Dynamic Instruction Reuse

Avinash Sodani and Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706 USA
{sodani,sohi}@cs.wisc.edu

Abstract

This paper introduces the concept of dynamic instruction reuse. Empirical observations suggest that many instructions, and groups of instructions, having the same inputs, are executed dynamically. Such instructions do not have to be executed repeatedly — their results can be obtained from a buffer where they were saved previously. This paper presents three hardware schemes for exploiting the phenomenon of dynamic instruction reuse, and evaluates their effectiveness using execution-driven simulation. We find that in some cases over 50% of the instructions can be reused. The speedups so obtained, though less striking than the percentage of instructions reused, are still quite significant.

1 Introduction

There are three parameters that influence the execution time of a program. Microarchitecture has concentrated on two of them: (i) the number of instructions executed per clock cycle, i.e., the IPC, and (ii) the clock cycle time. The third parameter: (iii) the total number of instructions, has been considered the domain of software. In this paper we address the following question: "Can we develop microarchitectural techniques to reduce the number of instructions that have to be executed dynamically, and what are the potential benefits of such techniques?"

Just as caches reduce the number of memory accesses made dynamically if a memory location is going to be accessed repeatedly, the number of instructions executed dynamically can be reduced if an instruction is going to produce the same value repeatedly. We have observed many instructions, and groups of instructions, having the same inputs (consequently producing the same output) when executed dynamically. This observation can be exploited to reduce the number of instructions executed dynamically as follows: by buffering the previous result of the instruction, future dynamic instances of the same static instruction can use the result by establishing that the input operands in both cases are the same. We call this *dynamic instruction reuse.*

Dynamic instruction reuse can benefit performance in two main ways. First, by not having to pass through all the phases of execution (e.g., issue, execute, result bypass) dynamically, utilization of machine resources could be reduced, alleviating resource conflicts. Second, and more important, the outcome of an instruction can be known much earlier, allowing instructions that are dependent upon the outcome to proceed sooner. As we shall see, the results of chains of dependent instructions could all be generated in a single cycle, short-circuiting dependence chains, and reducing the lengths of critical paths of execution.

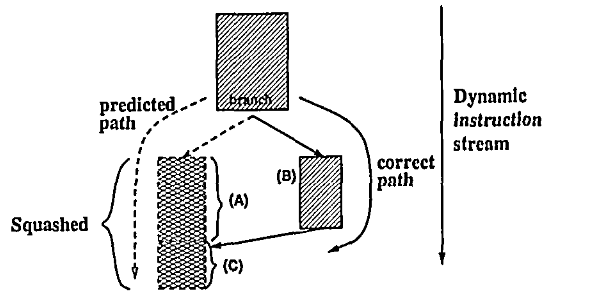


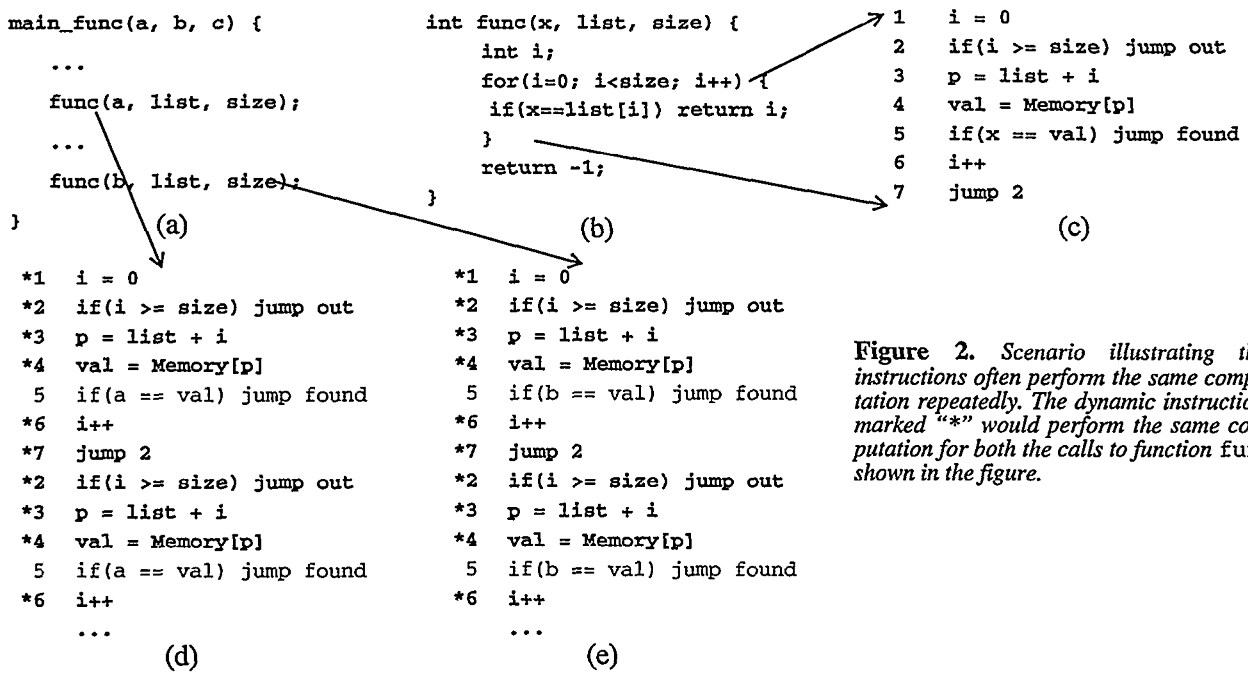
Figure 1. *Scenario where execution on the (mis)predicted path converges with the execution on the correct path. In such cases certain instructions from part (C) need not be re-executed when encountered on the correct path.*

This paper is concerned with exploiting the phenomenon of dynamic instruction reuse. Towards this end, we develop microarchitectural mechanisms that allow the outcome of an instruction to be known earlier than it would had the instruction have to pass through all the phases of its execution. We describe the concept of dynamic instruction reuse in section 2, and present scenarios to illustrate why it occurs and why might it be a useful phenomenon to exploit. In section 3, we present three different schemes for instruction reuse. Each scheme employs a *reuse buffer*, a buffer of previous outcomes of instruction execution. In section 4 we show how a reuse buffer can be incorporated into a generic superscalar processor. In section 5 we provide a quantitative evaluation, and in section 6 we discuss related work. Finally section 7 presents some concluding remarks.

2 Scenarios for Dynamic Instruction Reuse

Before developing mechanisms to allow dynamic instruction reuse, we need to understand why this phenomenon occurs. What causes instructions to be executed with the same input operands? Why are such instructions in the program in the first place? Are such instructions needed? Why might it be better to obtain the outcome of an instruction from a buffer rather than recompute it? In order to answer these questions, we look at a couple of scenarios

The first scenario involves speculative execution in a dynamically scheduled processor. As illustrated in Figure 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



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

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

```
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
```

```
*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++
     ...
```

```
*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++
     ...
```

Figure 2. *Scenario illustrating that instructions often perform the same computation repeatedly. The dynamic instructions marked "*" would perform the same computation for both the calls to function* `func` *shown in the figure.*

block A, the processor may execute instructions from another block (C), which is control independent of the branch. If the branch was mispredicted, instructions executed from both blocks A and C are discarded, execution resumes at block B, from where it proceeds 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 re-executed. If the results of such instructions were buffered, and we could detect that their operands values are the same, their results could be reused, thereby reducing the squash penalty. We term this scenario as *squash reuse.*

Our initial goal for developing an instruction reuse mechanism was to reduce the branch mis-prediction penalty, especially for short forward branches, as described above. However, when we were studying the effectiveness of the mechanisms that we developed for the above case, we discovered that the concept was much more powerful. Other scenarios where instructions can be reused dynamically also arise frequently. These scenarios are a result of two important artifacts of how the dynamic computation is expressed statically. First, programs are written to be generic, i.e., operate upon a variety of input data sets — we don't write programs that operate on only a single input data set. Starting out with a single static program, different inputs will cause different instructions to be executed dynamically, resulting in a different dynamic operation stream for each input data set. For a particular input set, operands for some of the instructions may not change dynamically (they would in another execution with a different input data set). Second, programs are written to express a desired computation in a concise manner. For example, the computation to carry out operations on each element of a 1000-element data structure is expressed using a loop structure; we don't write separate instructions for each element of the data structure. In the process of recreating the dynamic sequence of operations from the static representation, many operations may be repeated, as we shall see shortly.

The above situations are best illustrated by an example. Consider the example of Figure 2. The function `func` searches for a value 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`, exiting when the value is found. Instructions corresponding to the loop in `func` are shown in Figure 2(b). Figure 2(d) shows the dynamic instances of these instructions which are generated by the first call to `func`. In each iteration of the loop, the instruction 2 is dependent upon the `size` parameter, the instructions 3 and 4 are dependent upon the `list` parameter, instruction 5 is dependent upon both 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 2(e)) on the same `list` (and same `size`), but with a different search key, then all the different dynamic instances of instructions 1-4 and 6 produce the same outcomes as they did the last time the function was called (a total of `size` dynamic instances of instructions 2-4 and 6). Only the dynamic instances of instruction 5 produce results that might be different from what they were in the previous call to `func`. This "reuse" 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.)

How might performance benefit if we buffered the (`size`) dynamic instances of instructions 1-4 and 6 in the above example, and reused them? First, the dynamic instances of instructions 1-4 and 6 do not have to pass through all the different phases of execution (ALU, issue, result bus, etc.), thereby reducing the demand for processor resources. (In the above case, accesses to the data cache are also eliminated — these 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 reduced considerably. Without dynamic instruction reuse, the critical path through the computation, as expressed above, would be `size+3` steps (assuming that the loop executes all `size` iterations), `size` steps to generate all the dynamic instances for the induction variable `i`, plus 3 steps to execute 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. With instruction reuse, in the best case, the critical path, i.e., the height of the dataflow graph through the computation, is reduced to only 1 step. 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. In other words, dynamic instruction reuse allows us to *exceed the dataflow limit* that is "inherent" in the program. Of course, in an actual execution other constraints will prevent us from achieving the dataflow limit in either case, but concentrating on the dataflow limit illustrates the potential power of the concept. We call this second scenario *general reuse.*

The above example shows that the potential for instructions to be reused dynamically exists. As we shall see in section 5, in some cases over 50% of all executed instructions produce results that they produced earlier, suggesting a need to exploit the phenomenon. Our objective is to develop dynamic techniques to exploit repetitive behavior of the above type. While not impossible, doing the same statically in the compiler would require a tremendous (and very likely impractical) effort in the above case: constant propagation, function in-lining, loop unrolling `size` times, common sub-expression elimination, all carried out globally (and possibly inter-procedurally), sufficient registers to store `size` elements, as well as alias analysis to allow register allocation of the list elements. In the above example, to achieve the same effect as dynamic instruction reuse, the compiler would essentially have to end up putting the `size` elements of list in registers, and in-lining `func` as a sequence of `size` static instructions, each of which compares value with a register. This is a tall order, given the current state of the art. Accordingly we concentrate on developing dynamic schemes for instruction reuse.

3 Schemes for Instruction Reuse

In this section, we describe three hardware schemes to implement dynamic instruction reuse. To reuse an instruction we need to determine that its outcome is going to be the same as a previous outcome, and reuse the previous outcome. The reuse schemes described in this section implement this determination in different ways. In each scheme we store the result(s) of a previously-executed instruction in a hardware structure called *Reuse Buffer (RB)* (Figure 3).[1] When an instruction is encountered, the RB is queried to see if it contains a reusable result for the instruction. Three 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. The RB could 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 no 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.[2] Maintaining the consistency of information in the RB depends upon the reuse scheme, as we see shortly.

Next, we present details of three schemes for reusing instructions. 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 scheme (S_{n+d}) tracks dependence relationships among the instructions. 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?

In practice, the reuse schemes would be implemented in a variety of different ways. In this paper we concentrate on the functionalities required by each reuse scheme instead of their implementations.

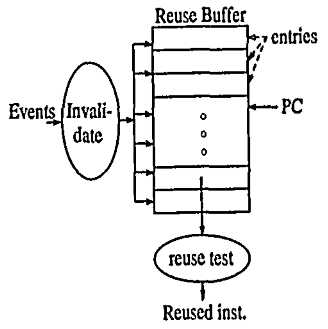


Figure 3. *Generic Reuse Buffer. It is indexed by the PC and it has mechanism for selectively invalidating entries based on some event.*

1. Depending upon the buffer mapping and management policy, the RB could contain the outcomes of many previous dynamic instances of the same instruction. For example, in Figure 2, the buffer could contain all the `size` dynamic instances of the instruction updating the induction variable, with each dynamic instance producing a different value.
2. In this paper, we do not explore this aspect of the problem — in our discussions and experiments we assume that all recently executed instructions are placed in the RB.

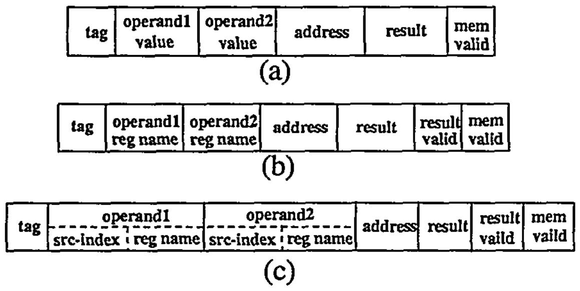


Figure 4. *RB entry (a) Scheme S_v (b) Scheme S_n (c) Scheme S_{n+d}*

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 values, 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, then 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 RB for this scheme is shown in Figure 4(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 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 and 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.

Note that the *address* field, *memvalid* field, and the associative search for invalidations are required only to maintain the integrity of load values. The RB can be split into two buffers: one for storing load values and another, the main RB, for storing everything except the load values (including entries for load addresses). This RB organization has two advantages: first, *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 invalidation logic, this logic would only be present in the buffer for load values, which probably would be much smaller compared to the main RB. Nevertheless, since our goal is to demonstrate the potential of instruction reuse, and not to compare the merits of different implementations, we assume a unified RB for our experiments presented in this paper.

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. 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:

- **RB entry:** An RB entry for this scheme is shown in Figure 4(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.

- **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 which match are marked invalid (*resultvalid* bit is reset).

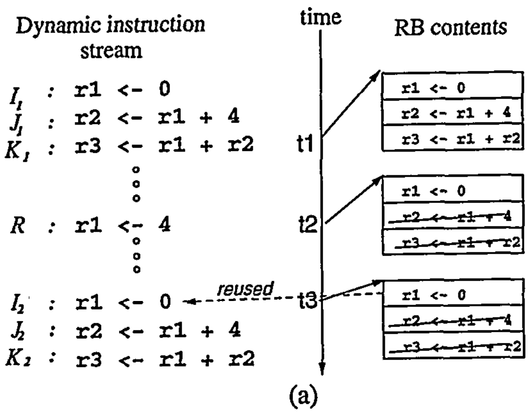
Note that the effect of invalidations (which is to purge stale results) can be obtained in other ways too, e.g., using *timestamps* for the operands. As mentioned earlier, in this paper we focus on the required functionality and leave the task of exploring different implementations as a future work.

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 in this scheme the reuse status of an instruction in the RB is established based on its operand names and/or its dependence information, we call this scheme S_{n+d} (the letters 'n' and 'd' stand for *name* and *dependence* respectively).

Figure 5(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 5(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 could be established simply by establishing the 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



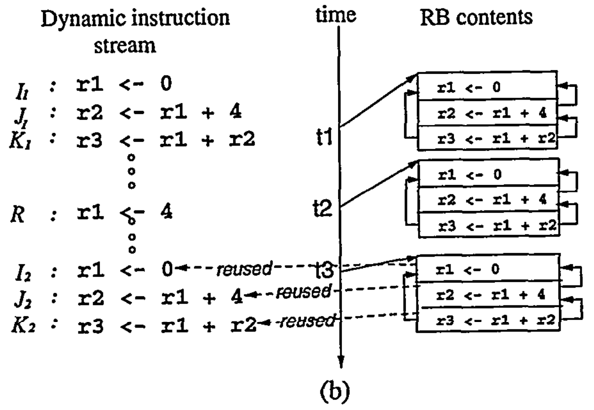


Figure 5. *Dependent sequence of instructions (a) not handled in Scheme S_n, but (b) handled in Scheme S_{n+d}.*

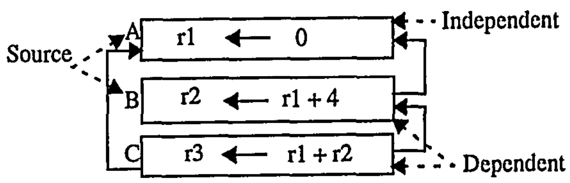


Figure 6. *Instructions with data dependence links. The arrows point from the instruction using the value to the instruction producing the value.*

(illustrated in Figure 6). 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 the *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 the instruction which is the latest producer of a register is not in the RB, then the RST entry for that register is set to invalid. 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 now present details of this scheme's operation.

- **RB entry:** An RB entry (shown in Figure 4(c)) is similar to that of 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 7).

- **State updates:** As in schemes S_v and S_n, stores invalidate the 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 RB, i.e., when the dependence information is lost.[3] To perform this operation the RB needs to be searched for entries whose *src-index* field matches the index (in RB) of the source instruction being evicted. The entries which result in a match are invalidated (*resultvalid* bit is

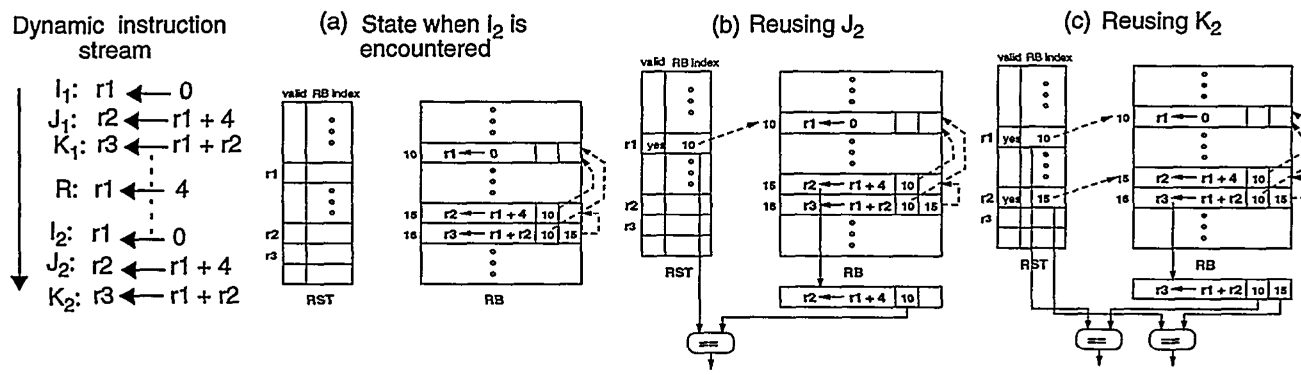


Figure 7. *Illustrating the reuse test for dependent instructions. (a) State when I_2 is encountered. (b) Testing r2 <- r1 +4 for reusability. (c) Testing r3 <- r1 + r2 for reusability. (Instruction R is not shown in RB for clarity).*

reset).

We illustrate the working of this scheme using the example shown in Figure 7 with the same dynamic stream of instructions as in Figure 5. Figure 7(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 RB and the index values in the *src-index* 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 RB entry 10 (the latest producer for *r1*)(Figure 7(b)). To establish the reusability of J_2, the *src-index* field for *r1* is compared with the RST entry for *r1* (Figure 7(b)). The 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 7 (c)). The instructions I_2, J_2, K_2 can be reused simultaneously if encountered in the same cycle. While performing the reuse test on each one, interdependence among them needs to be considered. The interdependence check is similar to what is done while renaming registers for multiple dependent instructions in the same cycle.

4 Microarchitecture with a Reuse Buffer

Figure 8 shows a generic microarchitecture with an RB. Except for the RB (and the datapaths associated with it), the rest of the microarchitecure is similar to what is found in a generic dynamically-scheduled superscalar processor.

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*. At this point, the RB is accessed to see if a reusable result for the instruction can be found. If a reusable result is found, the instruction does not need to be operated upon any further; it bypasses the *Instruction Window* (IW), and proceeds directly to the *Reorder Buffer* (ROB) [8]. Loads bypass the IW only if both micro-operations, address calculation and the actual memory operation, can be reused. 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 scheme S_{n+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 in the RB.

Since the RB contains state that will determine the outcome of future instructions, it needs to be maintained precisely (just like a register file). The straightforward way to do this is to update the RB only when an instruction is committed. However, this approach prevents speculatively-executed instructions from being entered into the RB, making it ineffective for one of our purposes, that of recovering squashed work. Accordingly, we must allow the RB to be updated speculatively, and take necessary actions (depending upon the reuse scheme) to ensure correct behavior. For scheme S_v, inserting instructions into the RB speculatively requires no special actions — the reuse test ensures that the correct result is obtained. For scheme S_{n+d}, the RST controls the reusability of instructions. Just like the rename map in a superscalar processor, checkpoints of the RST have to be taken when a speculation decision is made, and it has to be repaired in the case of an incorrect speculation.

Other issues, such as interlocks to ensure correct operations, flushing on context switches, etc., are fairly routine, and we don't discuss them further.

Though in all previous discussions we assumed that an RB

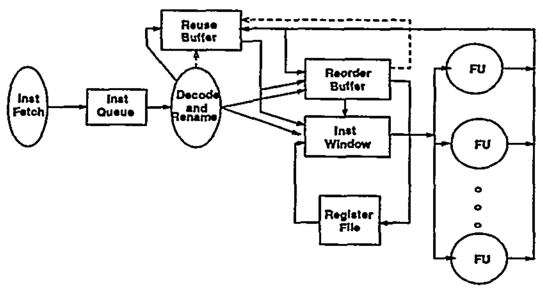


Figure 8. *Generic microarchitecture with a reuse buffer.*

3. 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.

access takes a single cycle, 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, the RB access can begin as early as in fetch stage); then only the reuse test needs to be performed in the decode stage. Other operations, like 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 free RB entries are always available, eliminating the search for a victim entry from the critical path. Thus, despite its size, the RB seems unlikely to be the structure that determines the cycle time.

5 Experimental Evaluation

Our simulator is built on top of the *SimpleScalar toolset* [1], an execution-driven simulator based upon the MIPS-I ISA. The base simulator models in detail a 4-way dynamically-scheduled processor with its first level of instruction and data cache memory. The parameters for the base out-of-order simulator are listed in Table 1. We extended this base simulator to incorporate the RB and the three instruction reuse schemes described earlier. The RB is integrated with the processor pipeline as described in section 4. In our simulations, the RB is capable of supporting 4 reads, 4 writes, and 4 independent invalidations simultaneously. We also assume that all RB accesses — read, write or invalidate — complete in one cycle, and that, like scheme S_{n+d}, scheme S_v can reuse multiple dependent results in a single cycle (the maximum length of a dependence chain reused in a cycle is equal to the read bandwidth of RB, which is 4 in the simulated configuration). This configuration of the RB, though aggressive, allows us to study the concept of instruction reuse without been limited by any particular implementation.

5.1 Benchmarks

The benchmark programs analyzed are listed in Table 2 along with their inputs and number of dynamic instructions executed on the timing simulator. There are five integer programs from SPEC '92 benchmark suite (*gcc, compress eqntott, espresso, xlisp*) and five integer programs from the SPEC '95 benchmark suite (*go, m88ksim, vortex, ijpeg* and *perl*). Other integer programs analyzed are: *Yacr2*, a VLSI channel router routing a channel with 230 terminals, and *Mpeg*, a mpeg decoder which decodes a mpeg file with 71 frames. Except for *go, m88ksim, vortex* and *ijpeg*, all programs were run to completion. These four programs were run for first 1 billion instructions on a functional simulator (so that we do not do all our measurement in the initialization phases) and for the next 500 million cycles (or completion) on the timing simulator. The exact number of instructions simulated in a fixed number of cycles is dependent on the microarchitectural enhancements applied. Thus, for these programs (except *ijpeg* which ran to completion) in Table 2 we show the approximate number of instructions executed on the timing simulator. All the benchmark programs were compiled using GNU *gcc* (version 2.6.3), *gas* (version 2.5.2) and *gld* (version 2.5) with maximum optimizations (-O3).

Benchmarks	Input	Inst. count (Mil.)
Gcc	1stmt.i	116.2
Compress	in	66.4
Eqntott	int_pri_3.eqn	1117.3
Espresso	bca.in	458.4
Xlisp	li-input.lsp	967.8
Go	null.in (reference)	~750.0
M88ksim	ctl.in (reference)	~1050.0
Vortex	vortex.in (training)	~900.0
Ijpeg	vigo.ppm(training)	442.2
Perl	scrabbl.in(training)	555.6
Yacr2	input2	397.5
Mpeg	foot.mpeg	385.9

Table 2: *Benchmark programs, inputs and instruction count.*

5.2 Experiments and Results

We performed several experiments to evaluate the concept of dynamic instruction reuse. Being the first paper on the concept (and mechanisms to exploit it), an exhaustive evaluation of all interesting cases is not possible. Furthermore, we also don't evaluate the concept in the abstract. Rather, we concentrate on

Instruction fetch	4 instructions per cycle. Aggressive: can fetch beyond multiple branches and across cache line boundaries. Fetch stops only on I-cache misses.
Instruction cache	16K bytes, direct mapped, 32 byte cache line, 6 cycles miss latency
Branch predictor	2048 BTB entries with 2-bit saturating counters.
Speculative execution mechanism	Out of order issue/commit of 4 operations per cycle, 32 entry reorder buffer, 32 entry load/store queue. Maximum of 8 unresolved branches. Loads execute only after all the preceding store addresses are known. Values bypassed to loads from matching stores ahead in load/store queue.
Architected Registers	32 integer, hi, lo, 32 floating point, fcc.
Functional units	4-integer ALUs, 2-load/store units, 4-FP adders, 1-integer MULT/DIV, 1-FP MULT/DIV
Functional unit latency (total/issue)	integer ALU-1/1, load/store 1/1, integer MULT 3/1, integer DIV 20/19, FP adder 2/1, FP MULT 4/1, FP DIV 12/12, FP SQRT 24/24.
Data cache	16K 2-way set associative, 32 bytes block, 6 cycles miss latency. Dual ported, non-blocking interface, one outstanding miss per register.

Table 1: *Details of the base simulator*

some key initial results for some sample configurations of the proposed mechanisms: how much dynamic reuse of instructions is there (as captured by our reuse schemes), what types of instructions are reused, how does it vary with RB size, and how much speedup results. We categorize total instruction reuse into squash reuse and general reuse, and show the contribution of either category to total speedup. We also evaluate the impact of associativity and the effectiveness of instruction reuse in alleviating dependences.

For most of our experiments we use fully-associative RBs of three different sizes: 32, 128, and 1024 entries with a FIFO 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 future work. We expect that with clever RB management policies, small RBs will be able to achieve the same performance as the larger RBs presented in the next several sections.

5.2.1 Instructions Reused

Figure 9 shows the percentage of total dynamic instructions reused for the three different schemes, with 3 different RB sizes for each scheme. The harmonic mean (HM) over all benchmarks for each RB size is also shown in the figure. All the analyzed benchmarks exhibit significant instruction reuse, especially for the larger buffer sizes. For scheme S_v with 1024 entries the percentage of instructions reused are 63% for eqntott, 39% for espresso, 76% for yacr2 and 34% for xlisp. Even for small RB sizes the instructions reused are significant (21% for eqntott, 24% for espresso, 26% for yacr2). For other benchmarks and for other reuse schemes the percentage of instruction reuse is also appreciable.

For larger RB sizes, scheme S_n does not reuse as many instructions as schemes S_v and S_{n+d}. This is because invalidations are more frequent in scheme S_n (being done every time a register or memory location is written), which limit the number of reusable instructions irrespective of the RB size (for large RB sizes). Frequent invalidations help small size RBs (32 entries); the instructions which are more likely to be reused remain in the RB, resulting in better utilization. Thus, S_n performs better than other two schemes for RB with 32 entries.

In scheme S_v, invalidations are infrequent: only stores that match loads cause invalidations. Accordingly, larger buffers are able to retain more reusable instructions, resulting in S_v performing better than S_n and S_{n+d} for RB with 1024 entries. The smaller number of invalidations also means that instructions which are not likely to be reused remain in RB. This phenomenon results in scheme S_v performing worse than S_n and S_{n+d} for an RB size of 32 entries.

Since scheme S_{n+d} uses selective invalidations (only independent instructions are invalidated), the frequency of invalidations is reduced while still retaining the ability to purge unusable instructions from the RB. Thus, S_{n+d} not only continues to benefit as the RB size is increased to 1024 entries but also out performs other two schemes for RB with 128 entries.

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 part is reused. (As noted earlier, for

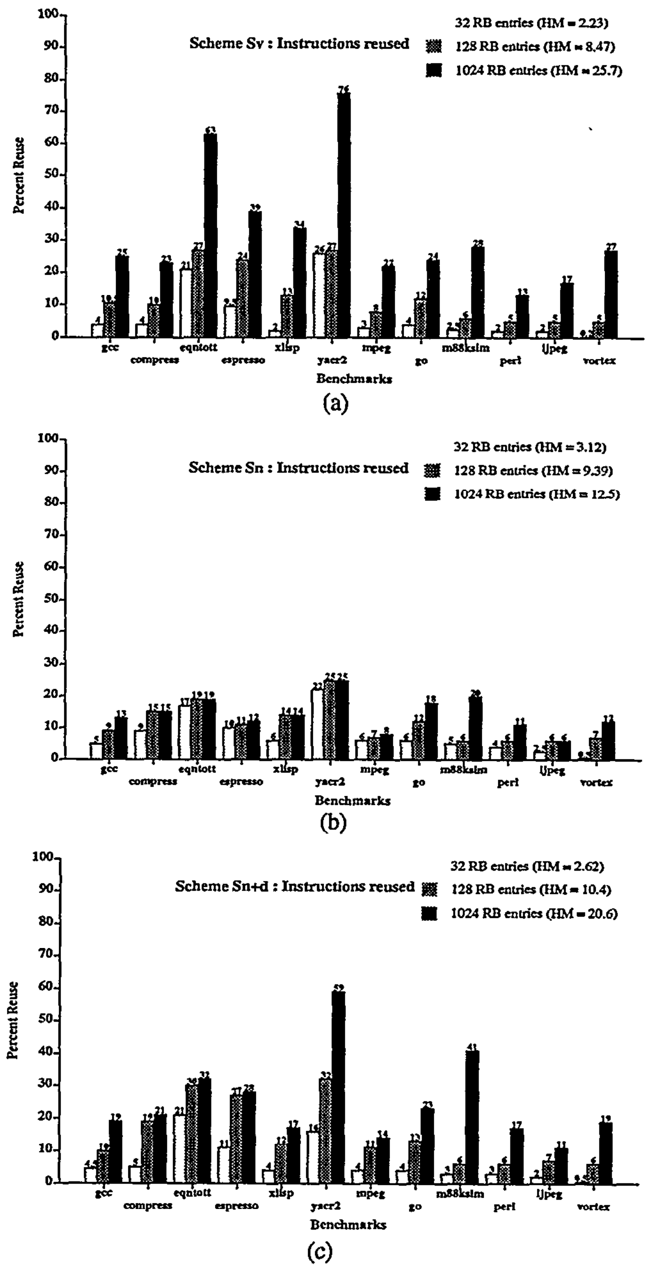


Figure 9. *Percentage of instructions reused for RB sizes: 32, 128 and 1024 entries. The RB in these experiments was fully associative. (a) Scheme S_v (b) Scheme S_n (c) Scheme S_{n+d}. HM stands for harmonic mean.*

stores we reuse only 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*. Table 3 shows the percentage of instructions reused (average over all benchmarks) from each category using a 1024 entry RB (e.g. 36.9% of all integer instructions with two register operands are reused). 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.

Figure 10 shows the contribution of each instruction category to the total instruction reuse (averaged over all benchmarks), for 3 different RB sizes. 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, it is worth noting that almost 40-50% of the reuse comes from the load instructions (about 15%) and address calculations (25-35%).

5.2.2 Speedups

Figure 11 shows the speedups ($IPC_{withRB}/IPC_{withoutRB}$) obtained with the different reuse schemes for varying RB sizes. The harmonic mean (HM) over all the benchmarks for each RB size is also shown in the figure. The speedups are not as impressive as the percentage of instructions reused, however, they are still significant in many cases; they range from no speedup to 19% for a 32 entry RB, from 2% to 28% for a 128 entry RB, and from 3% to 43% for a 1024 entry RB. The speedups are not as impressive because there are many other parameters that contribute to overall performance (e.g., cache misses) and reducing the instruction execution component of the execution time may not result in a proportionate decrease in the overall time. With more ideal system parameters (for which instruction issuing and/or the lengths of dependence paths are more important), the speedups will mirror the percentage of instructions reused more closely.

Comparing the harmonic means of the speedups we can see that S_n performs best (among the three schemes) for a 32 entry RB (harmonic mean 4.3), S_{n+d} performs best for a 128 entry RB (harmonic mean 7.2), S_v works best for a 1024 entry RB (harmonic mean 14.9). Comparing percent instruction reuse (section 5.2.1) with the resulting speedups, we see that the scheme that reuses more instructions also delivers better overall speedup.

Instruction Categories		Instruction Reused (%)		
		S_v	S_n	S_{n+d}
Loads (value)		21.2	6.7	11.9
Address Calculations		20.8	10.3	11.9
Control		35.1	1.8	20.2
integer	two reg operands	36.9	26.8	32.0
	one reg operand	38.6	17.9	30.9
	immediate	51.0	98.1	90.4

Table 3: *Percent reuse per instruction category for a 1024 entry RB*

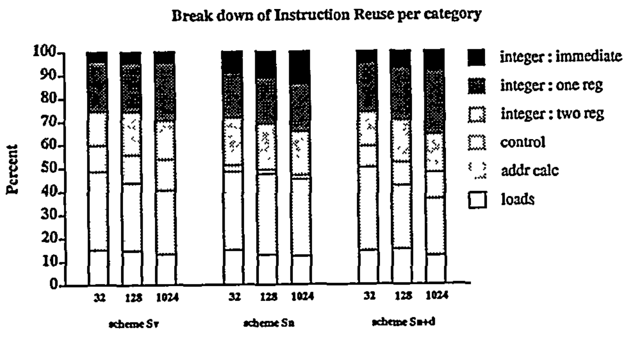


Figure 10. *Contribution of each instruction category to total reuse. These numbers are average over all benchmarks for full associative RB.*

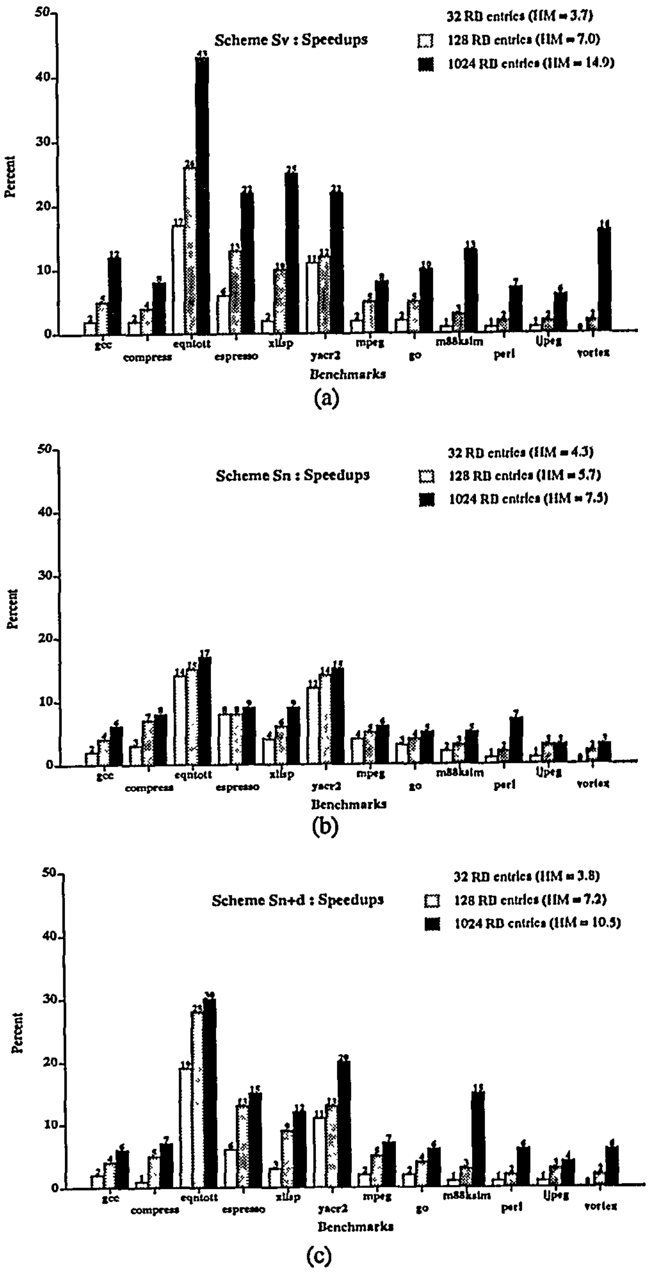


Figure 11. *Speedups obtained due to instruction reuse. The numbers are presented for RB entries 32, 128 and 1024. (a) Scheme S_v (b) Scheme S_n (c) Scheme S_{n+d}. HM stands for harmonic mean.*

5.2.3 Squash Reuse vs. General Reuse

Figure 12 gives a break down of the number of instructions reused into two categories: *squash reuse* and *general reuse.* Squash reuse and general reuse have been illustrated in Figure 1 and Figure 2 respectively. Due to a lack of space, we do not present the breakdown for all three schemes; Figure 12 contains the information only for scheme S_{n+d}. The figure suggests that, as one might expect (with a couple of exceptions), smaller RB sizes have a larger percentage of squash reuse. This is also true for the other schemes.

Figure 13 separates the performance obtained by squash reuse from that obtained by general reuse for each benchmark for scheme S_{n+d} with the three different RB sizes. Observe that 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 12). This suggests that squash reuse is more time critical than general reuse — the squash penalty impacts the bottom line more than the latency of an instruction (or a set of instructions), especially in a dynamically scheduled processor.

5.2.4 Set Associative RB

Figure 14 presents the results for a 4-way set associative, 128 entry RB, for scheme S_v. (Figure 14(a) presents the instruction reuse and Figure 14(b) presents the speedups.) Other schemes and buffer sizes show similar results. As we can see, the performance is comparable in either case. For some programs (e.g., *eqntott, espresso,* and *compress*) set associativity actually improves performance. This is due to the fact that the FIFO replacement policy that we use does not discriminate between reusable and not reusable instructions. Reusable entries are evicted even though non-reusable entries are present (the non-reusable entries are evicted in the set-associative case because of limited choice).

5.2.5 Early resolution of data dependence

We now evaluate the effectiveness of instruction reuse in reducing the length of dependence chains. To do so, we measure the average number of cycles an instruction spends waiting for its operands to be ready (this is called the *data dependence resolution latency* [5]). Figure 15 plots the data dependence resolution latency with instruction reuse, normalized to that without instruction reuse. The data in the figure is for 4-way set associative RB implementing reuse scheme S_v. As is evident from the figure, instruction reuse causes significant reduction in operand waiting times. For *eqntott* and *espresso,* the waiting time is cut down by 40% and 32%, respectively, for an RB size of 128 entries, suggesting that dynamic instruction reuse is quite effective in collapsing true dependences.

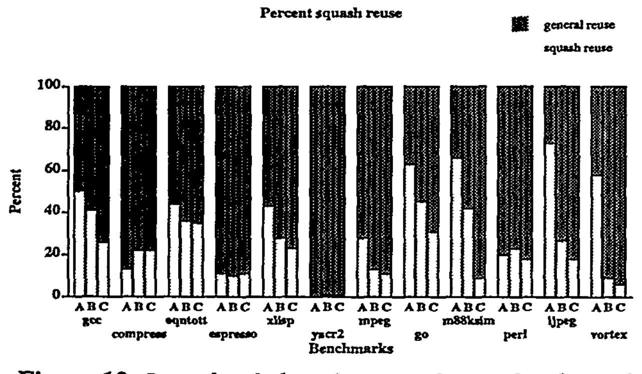


Figure 12. *Reuse break down in terms of general and squash reuse using scheme S_{n+d}. Bar 'A' stands for a 32 entry RB, 'B' stands for a 128 entry RB, and 'C' stands for a 1024 entry RB.*

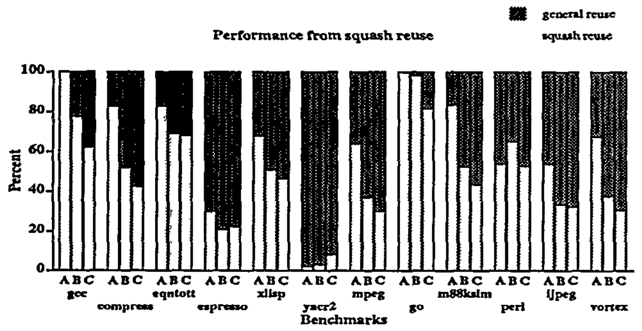
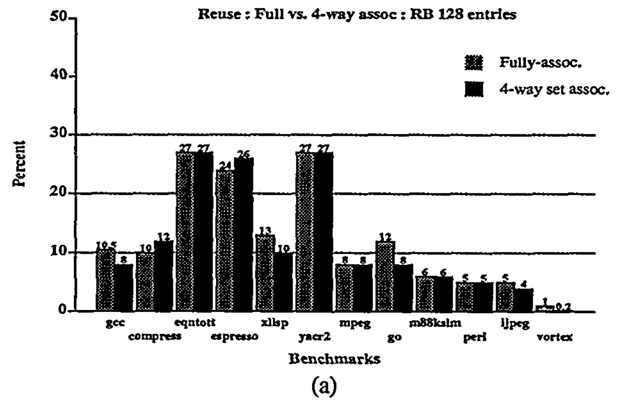


Figure 13. *Performance break down in terms of general and squash reuse using scheme S_{n+d}. Bar 'A' for a 32 entry RB, 'B' for a 128 entry RB, and 'C' for a 1024 entry RB*



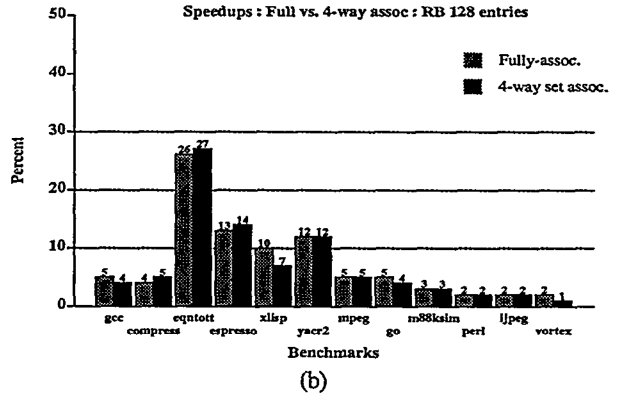


Figure 14. *Comparison of 4-way set associative RB against fully-associative RB. (a) Percentage instruction reused, (b) Speedups. The results are for RB with 128 entries using Scheme S_v.*

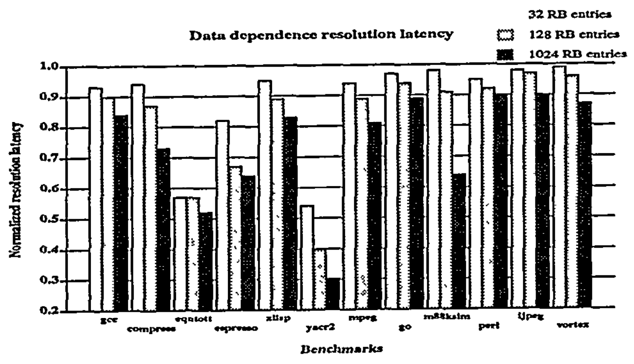


Figure 15. *Data dependence resolution latency with instruction reuse normalized to that without instruction reuse. The result is shown for scheme S_v using 4-way set associative RBs of three different sizes.*

6 Related Work and Discussion

The idea of not having to redo computation is not a new one — it has been used before in several different contexts. A technique called *memoization* has been used for functional and logic programs. 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.

Harbison in [2][3] proposes a stack-oriented architecture, the *Tree Machine*, which uses a hardware mechanism, the *value cache*, for eliminating common subexpressions 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 [7] 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 *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.

In [6], Oberman and Flynn, 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 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 [2][3] approach is tailored for an architecture which expresses computation in the form of *parse trees* (Tree Machine). The result caches [7], and the division and reciprocal caches[6] 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 [3], result cache in [7] and division and reciprocal caches in [6]) by using either the operand address [3] or operand values [7][6], which are only available later in the pipeline. Thus, the result buffer access is delayed till the execute stage, which restricts the usefulness of these techniques only to instructions which have multi-cycle latency ([7] uses it for floating point instruction, while [6] uses it for floating point divides only). In contrast, the reuse schemes presented in this paper 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 prediction have been proposed to achieve the same effect [4][5]. In [5], Lipasti et. al. propose predicting the value of loads to collapse the true dependencies. In [4] they extend this concept to predict the value of registers. The fundamental difference from our schemes is that these approaches are based on prediction. The instructions still must execute to generate result for later verification. Our schemes are not based on prediction, and the reused result is guaranteed to be correct.

7 Conclusions

In this paper we introduced and studied the concept of dynamic instruction reuse. Empirical observations suggest that in a program execution, many instructions (and groups of instructions) are executed repeatedly with the same inputs, generating the same results. We discussed two causes of this behavior: (i) the re-execution of (control-independent) computation when recovering from a branch mis-prediction, and (ii) the generic nature of programs which are written to operate on a variety of data inputs.

We presented three schemes for exploiting the phenomenon. All three 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, and scheme S_{n+d} uses operand names as well as 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: 32, 128, and 1024 entries, using execution-driven simulation. Significant instruction reuse was found in many cases, with as many as 76% of the instructions being reused in one case. Furthermore, reuse was not limited to a particular category of instructions; a significant number of instructions were reused from all the broad categories of instructions considered. We also measured the resulting speedup in the program execution time. The speedup is not as pronounced as the percentage of instructions reused, but it is still quite significant, with as much as 43% speedup in one case. We also observed that a 4-way set associative reuse buffer compared favorably in performance for the cases considered. Finally, we measured the effectiveness of reuse in cutting down data dependence path lengths by measuring the average time that an instruction waits for operands. We found that the waiting time was cut down by 40% in one case.

This paper represents only a first attempt at studying a phenomenon (and associated means to exploit it) that could have significant implications for the microarchitecture of the future processors. There is a great deal of work that remains to be done; this work can broadly be classified into two-related categories: (i) better exploiting the reuse phenomenon, and (ii) impact of reuse on other microarchitectural components. Several issues need to be investigated in either category. In the former, we have observed that only about 20% of the instructions inserted into the RB constitute all the reuse. This calls for selective insertion policies that result in better reuse characteristics with small reuse structures. Another issue that needs to be investigated is better invalidation mechanisms (for schemes S_n and S_{n+d}) such as invalidation using time-stamps. Software transformations that facilitates reuse are also an area that deserves study. Considerable work is also needed in the second category. Success at instruction reuse might cause us to rethink the need for aggressive speculation structures: there might be no need to predict the outcome of a branch if its outcome can be determined from a reuse buffer. Likewise, there might be no need to carry out value prediction, if the value can be determined from the reuse buffer. In this case, resources that would otherwise be spent in making more powerful speculation structures might be better spent in structures that improve instruction reuse, backed up by less powerful speculation structures.

Acknowledgments

We thank Scott Breach, Doug Burger, Andy Glew, Andreas Moshovos, Shubu Mukherjee, Subbarao Palacharla and the anonymous referees for their comments on earlier drafts of the paper. We would also like to thank Haitham Akkary and Todd Austin for their comments on this work.

This work was supported in part by NSF Grants CCR-9303030 and MIP-9505853, the U.S. Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order no. D346, a donation from Intel Corp, and an equipment donation from Sun Microsystems. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S Army Intelligence Center and Fort Huachuca, or the U.S. Government.

References

[1] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996.
(URL: http://www.cs.wisc.edu/~mscalar/simplescalar.html)

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

[3] S. P. Harbison. An architectural alternative to optimizing compilers. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 57 65, Mar. 1982.

[4] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. of 29th International Symposium on Microarchitecture*, pages 226–237, Dec. 1996.

[5] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proc. of ASPLOS VII*, pages 138–147, Sept. 1996.

[6] S. F. Oberman and M. J. Flynn. On Division and Reciprocal Caches. Technical Report CSL-TR-95-666, Stanford University, Apr. 1995.

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

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