions can free up execution bandwidth, which can then be utilized in executing other ready instructions, reuse may have a larger impact on the performance when the branch prediction rate is improved.

6.6.3 Results

In this section, we show how the reuse results are impacted when the branch prediction rates are improved. To do this study, we run experiments with a perfect branch predictor — where all branches get predicted correctly — and compare the results with those obtained with gshare predictor. Using the perfect predictor, instead of any actual predictor, lets us see the maximum impact on reuse results of the improvements in branch prediction accuracy. We conduct the experiments for 2 pipeline widths: 4-way (our default pipeline) and 2-way. We study the 2-way pipeline to show the variations in reuse results in a pipeline where execution

4. This is analogous to why increasing the window size makes more reuses useful and, thereby, improves reuse performance as discussed in Section 6.3

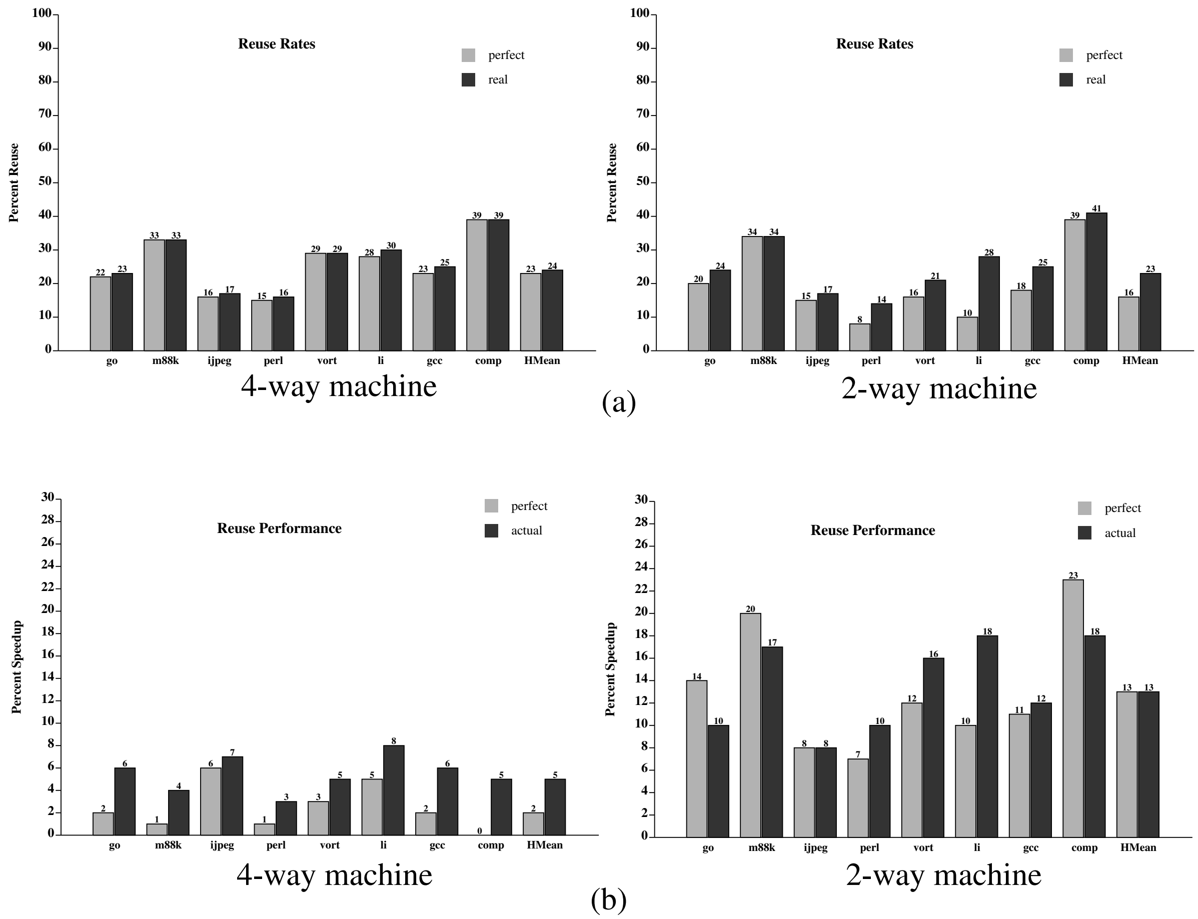


Figure 6.6 Impact of perfect branch prediction on (a) reuse rates, and (b) reuse performance. The reuse results are shown for both 4-way superscalar and 2-way superscalar machines.

bandwidth may be a bottleneck. The results are presented in Figure 6.6.

First, we discuss the impact on reuse rates (Figure 6.6 (a)). For the 4-way machine, the reuse rates are not affected appreciably by the improvements in the prediction rate. This implies that, in this pipeline, the problem of operands not being ready, which may be caused by higher branch prediction rates (as described earlier), is not very severe (because of sufficient execution bandwidth). However, we see a more pronounced decrease in the reuse rate for the 2-way machine. This is because a better branch predictor improves the effective fetch rate, and the low execution bandwidth of the machine is not able to execute instructions soon enough to make the operand values ready before the instructions needing them arrive at the reuse stage. Thus, we see that the reuse rate becomes sensitive to and decreases with branch prediction accuracy when the underlying processor is not aggressive.

Next, we discuss the impact on reuse performance (Figure 6.6(b)). We see interesting effects on reuse performance because of improvements in branch prediction accuracy. In the case of the 4-way machine, we see a marked decrease in speedups with perfect prediction. This is due to couple of reasons mentioned earlier, including the absence of squash reuse, which is a significant factor in performance (as shown in Figure 4.14). Also, the ability of reuse to free execution bandwidth is of little advantage in this case since a 4-way machine potentially has enough execution bandwidth to exploit the available ILP. However, we see a different story in the case of the 2-way machine. For several benchmarks (e.g., *go*, *m88ksim*, *compress*), the reuse performance improves significantly with perfect prediction. This is because in 2-way machine the execution bandwidth is in short supply, and the ability of reuse to free up execution bandwidth (and thereby allow other ready instructions to execute) helps it to impact the performance to a larger extent. The average reuse performances, in this case, for the perfect and actual predictor are the same. Thus, we see that, overall, the reuse performance is sensitive to branch prediction accuracy, but whether it increases or decreases with prediction accuracy depends on the aggressiveness of the underlying machine.

6.7 Memory Latency

In our simulations, we have used a L1 miss latency of 6-cycles. In this section, we study how varying this parameter may affect the reuse results. We begin with a qualitative discussion.

6.7.1 Possible impact on reuse rates

Increasing the memory latency may both improve or decrease the reuse rates. It may improve the reuse rates for the same reason decreasing the instruction window size may improve the reuse rates (Section 6.3). With higher memory latency, the instruction window may get full more often. The resulting pipeline stalls may give source instructions in the window more time to execute and prepare the operands for the dependent instructions before the latter reach the reuse stage. This may decrease the number of reuses that were missed due to operands not being ready.

However, increasing memory latency can also delay the generation of operands and, hence, can also hurt reuse rate. This may happen for instructions that depend on the load that misses in the cache. Due to the long memory latency, the dependent instructions may not have their operands ready in time and hence may miss reuse.

6.7.2 Possible impact on reuse performance

Increasing the memory latency may also either improve or decrease the reuse performance. This may happen because of several reasons. The reuse performance may increase or decrease according to whether the reuse rate increases or decreases with the increase in memory latency. However, there are other types of interactions that can take place, which may govern how the reuse performance is impacted. We mention them next.

As mentioned in the previous section, with higher memory latency, it is possible for the window to fill up more often. The ensuing stall, apart from affecting the reuse rates in the way described earlier, may also decrease the reuse performance by rendering many reuses useless. The source instructions may be reused, but the dependent instructions may be stalled; by the time the stall clears, the dependent instructions may not execute any earlier than they would have in the base case (as in the case of smaller instruction windows).

The increase in memory latency may increase the impact reuse has on the bottom line performance. This may happen in the unlikely case when the loads that miss the cache get reused. Since this would amount to short-circuiting a very long latency operation, it will improve performance significantly.

6.7.3 Results

In Figure 6.7, we show the impact of an increase of memory latency on reuse results. We perform experiments with four memory latencies (which in our case is the same as the L1 miss latencies): 6 cycles, 10 cycles, 20 cycles, and 100 cycles. (The latency of 6 cycles is the default for our baseline processor.) The size of the D-cache used in these experiments is the same as that shown in Table 2.1. The reuse rates and the reuse performance are shown in Figure 6.7 (a) and (b), respectively.

From Figure 6.7 (a), we see that for most benchmarks the reuse rates are not affected by an increase in the memory latency. On average, we see the same reuse rate for all latencies. Similarly, from Figure 6.7 (b), we see that for most benchmarks (and on average) the reuse performances are the same for the memory latencies of 6, 10, and 20 cycles. For the memory latency of 100, although we see a consistent decrease in performance for all benchmarks (for reasons discussed earlier), on average the performance is comparable to that with other latencies. This result shows that for the D-cache miss rates that we see (which are small), the reuse rates and the reuse performance are largely insensitive to memory latency — only slight changes in

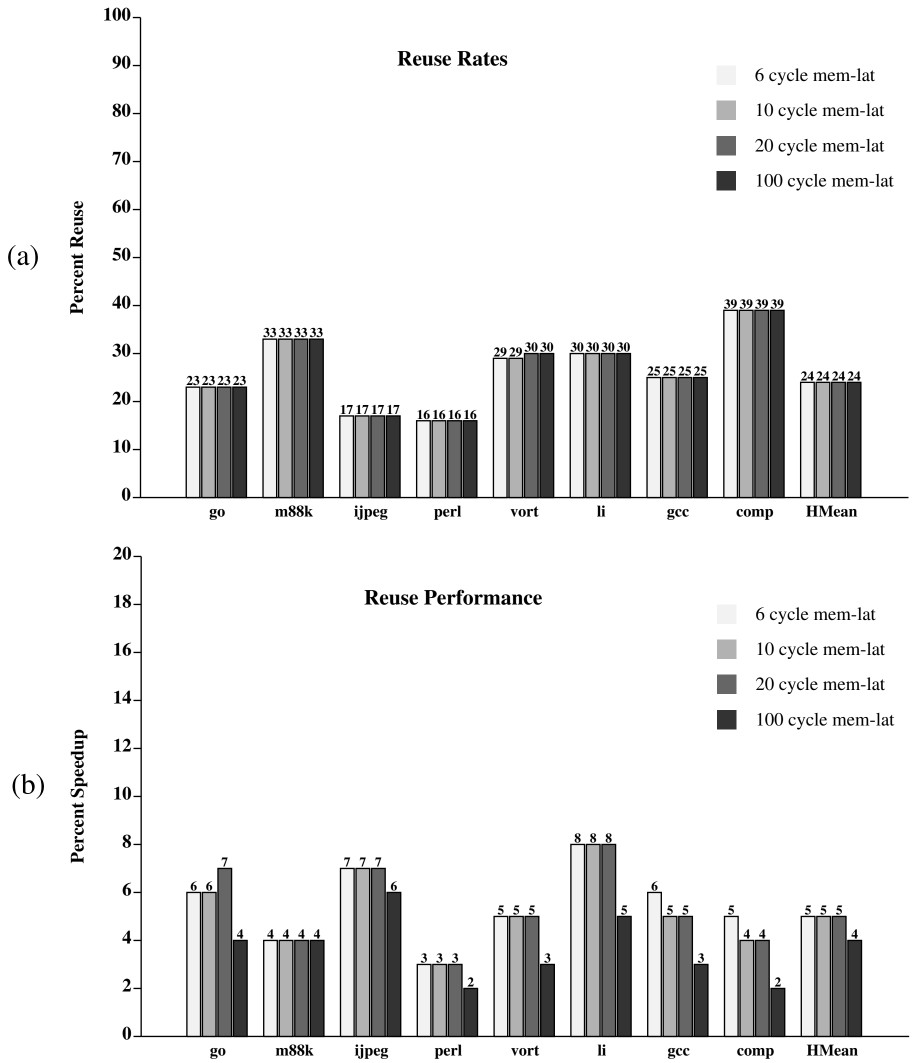


Figure 6.7 Impact of increase of memory latency on (a) reuse rates, and (b) reuse performance.

results occur by increasing the latency. But, for applications that incur higher data cache misses, the reuse results may get impacted more significantly in ways that were discussed in the previous two sections.

6.8 Reuse Latency

In all our previous simulations, we had assumed that the reuse test — the test that establishes whether an instruction can be reused — completes within a single cycle. Depending on which reuse scheme is used, it is possible that the reuse test may take multiple cycles to complete, particularly in the case of the value based schemes, S_v and S_{v+d}, which require that the values of operands be compared to determine reuse.[5] In this section, we evaluate the impact on reuse results if the latency of the reuse test — *i.e.*, *reuse latency* — is more than one cycle. However, in the next section, we first describe how IR with multiple cycle reuse latency integrates in a pipeline. Then, in Sections 6.8.2 and 6.8.3, we qualitatively discuss how reuse latency may affect reuse rates and performance. Finally, in Section 6.8.4, we present the results.

6.8.1 Pipeline with reuse latency

The pipeline with IR, shown in Figure 4.6 (page 94), can be modified for multiple-cycle reuse latencies as follows. At the read register stage, the pipeline is divided into two pipelines: the regular pipeline and the reuse pipeline. The number of stages in the reuse pipeline is the same as the reuse latency (e.g., if reuse latency is 2, the reuse pipeline will have two stages). This is based on the assumption that the reuse test can be pipelined. The first stage of the reuse pipeline overlaps with the register read stage — thus, with reuse latency of 1, the modified pipeline will be same as the original pipeline shown in Figure 4.6.

At the read register stage, the instructions are sent down both pipelines. If an instruction gets reused before it is executed in the main pipeline, then the reused results are written to the reorder buffer entry of the instruction, and are also forwarded to dependent entries in the

5. For S_{v+d}, the value comparison is needed only for the independent instructions; the dependent instructions are reused using the dependent information (Section 4.3.4, page 89).

instruction window. A reused instruction is not issued for execution in the main pipeline (if it has already been issued for execution then its writeback is ignored). If, on the other hand, the instruction completes execution before it is reused, then the reused result, if generated, is ignored.

6.8.2 Possible impact on reuse rates

Whether the reuse rates are affected by the reuse latency, depends both on the details of how the main and the reuse pipelines interact and on how the reuse test itself is partitioned among the various stages of the reuse pipeline. Conceptually, the reuse rates may improve due to the multiple-cycle latency of the reuse test because the requirement that the operand values should be ready may now need to be met latter in the pipeline. But exploiting this opportunity will require that the main pipeline be able to communicate results to the latter stages of the reuse pipeline and, therefore, will also require us to make specific assumptions about the structure of the reuse pipeline. We do not want to be so specific about the reuse pipeline structure in this study. Therefore, we only allow bypasses within the reuse pipeline from instructions that get reused to dependent instructions behind in the pipeline. With this assumption, the reuse rate that we will achieve for the multiple-cycle reuse latency case will be exactly the same as that for 1-cycle reuse latency;[6] only the reuse performance will be affected by the increased reuse latency. (We can also infer this from the results of Section 6.5, where we showed that non-stalling stages before the reuse stage will not affect reuse rate.)

---

6. We also count the reused instructions that get ignored (because the instruction result was available from the execution before the reuse pipeline) as part of total reuse.

6.8.3 Possible impact on reuse performance

Increasing the reuse latency may reduce the profitability of reuse and, hence, may decrease the reuse performance for several obvious reasons. Due to an increase in reuse latency, many instructions may get reused after they finish execution, making the reuses profitless. An increase in reuse latency may delay the execution of the instructions dependent on the reused value and, thereby, diminish the benefits of reuse. It may also degrade other benefits of reuse (such as early resolution of branches), which arise because of IR's ability to generate results early.

However, the reuse performance may be tolerant to some amount of reuse latency — *i.e.*, the performance may not degrade completely when the reuse is delayed by few cycles. Some of the reasons why this may be the case are described below. First, although the execution stage in a pipeline may logically be a few cycles away from the register read stage (2 cycles in our pipeline), it may take many more cycles for instructions to get executed from the time they get past the register read stage because of other instructions ahead of them in the window. In such cases, if the reuse latencies are short, it may be possible for instructions to get reused before they complete execution. This reuse may still improve performance. Second, sometimes, even after the reuse latency the reuse results may still become available to the dependent instructions before they can use them. Hence, the execution of the dependent instructions may not occur any later than with 1-cycle reuse latency. In such cases, an increase in reuse latency may not affect performance at all. Finally, the reuse of the dependent chain of instructions in the same cycle may still be profitable since it is likely that the time taken to execute a chain of dependent instructions is more than the reuse latency.

6.8.4 Results

In Figure 6.8, we show how increases in the reuse latencies affect the reuse performance. (We do not show the reuse rates because, as discussed earlier, they are not impacted by the increase in the reuse latency.) We present speedups for 3 reuse latencies: 1-cycle, 2-cycles, and 3-cycles. The 1-cycle reuse latency is the reuse latency that we have used in our other IR experiments so far. The speedups are shown for 4-way associative RBs with sizes ranging from 256-entries to 16k-entries.

From the figure, we see that the impact of reuse on performance is largely tolerant of small increments in reuse latencies. For many cases, we see negligible difference in the speedups for different reuse latencies (e.g., vortex for RB sizes 512 entries - 4k entries, perl for 2k entries to 4k entries). In most cases, where there is a difference in speedups, the difference is small: about 1%-point. The maximum degradation occurs in the case of *compress*, where the speedups decrease from 5% to 2%.

6.9 Summary and Conclusions

In this chapter, we studied the sensitivity of reuse results — reuse rates and reuse performance — to various processor parameters such as window size, pipeline width, pipeline length, branch prediction accuracy, memory latency, and reuse latency. We vary these parameters within reasonable limits and determine the impact that this change has on the reuse results. We also discuss qualitatively, for each parameter, why and how we might expect the reuse results

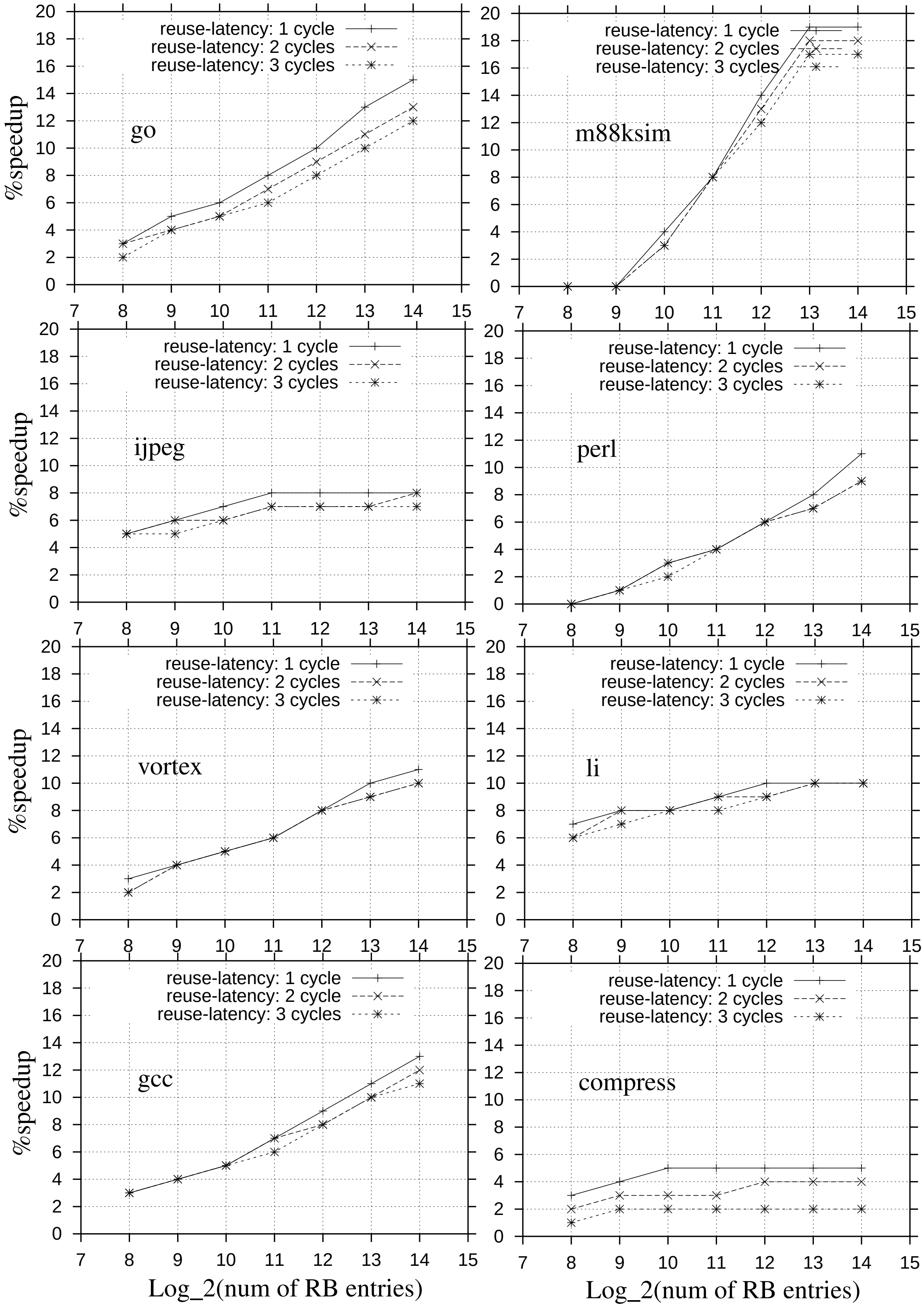


Figure 6.8 Impact of reuse latency on performance. The speedups are shown for different RB sizes (in $\log_2$(num of entries)) with 3 reuse latencies: 1 cycle, 2 cycle, and 3 cycles.

to change with variations in the parameter values. We summarize the finding for each of these parameters below.

- We simulate three window size: 32, 64, and 128 instructions. We see that the reuse results are fairly insensitive to this variation in window size: on average, the reuse rates and the reuse performance are the same for all three window sizes.
- We simulate four pipeline widths: 1-, 2-, 4-, and 8-way wide. Although we obtain comparable reuse rates for all four widths, we see that the reuse performance can vary widely depending on the pipeline width. The narrow machines (1- or 2-way) see significantly more performance improvement with IR than do the wide machines (4- or 8-way). This is because narrow machines do not have sufficient execution bandwidth to exploit all the available ILP in the window. Reusing instructions frees up execution bandwidth that can be used to execute other ready instructions. Also, the reuse performance for the 8-way machine is consistently higher than that for the 4-way machine. This is because a wider width facilitates the reuse of longer chains of instructions.
- We vary the length of the pipeline by adding 1 or 2 extra stages before or after the reuse stage. We find that the reuse rates and the reuse performance are fairly insensitive to these changes in the pipeline length: on average, we see comparable results in all cases.
- To study the effect of improving the branch prediction accuracy on the reuse results, we simulate the effects of perfect branch prediction. We find interesting changes in the reuse rates and reuse performance with the increase in the branch prediction accuracy. The reuse rates reduce with the increase in branch prediction rate for 2-way machine, but they do not change appreciably for 4-way machines. This implies that the reuse rates are sensitive to branch prediction accuracy only when the execution bandwidth is a bottleneck (since

operands don't become ready soon enough). The reuse performance, on the other hand, is sensitive to branch prediction accuracy, but whether it improves or decreases for more accurate predictors depends on the aggressiveness of the underlying machine: if the underlying machine is not aggressive then the reuse performance increases; otherwise, it decreases.

- We simulated four different memory latencies: 6, 10, 20 and 100 cycles. We see that for the small D-cache miss rates that our benchmarks experience, these changes in memory latencies do not impact the reuse results; on average, the results for all these memory latencies are comparable. However, we note that with higher D-cache misses, the variation in memory latencies may have a more pronounced impact on the reuse results.
- Finally, we simulate 3 different reuse latencies: 1-, 2-, and 3-cycles. We see that the reuse performance is largely insensitive to small changes in the reuse latency. For most programs we see no or a very small (1%-point) difference in the reuse performance with different reuse latencies.

Chapter 7

Conclusions

In this chapter, we first present a summary of this thesis and then discuss the various directions in which this work can be extended.

7.1 Thesis Summary

In this thesis, we performed two main tasks: (i) we studied a new phenomenon exhibited by programs, called *instruction repetition*; and (ii) we introduced and studied a novel microarchitectural technique, called *instruction reuse*, for exploiting that phenomenon.

The phenomenon of instruction repetition is that instructions often execute repeatedly with the same input values and produce the same results. We observed that this phenomenon is very pervasive, with the majority of dynamic instructions getting repeated for most of the benchmarks. Instruction reuse (IR) is a non-speculative technique that exploits this phenomenon to reduce the amount of work that needs to be done to execute programs and, therefore, to improve performance. It obviates the re-execution of repeating instructions by reusing their results from a hardware table, called the *Reuse Buffer (RB),* where they were stored previously. The instructions get reused early in the pipeline (in our case, in the read register stage of the pipeline), after which they skip the rest of the pipeline stages (such as, issue, execute and writeback) and become ready for retirement.

We identified two broad reasons why instructions may get repeated: (i) due to speculative execution, and (ii) due to the nature of programs themselves. The repetition due to the first reason occurs when executed instructions that are squashed due to mis-speculation are re-executed with the same input values (e.g., due to control independence). We called this form of repetition, *squash repetition*. The repetition due to the second reason occurs because of the way programs are normally written. We write programs to be *concise* (using loops), *modular* (using functions), and *generic* in nature. To support the concise and modular ways of expressing computation, programs contain many "support" instructions such as, loop-control instructions and function prologues and epilogues. These instructions often end up performing the same tasks repeatedly during execution. To make programs generic in nature, we write them so that they are capable of operating on different data values. But, when generically-written programs see the same input values repeatedly, many instructions in them end up producing the same results again and again. We called this second form of repetition, *general repetition*. The reuse engendered by these two forms of repetitions were called *squash reuse* and *general reuse*, respectively.

We outlined several reasons why IR may improve performance. First, since a reused instruction is not executed, it frees up several pipeline resources (e.g., issue ports, functional units, cache ports, etc.). These resources can then be used for executing other waiting instructions. Second, reused results become available early in the pipeline, which allows instructions dependent on them to execute sooner. Third, IR salvages useful work from mis-prediction squashes, which reduces the misprediction penalty. Fourth, IR reuses chains of dependent instruction in the same cycle. This allows instruction sequences that would have otherwise taken multiple cycles to execute, to complete in a single cycle (or *reuse-latency* number of cycles) when reused.

The study conducted in this thesis was performed using a benchmark suite consisting of 21 programs: 8 of them were SPEC '95 integer programs, 10 were SPEC '95 floating-point programs, and 3 were (self-picked) graphics programs — *Viewperf+Mesa*, *MPEG-2 decoder*, and *POV-Ray*. Viewperf is a benchmark that evaluates the performance of OPenGL® implementations (Mesa is a publicly available OpenGL implementation that we evaluated). MPEG-2 decoder plays an MPEG-2 format movie; and POV-ray is a scene renderer that uses ray-tracing technique to create 3-D images.

Next, we summarize the work we performed on instruction repetition and reuse.

7.1.1 Analysis of Instruction Repetition

We studied the phenomenon of instruction repetition elaborately, for the purposes of understanding it better. This study consisted of two parts. In the first part, we determined the statistical characteristics of the phenomenon. We collected numerous results such as, percent of dynamic instructions that get repeated, percent of static instructions that generate repeated instances, fraction of static and dynamic instructions that account for most of the repetition, number of different values with which repetition takes place, and so on. We found that there is significant repetition in programs — more that 75% of dynamic instructions are repeated for several benchmarks (e.g., 88% for *gcc*, 93% for *vortex*, 77% for *viewperf*, and 83% for *pov-ray*). We also found that a very few static and dynamic instructions contribute to most of the repetition — less than 20% of executed static instructions and less than 20% of unique dynamic instances give rise to more than 90% and 80% of total repetition, respectively.

In the second part of this study, we tracked various sources of instruction repetition to better understand its causes. For this purpose, we grouped instructions in programs in categories based on the type of data they used (e.g., program input data and immediate values) and the type of work they performed (e.g., global-address calculation and function prologue and epilogue). We then determined how the total repetition was distributed across these categories. We performed this analysis at two levels: (i) global-level, where we analyzed how repetition is distributed over whole programs; and (ii) local-level, where we analyzed how repetition is distributed with functions. We also performed a function-level analysis, where we determined the degree of repetition in the argument values of functions.

Many different types of results were presented, some which are as follows. The global analysis showed us that most of the repeated instructions used data that originated from the program internal values (immediate values) and the global initialized data — *i.e.*, from the data that is *hardwired* in program binary — and less used data that originated from the program inputs. This suggested that the phenomenon of repetition may be more a property of the program itself than of the input data. The local analysis showed that most of the repetition was due to values that originated as function arguments or global values. We also saw significant repetition due to instructions that constitute function prologues and epilogues and those that are involved in computing addresses of global loads. The function analysis showed that there is a significant amount of repetition in function arguments, with most dynamic function calls (e.g., 78% in *go*) being calls with the exact same set of argument values as previous calls to the same function.

Although we performed a very elaborate analysis, we note that this was only an initial attempt to understand this phenomenon. Our choice of instruction categories and types of analyses (global, local, etc.) was largely empirical. Better insights into the phenomenon may be gained by categorizing the instructions differently or by conducting the analysis at a different level (e.g., algorithmic level).

7.1.2 Instruction Reuse

Substantial effort in this thesis was devoted to developing the instruction reuse techniques and cultivating a better understanding of it. This work was divided into three categories: (i) devising instruction reuse schemes, (ii) studying the storage issues for instruction reuse; and (iii) investigating the different ways in which instruction reuse may interact with other microarchitectural features. We summarize these categories below.

7.1.2.1 Reuse schemes

We studied four schemes for implementing instruction reuse. All these schemes reuse instruction results from the RB by establishing that the current values of instruction operands are the same as those used to calculate the results present in the RB. However, these schemes differ in how they establish the sameness of operand values. Scheme S_v stores the operand values along with the results in the RB. To establish reusability, it compares the current values of operands with those stored in the RB. The result is reused if the values are the same. Scheme S_n stores operand register identifiers in the RB with the results. It invalidates results in the RB whose operands registers are overwritten with a new value. A result is reused if it is still valid. Schemes S_{v+d} and S_{n+d} extend the schemes S_v and S_n, respectively, with the dependence information, to facilitate the reuse of dependent instructions. The instructions in the RB are linked together according to their data dependences, with the dependent instruction pointing to the source. With this arrangement, the dependent instruction in the RB can be reused simply by establishing that their source instructions are reused. The instructions for which no dependent information is available is reused as in the base scheme S_v or S_n.

Our results showed that, in general, a significant percentage of dynamic instructions get reused — for several benchmarks (e.g., *m88ksim*, *vortex*, and *perl*) more than 50% of dynamic instructions were reused. Comparing the different reuse schemes, we saw that scheme S_v performed the best (with average reuse rates of 48% for integer benchmarks with 4096 entry RB), while scheme S_n performed the worst (the average reuse rates being 16%). Scheme S_{n+d} allowed the reuse of dependent instruction and, hence, improved the reuse rates over scheme S_n (with an average of 25%). Scheme S_{v+d} performed nearly as well as scheme S_v even with using only the dependent information to reuse the dependent instructions (with an average of 45%). We also presented other reuse characteristics such as reusability of different instruction types, and contributions of each category to total reuse. These results showed that all instructions categories are amenable to repetition; however, loads (and their address calculation micro-operation) make the largest contribution to total reuse. The speedups over the base case due to IR were not as pronounced as the reuse rates; nevertheless, they were significant — in several cases we saw more than 15% improvement in performance.

7.1.2.2 Storage issues for IR

The RB is a central hardware structure that is used in the IR technique. We studied three main parameters of this structure — size, associativity, and management policy — in greater detail. We presented the results on how reuse rates vary with size and associativity of the RB. We presented the maximum (*limit*) reuse rates for a range of RB sizes and associativities —

or, alternatively, showed the minimum RB sizes and associativities required for capturing a certain amount of reuse. The limit results showed that it is possible to capture significant amounts of reuse with a small RB, e.g., for several benchmarks, we saw that close to or more than 50% of dynamic instructions can be reused with the RB with 1K entries. One of the determinants of the degree of RB associativities is the number of instruction instances that need to be buffered to reuse a significant number of instructions. We presented the number of instances that we need to buffer to capture a certain level of reuse. The results showed that, for most benchmarks more than 70% of dynamic instructions can be reused by just buffering the last four instances. This showed that the RB need not be of a high associativity for capturing large amounts of reuse.

The limit results also showed the RB as being inefficiently utilized with the current management policies. Motivated by this result, we studied four RB management policies to utilize the RB space efficiently. Two of these policies, FnReused and FnReady, performed selective insertion in the RB, filtering our instructions that are not likely to get reused. The third policy, RR, performed selective eviction from the RB, evicting the likely unreusable instructions from the RB before the reusable ones. The fourth policy, FiF, was a novel management policy designed along the lines of the Belady's optimal management policy. For each instruction, this policy determined how far in the future that instruction is likely to get reused. Using this information, it scheduled instructions in the RB, giving priority to instructions with shorter distance values.

The success of these new policies was mixed. For some benchmarks, we saw a significant improvement in the reuse rates; for others, we saw only small improvement or, in some cases, a slight degradation in reuse rates. Overall, FiF showed potential to perform better than the other policies; however, we noted that policies FnReady and RR may be less expensive to implement and, hence, the reuse rate improvement caused by them may be noteworthy.

7.1.2.3 Sensitivity analysis

Finally, we studied the sensitivity of IR with several other microarchitectural parameters, such as instruction window size, pipeline width, pipeline length, branch prediction accuracy, memory latency, and reuse latency. We first discusses qualitatively how the IR results may vary with each of the parameters. We then conducted experiments by changing the parameters (within reasonable limits) to measure the extent of sensitivity. We saw that the reuse results were largely insensitive to the window size and pipeline length for the range in which we varied them. The reuse results were, however, quite sensitive to changes in pipeline width. For example, we saw that IR was far more effective in improving the performance for narrow machines (1- or 2-way superscalar) than for wide machines (4- or 8-way superscalar). The reuse results were also sensitive to branch prediction accuracy, but the "direction" of sensitivity depended on the width of the underlying pipeline: for narrow machines, improving the branch prediction accuracy improved reuse performance, but for wide machines, doing so decreased reuse performance. Finally, we saw that the reuse results were largely insensitive to small changes (by 1 or 2 cycles) in reuse latency. The differences in reuse performances with 1, 2, and 3 cycle reuse latency were negligible in most cases.

7.2 Future Work

This work is an initial effort in the area of instruction repetition and reuse. There is an immense potential for further research in this field. In the next several sections, we present the number of different ways in which this work can be extended further.

7.2.1 Reuse at higher granularity

In this thesis, we performed reuse at instruction-level granularity — *i.e.*, instructions were individually checked for reusability. We have seen that very often groups of instructions get reused together. The concept of reuse can be extended from instruction-level to group-level, where each group is checked for reuse as a single entity. The inputs and outputs of groups of instructions can be identified, and a group can be reused when its input values are repeated — without having to check the individual instructions within the group.

The group-level reuse may have several advantages. For example, (1) instructions within groups may be skipped altogether, *i.e.*, not fetched at all, when the groups are reused. (2) The reuse information may be stored more concisely at group-level than at instruction-level, since, only overall group information needs to be stored instead of per instruction information. (3) The group-level reuse may require less number of ports in the RB than the instruction-level reuse for reusing the same number of instructions simultaneously: a single group access may supply the same number of instructions from the RB for which several single instruction accesses may be needed.

However, there may be several issues with implementing group-level reuse. We performed some work in this area. Below, we discuss some of the issues that we discovered in the process. Later in this section we also discuss some issues in performing group reuse when the instructions within groups are not contiguous.

We implemented group-level reuse as follows. We constructed groups using instructions that were dynamically contiguous and “reusable”. An instruction was considered “reusable” if it got reused from a single-instruction RB. Each group consisted of a *starting PC*, a *following PC*, *inputs* and *outputs*. The instructions themselves were not stored in the group. The starting and the following PCs of a group were the PC of the first instruction in the group and the PC of the instruction after the group, respectively. The inputs to a group were the registers or memory locations that were read within the group without been first defined. While the outputs of a group were the registers or memory locations that were defined within the group. The groups were stored, indexed by their starting PC, in a *group reuse buffer* (GRB), from where they were reused, in the register read stage of the pipeline, when the same starting PCs were re-encountered with the same set of input values. When a group was reused, its output values were used to set appropriate registers and memory locations, and the instruction fetch was diverted to the following PC, thereby, skipping the instructions internal to the group.

We faced two main issues while studying the above group-level reuse implementation. Below, we describe these issues and some ways in which they can be addressed in future work.

- First, the problem of inputs not being ready at the reuse stage got aggravated for group reuse, resulting in our implementation of group reuse not capturing significant amount of repetition. This problem, which was not as severe for single instruction reuse (as suggested by the high reuse rates in Figure 5.1), became acute for group reuse because when reusing groups certain input values were required sooner than they would be when reusing individual instructions. We illustrate this scenario in Figure 7.1, where we show a group consisting of an instruction sequence, ‘A’, ‘B’, ‘C’. The external inputs to ‘A’ and ‘C’, ‘i’ and ‘k’, become the inputs to the group. With this group formation, the input ‘k’, which would have been required at time *t+2* for instruction reuse (Figure 7.1 (a)), would be required at time *t* for group reuse (Figure 7.1 (b)). This early requirement may thwart the group reuse, if the input is not ready at the prior time.

We suggest a couple of ways in which this problem may be tackled in future work. (1) A part of the reason why this problem arose was because we reused groups (and instructions) in fixed pipeline stages only: if inputs were not ready in those stages then the reuse was forgone. We can alleviate this problem to some extent by making the reuse “floating”, *i.e.*, by allowing groups to get reused when ever all their inputs become ready (if their reuse is still beneficial), irrespective of their position in the pipeline. (2) We may also alleviate this problem by devising more sophisticated algorithms for constructing groups that do not include those instructions in groups whose inputs are unlikely to be ready when the groups are re-encountered.

- The second issue we faced was the decrease in control prediction accuracies for several benchmarks when performing group reuse, which neutralized the performance improve-

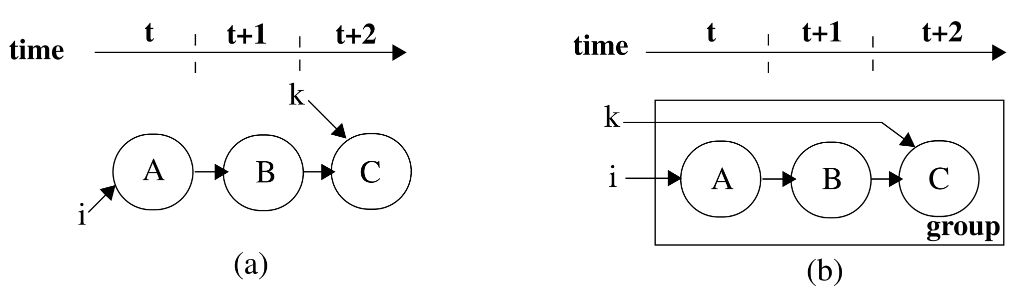


Figure 7.1 Input requirement times for an instruction sequence A, B, C: (a) when instructions are separate, (b) when instructions are in a group.

ment due to reuse. This decrease occurred because the prediction structures did not see the control instructions that were within the reused groups, since these instructions were skipped. The prediction accuracies suffered due to incomplete knowledge about the history.

This issue may be tackled in future work by storing the prediction information for control instructions in groups as part of the group representation. This information can be used to appropriately update the prediction structures when a group is reused. For example, the directions for the branches present inside a group can be stored in the group's GRB entry. This information can be used to update the branch history register when the group is reused. Similarly, the return addresses for the calls and the count of returns present in a group can also be stored with the group representation. This information can be used to update the return address stack when the group is reused: the return addresses can be pushed on to the stack, while the count of returns can be used to pop the right number of addresses off the stack.

Some more initial work on group-level reuse has also been performed by other researchers. Huang and Lilja [24] have studied performing reuse at basic-block level, while González, *et. al.* [20], have evaluated the potential of performing reuse at the dynamic instruction-trace level.

7.2.1.1 Non-contiguous-instruction group reuse

The groups may also constitute of non-contiguous instructions. One example of such groups is a chain of dependent instructions. The advantage of reusing dependent chains over reusing contiguous instruction blocks is that a contiguous block may contain unrelated and unreusable instructions that can thwart the reuse of the whole block. However, for dependent chains, since the instructions are related, they are likely to exhibit same degree of reusability, increasing the chances of the reusing the whole chain. However, a problem with the non-contiguous group reuse may be the merging of the reused results back into the main pipeline. Unlike in the case of contiguous instructions, all results cannot be merged immediately, since there may be write-after-write hazards with the intermediate instructions not part of the group. Some recent work [38] has shown how results from one computing subsystem can be integrated into another using a technique that is similar in spirit to scheme S_{n+d}; such a technique can also be used to integrate the reused results from a non-contiguous group into the main pipeline.

7.2.2 Compiler support for reuse

Compiler may be able to play an important role in making the reuse technology more practical. It can do so in several ways. First, it may help manage the RB efficiently, and thereby, help make the RB small. It can identify the likely reusable instruction and pass this knowledge as hints to the processor. These hints can then be used to choose which instructions to place in RB. This way we may be able to reduce RB pollution and to achieve high reuse rates from a small size RB.

Similarly, compiler may facilitate group-level reuse by identifying (or by creating) reusable groups of instructions for the hardware to exploit. Some work has started in this area. In [16], compiler creates regions of code that are likely reusable. The reuse of these regions in this work is software controlled, the results of these regions are saved and reuse by instructions inserted in the code by the compiler.

7.2.3 Low power

IR reduces the amount of work that needs to be performed for executing programs. For example, the reused instructions don't have to flow through many stages of the pipeline (e.g., issue, execute, and writeback), or the reused loads don't access the data cache. Due to this work savings, IR may have the potential to reduce the amount of power consumed in processors (if, of course, the act of obtaining the results from the RB itself does not consume substantially more power). Further investigations are required in this area.

7.2.4 Other uses of IR

IR provides the ability to save work and reuse it later on. This ability can be put to several other uses, some of which are described below.

The I-cache misses limit performance significantly, particularly because they cannot be overlapped. IR can be used to perform some useful work while waiting for the missed line to return. The instructions following the missed line can be fetched (if they are in the cache) and executed assuming no data dependence from the missed line. These instructions are treated differently than the normal instructions. They are neither inserted in the reorder buffer nor renamed, and their results after execution are stored in the RB. If, later, the control reaches these instructions (after the instruction fetch resumes), they can be reused if their results in the RB are found to be valid. This approach can also be used for performing potentially useful work when the processor is stalled for other reasons, such as when the instruction window becomes full.

Recently, many researchers have started working on using separate threads in processors to optimize the performance of a main thread [13, 38]. One optimization that is being studied is to let a subordinate thread run ahead and execute the performance-hurting events (such as, cache misses and branch mis-prediction) in advance and communicate the results to the main thread. The communication between the subordinate thread and the main thread can be provided using IR-like scheme. The subordinate threads may write the results in an RB, while the main thread may selectively use (appropriate) results from the RB after checking the operands.

7.2.5 Further developing the FiF policy

The FiF policy, as we have mentioned earlier, is a general policy in that it can be used for managing other forms of storage as well. Its use in managing the cache hierarchy can be particularly interesting and warrants further investigation.

Bibliography

[1] OpenGL. *http://www.opengl.org/*.

[2] OpenGL Performance Benchmarks – Viewperf. *http://www.specbench.org/gpc/opc.static/vp50.html*.

[3] Persistence of Vision(tm) Ray-Tracer. *http://www.povray.org*.

[4] SPEC CPU95 Benchmarks. *http://www.specbench.org/osg/cpu95/*.

[5] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[6] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, Effective Dynamic Compilation. In *Symposium on Programming Language Design and Implementation*, pages 149–159, May 1996.

[7] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[8] Jon Louis Bentley. *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[9] Ratislav Bodik. *Path-Sensitive Value-Flow Optimizations of Programs*. Ph.D. thesis, University of Pittsburg, November 1999.

[10] Ratislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete Removal of Redundant Expressions. In *Proceeding of the SIGPLAN '98 Conference on Plrogramming Language Design and Implementation (PLDI)*, pages 1–14, June 1998.

[11] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996.

[12] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Proc. of 30th Annual international Symposium on Microarchitecture (MICRO-30)*, December 1997.

[13] R. Chappel, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 186–195, May 1999.

[14] Yuan Chou, Jason Fung, and John Paul Shen. Reducing Branch Misprediction Penalties Via Dynamic Control Independence Detection. In *Proceeding of the Internation Conference on Supercomputing(ICS)*, June 1999.

[15] Daniel Citron, Dror Feitelson, and Larry Rudolph. Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units. In *Proc. of 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–261, October 1998.

[16] Daniel A. Connors and Wen mei W. Hwu. Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results. In *Proc. of the 32nd International Symposium on Microarchitecture (MICRO)*, November 1999.

[17] C. Consel and F. Noel. A General Approach for Run-time Specialization and its Application to C. In *Symposium on Principles of Programming Languages*, pages 145–156, January 1996.

[18] Freddy Gabbay and Avi Mendelson. Speculative Execution based on Value Prediction. Technical Report EE Department TR 1080, Technion - Israel Institute of Technology, November 1996.

[19] Freddy Gabbay and Avi Mendelson. Using Value Prediction to Increase the Power of Speculative Execution Hardware. *ACM Transaction on Computer Systems (TOCS)*, August 1998.

[20] Antonio Gonzalez, Jordi Tubella, and Carlos Molina. Trace-Level Reuse. In *Proc. of Internation Conference on Parallel Processing(ICPP)*, September 1999.

[21] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-Directed Run-Time Specialization in C. In *Proc. of Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 163–178, June 1997.

[22] Samuel P. Harbison. *A Computer Architecture for the Dynamic Optimization of High-Level Language Programs*. Ph.D. thesis, Carnegie Mellon University, September 1980.

[23] Samuel P. Harbison. An architectural alternative to optimizing compilers. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 57–65, March 1982.

[24] Jian Huang and David J. Lilja. Exploiting Basic Block Value Locality with Block Reuse. In *Proc. of 5rd Annual International Symposium on High-Performance Computer Architecture*, January 1999.

[25] Gerry Kane. *MIPS R2000/R3000 RISC Architecture*. Prentice Hall, 1987.

[26] Mikko H. Lipasti and John P. Shen. Exceeding the Dataflow Limit Via Value Prediction. In *Proc. of 29th International Symposium on Microarchitecture*, pages 226–237, December 1996.

[27] Mikko H. Lipasti, Christopher B. Wilkerson, and John P. Shen. Value Locality and Load Value Prediction. In *Proc. of 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, September 1996.

[28] Scott MacFarling. Combining Branch Predictors. Technical Report TN-36, WRL, June 1993.

[29] D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[30] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1992.

[31] Carlos Molina, Antonio Gonzalez, and Jordi Tubella. Dynamic Removal of Redundant Computations. In *Proc. of Internation Conference on Supercomputing(ICS)*, June 1999.

[32] E. Morel and C. Renviose. Global optimizations by suppression of partial redundancies. *Communications of the ACM (CACM)*, 22(2):96–103, 1979.

[33] MPEG Software Simulation Group. MPEG-2 Encoder/Decoder. *http://www.mpeg.org/MSSG/*.

[34] Stuart F. Oberman and Michael J. Flynn. On Division and Reciprocal Caches. Technical Report CSL-TR-95-666, Stanford University, April 1995.

[35] Brian Paul. Mesa Library. *http://mesa3d.sourceforge.net/*.

[36] Stephen E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical Report SMLI TR-92-1, Sun Microsystems Laboratories, September 1992.

[37] Stephen E. Richardson. Exploiting Trivial and Redundant Computation. In *Proc. of the 11th Symposium on Computer Arithmetic*, pages 220–227, July 1993.

[38] Amir Roth and Gurindar S. Sohi. Speculative Data-Driven Sequencing for Imperative Programs. Technical Report UW CS TR #1411, University of Wisconsin-Madison, February 2000.

[39] Yiannakis Sazeides and James E. Smith. The Predictability of Data Values. In *Proc. of 30th Annual international Symposium on Microarchitecture (MICRO-30)*, pages 248–258, December 1997.

[40] Yiannakis Sazeides and James E. Smith. Modeling Program Predictability. In *Proc. of 25th Annual International Symposium on Computer Architecture (ISCA)*, pages 73–85, July 1998.

[41] J. E. Smith. Decoupled Access/Execute Computer Architecture. In *Proc. of the 9th Annual International Symposium on Computer Architecture*, pages 112–119, April 1982.

[42] J.E. Smith and A.R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[43] Avinash Sodani and Gurindar S. Sohi. Dynamic Instruction Reuse. In *Proc. of 24th Annual International Symposium on Computer Architecture*, pages 194–205, July 1997.

[44] Avinash Sodani and Gurindar S. Sohi. An Empirical Analysis of Instruction Repetition. In *Proc. of 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[45] Avinash Sodani and Gurindar S. Sohi. Understanding the Differences Between Value Prediction and Instruction Reuse. In *Proc. of 29th International Symposium on Microarchitecture*, pages 205–215, December 1998.

[46] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39-3:349–359, March 1990.

[47] L. Sterling and E. Shapiro. *The Art of Prolog, 2nd Ed.*. MIT Press, Cambridge, MA, 1992.

[48] Kai Wang and Manoj Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. In *Proc. of 30th Annual international Symposium on Microarchitecture (MICRO-30)*, pages 281–290, December 1997.

[49] N. Weste and K Eshraghian. *Principles of CMOS VLSI Design, 2nd edition*. Addison-Wesley Publishing Company, 1993.

[50] T. Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive training branch prediction. In *Proc. 19th Annual International Symposium on Computer Architecture*, pages 124–134, May 1992.

Appendix A

Additional Results

A.1 Repetition Results with second set of inputs

We show the repetition rates and the global analysis results for the second set of inputs (shown in Table 2.5) in Table A.1 and Table A.2, respectively

A.2 Additional squash reuse results

We show the breakdown of percentage of instruction reused and total performance improvement in terms of general and squash reuse. In Figure A.1, we present the breakdown for the SpecInt '95 and the graphics benchmarks for schemes S_v and S_n. In Figure A.2, we show the breakdowns for the SpecFP '95 benchmarks for schemes S_v, S_n, and S_{n+d}. The numbers for scheme S_{v+d} match those of scheme S_v closely, and hence are not shown separately.

SpecInt '95	Dynamic Instructions		SpecFP '95	Dynamic Instructions	
	Total (millions)	Repeat (%)		Total (millions)	Repeat (%)
go	1000	94.1	tomcatv	1000	56.0
m88ksim	100.1	94.9	swim	1000	25.1
ijpeg	1000	74.6	su2cor	729.4	47.8
perl	12,396	97.7	hydro2d	623.0	43.9
vortex	1000	95.2	mgrid	1000	16.2
li	1000	85.6	applu	1000	52.6
gcc	400	89.4	turb3d	1000	90.0
compress	1000	51.8	apsi	212.8	68.7
Graphics			fpppp	226.4	37.4
Viewperf+Mesa	485.8	83.5	wave5	826.1	36.7
Mpeg-2 decoder	38.1	69.0			
POV-Ray	1000	81.5			

Table A.1 Total number of dynamic instructions executed and percentage of them repeated with the second set of benchmark inputs (Table 2.5). Most results tally very well with the results with the first set of inputs, shown in Table 3.1 (except of *wave5*, where the repetition rates are lower due to the limited per instruction buffering available in our repetition tracking buffer).

Categories	go	m88k	ijpeg	perl	vort	li	gcc	comp
Overall	% of all dynamic instructions							
internals	80.2	42.6	58.3	58.7	54.2	46.1	57.9	63.3
global init data	19.3	27.9	22.4	30.2	28.3	12.4	25.7	29.3
external input	0.5	29.3	19.4	8.0	17.5	40.7	16.4	7.4
uninit	0.0	0.2	0.0	3.1	0.0	0.8	0.1	0.0
Repeated	% of all repeated dynamic instructions							
internals	80.5	42.8	57.3	60.1	55.2	47.3	61.4	77.4
global init data	19.0	26.3	24.4	30.9	28.8	14.4	28.4	21.2
external input	0.4	30.7	18.3	5.8	15.9	37.4	10.1	1.4
uninit	0.0	0.2	0.0	3.2	0.0	0.9	0.1	0.0
Propensity	% of all dynamic instructions in each category							
internals	94.5	95.4	73.3	99.9	96.9	87.8	94.8	63.3
global init data	92.7	89.4	81.5	99.9	97.0	99.6	98.8	37.4
external input	90.0	99.3	70.58	71.8	86.7	78.5	55.3	9.7
uninit	0.0	100.0	0.0	99.9	0.0	99.8	100.0	0.0

(SpecInt)

Categories	tomcatv	swim	su2cor	hydro2d	mgrid	applu	turb3d	apsi	fpppp	wave5
Overall	% of all dynamic instructions									
internals	37.2	33.6	23.0	43.6	4.7	24.6	16.4	9.7	8.2	17.7
global init data	37.8	6.3	24.7	16.9	95.3	18.2	83.5	9.1	13.0	15.8
external input	24.6	59.9	51.9	39.3	0.0	57.2	0.0	80.9	78.8	66.5
uninit	0.5	0.2	0.4	0.2	0.0	0.0	0.1	0.3	0.0	0.0
Repeated	% of all repeated dynamic instructions									
internals	58.4	47.0	31.2	37.3	13.9	33.6	13.6	13.3	21.9	39.8
global init data	30.8	1.9	28.8	20.0	86.1	26.8	86.4	11.4	34.5	12.2
external input	10.8	51.2	40.0	42.7	0.0	39.6	0.0	74.9	43.6	48.0
uninit	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.4	0.0	0.0
Propensity	% of all dynamic instructions in each category									
internals	87.9	35.2	64.9	37.6	48.3	71.7	74.3	93.9	99.6	82.3
global init data	45.6	7.5	55.8	51.9	14.6	77.5	93.0	86.2	99.2	28.4
external input	24.7	21.5	36.9	47.7	0.0	36.4	0.0	63.6	20.7	26.5
uninit	0.0	0.0	0.0	2.3	0.0	0.0	89.9	100.0	0.0	0.0

(SpecFP)

Table A.2 Global analysis results for the second set of inputs (Graphics benchmarks on the next page).

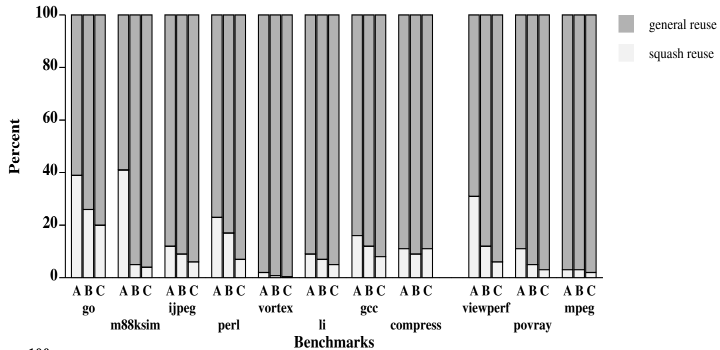
Categories	viewperf	mpeg-2	povray
Overall	% of all dynamic instructions		
internals	29.8	60.3	26.4
global init data	12.6	11.6	24.2
external input	56.8	28.1	47.2
uninit	0.8	0.1	2.1
Repeated	% of all repeated dynamic instructions		
internals	34.6	60.6	30.4
global init data	15.0	16.7	29.7
external input	49.5	22.6	37.4
uninit	0.9	0.1	2.6
Propensity	% of all dynamic instructions in each category		
internals	97.0	69.3	93.7
global init data	99.2	99.6	99.8
external input	72.7	55.6	64.5
uninit	100.0	92.0	99.8

(Graphics)

Table A.2 (contd) Global analysis results for the second set of inputs for graphics benchmarks

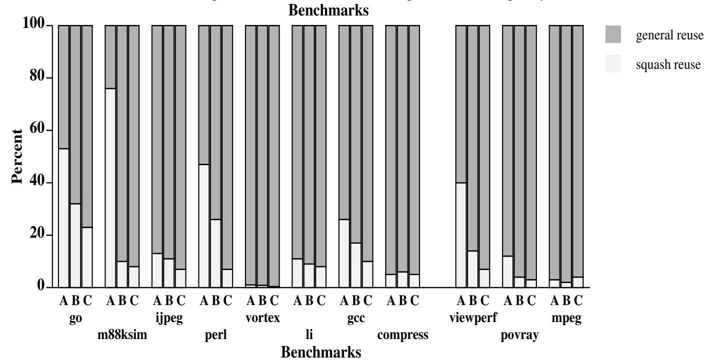
Integer Benchmarks

Breakdown of % reuse

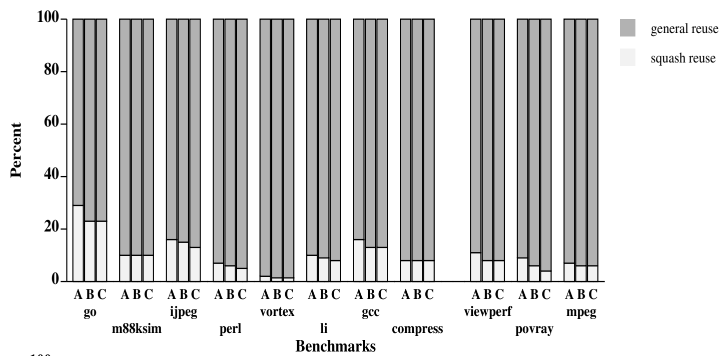


Scheme Sv

Breakdown of performance



Breakdown of % reuse



Scheme Sn

Breakdown of performance

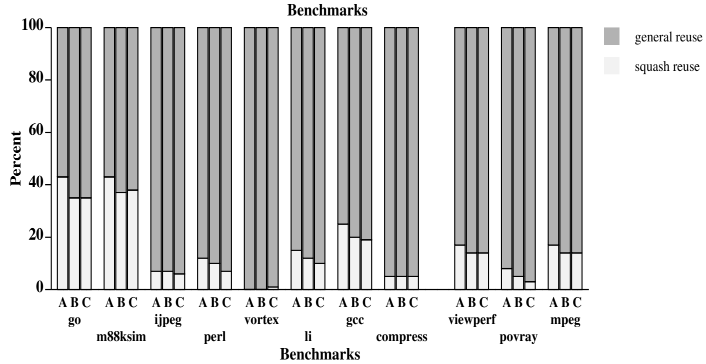


Figure A.1 Breakdown of percentage instruction reused and performance in terms of *general* and *squash* reuse for schemes Sv and Sn. Bar 'A' is for a 256-entry RB, 'B' is for a 1k-entry RB, and 'C' is for a 4k-entry RB.

Floating Point Benchmarks

Scheme S_v

Breakdown of % reuse

Breakdown of performance

Scheme S_n

Breakdown of % reuse

Breakdown of performance

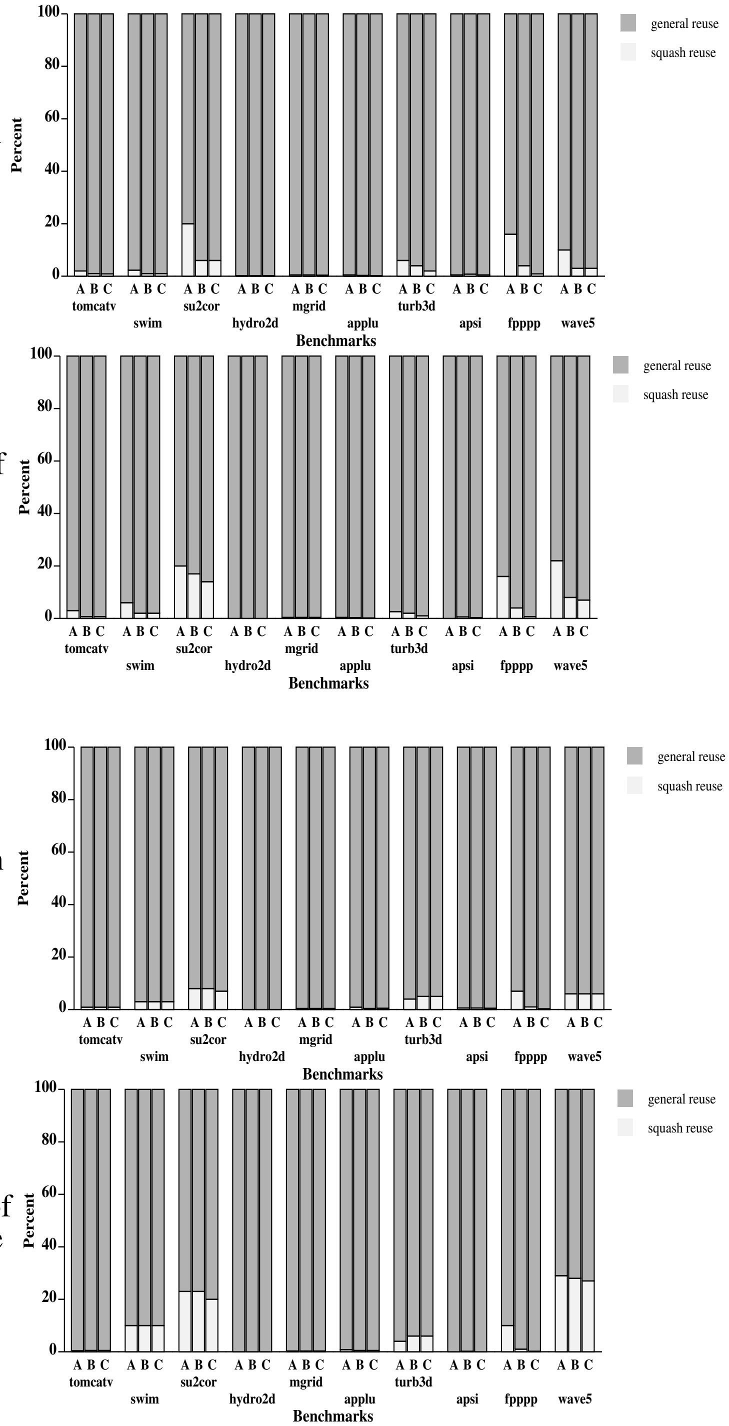
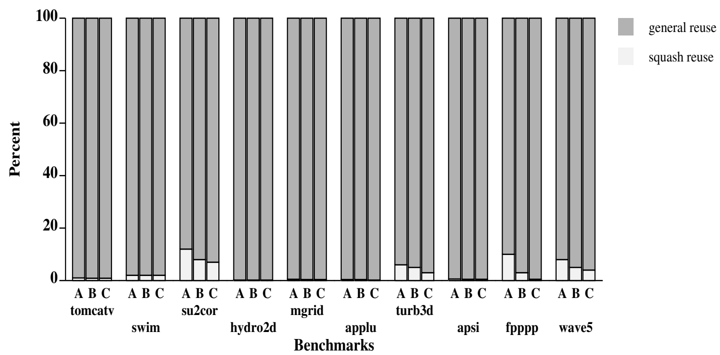


Figure A.2 Breakdown of percentage instruction reused and performance in terms of *general* and *squash* reuse for schemes S_v, and S_n. Bar 'A' is for a 256-entry RB, 'B' is for a 1k-entry RB, and 'C' is for a 4k-entry RB.

Floating Point Benchmarks

Breakdown of % reuse



Scheme S_{n+d}

Breakdown of performance

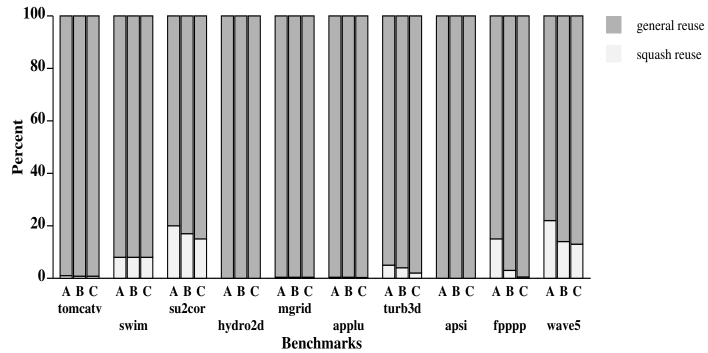


Figure A.2 (continued) Breakdown of percentage instruction reused and performance in terms of *general* and *squash* reuse for scheme S_{n+d}. Bar 'A' is for a 256-entry RB, 'B' is for a 1k-entry RB, and 'C' is for a 4k-entry RB