Understanding the Differences Between Value Prediction and 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

Recently two hardware techniques — Value Prediction (VP) and Instruction Reuse (IR) — have been proposed for exploiting the redundancy in programs to collapse data dependences. In this paper, we attempt to understand the different ways in which VP and IR interact with other microarchitectural features and the impact of such interactions on net performance. More specifically, we perform the following tasks: (i) we identify the various differences between the two techniques and qualitatively discuss their microarchitectural interactions, (ii) we evaluate the impact on performance of these interactions, and (iii) since IR is more restrictive of the two techniques, we also estimate the amount of total redundancy, present in programs, that can be captured by IR.

Our results show that the performance obtained by VP is sensitive to the way branches with value-speculative operands are handled. We also see that, although IR captures less amount of redundancy, it may perform equally well because it validates results early, it is non-speculative, and it reduces branch misprediction penalty. Finally, we show that 84-97% of redundancy in programs can be reused, implying that the approach of detecting redundant instructions non-speculatively, based on their operands, does not significantly restrict IR's ability to capture redundancy present in programs.

1. Introduction

Several recent studies [2, 5, 8, 10] have shown that there is significant result redundancy in programs, *i.e.*, many instructions perform the same computation and, hence, produce the same result over and over again. These studies have found that for several benchmarks more than 75% of the dynamic instructions produce the same result as before. Also, recently, two hardware techniques have been proposed to exploit this redundancy: (i) Value Prediction (VP) [3, 4, 5], and (ii) Instruction Reuse (IR) [9]. Both techniques attempt to reduce the execution time of programs by alleviating the dataflow constraint. They use the redundancy in programs to determine — speculatively (Value Prediction) or non-speculatively (Instruction Reuse) — the results of instructions without actually executing them. The advantage of doing so is that instructions do not have to wait for their source instructions to execute first; they can execute sooner using the results obtained by the above two techniques, thus, relaxing the dataflow constraint.

Although both VP and IR attempt to shorten the critical path through a computation, they follow very different approaches. VP predicts the results of instructions (or, alternatively, the inputs of other instructions) based on the previously seen results, performs computation using the predicted values, and confirms the speculation at a later point. The critical path is shortened since the instructions that would normally be executed sequentially could be executed (speculatively) in parallel. On the other hand, IR recognizes that a certain computation chain has been performed before and therefore need not be performed again, *i.e.*, it "splices out" a chain of computation from the critical path.

The effectiveness of any microarchitectural technique in improving the net performance of a processor not only depends on how well it performs by itself, but also on how it interacts with other microarchitectural features (e.g., branch prediction, availability of resources) when it is integrated in a pipeline. Since VP and IR are different techniques, they not only perform differently by themselves (*i.e.*, capture different amounts of the redundancy present in programs) but also interact with other microarchitectural features in different ways, thereby, impacting the net performance differently. The purpose of this work is to identify and evaluate the different microarchitectural interactions of these techniques. The intent is not to argue which technique is better, but is to gain a better understanding of the working of each technique. We feel, that will help in designing other techniques (possibly hybrid of VP and IR) that exploit the redundancy in programs more profitably. More specifically, in this paper we achieve the following three tasks. (i) We identify the various differences between the two techniques and qualitatively discuss their microarchitectural interactions. (ii) We evaluate the impact on performance of these interactions. And finally, (iii) since IR is more restrictive of the two techniques (we discuss this later), we also estimate how much of the total redundancy present in programs can be captured by IR.

The layout for the rest of the paper is as follows. In Section 2, we describe VP and IR in more detail. In Section 3, we identify the various differences between them, and qualitatively discuss various interactions and their the impacts on performance. In Section 4, we evaluate these interactions quantitatively. Finally, in Section 5, we summarize and provide conclusions.

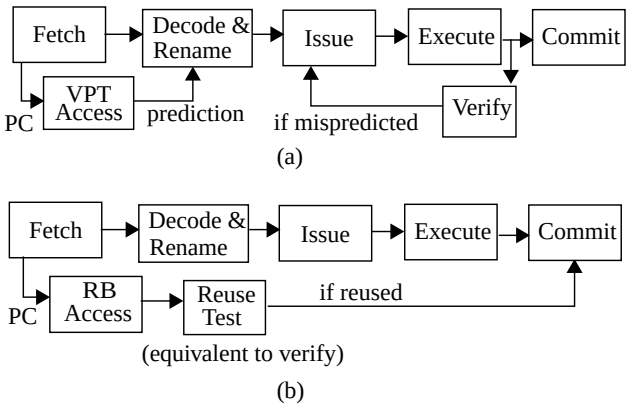

Figure 1: (a) Pipeline with VP, (b) Pipeline with IR.

2. Value Prediction and Instruction Reuse

As mentioned earlier, VP is a *speculative* technique that exploits redundancy in programs to predict values that are either produced (results) or used (inputs) by instructions. In Figure 1(a), we show a pipeline with VP. The predictions are obtained from a hardware table, called *Value Prediction Table (VPT)*. These predicted values are used as inputs by instructions, which can then execute earlier than they could have if they had to wait for their inputs to become available in the traditional way. When the correct values become available (after executing an instruction) the speculated values are verified; if a speculation is found to be wrong, the instructions which executed with the wrong inputs are re-executed (Figure 1(a)). The execution of such instructions is delayed by the latency of verifying the prediction (*VP-verification latency*). However, if the speculation is found to be correct then nothing special needs to be done; instructions get executed earlier than they would have otherwise. VP collapses true dependences by allowing dependent instructions, that would have executed sequentially, to execute in parallel.

Unlike VP, IR is a *non-speculative* technique that exploits redundancy in programs by obtaining results of instructions from their previous executions, and thereby, not executing them. In Figure 1(b) we show a pipeline with IR. When an instruction is first executed, its results are stored in a hardware structure called a *Reuse Buffer* (RB), indexed by its PC. When the instruction is encountered again, its previous results are read from the RB (in parallel with fetching the instruction) and their validity established by a *reuse test* (in parallel with decoding the instruction). The reuse test validates results by establishing that the current operands values are the same as those used to calculate the results. There are different ways of doing so, one of which is described later in Section 4.1.2 of this paper. Since the correct results are known, a reused instruction is not executed, and instead it is queued for retirement. IR collapses true dependences by reusing in the same cycle a dependent chain of instructions that would normally execute sequentially.

In Figure 2, we illustrate how VP and IR improve performance by collapsing data dependences. In the figure, we show a flow of a dependent chain of instructions (I, J, and K) through three different pipelines: (i) a base pipeline (without VP or IR); (ii) a pipeline with VP; and (iii) a pipeline with IR. In all three cases, we assume the instructions I, J, and K, are fetched, decoded and renamed together. In the base pipeline, the instructions execute sequentially, since they are data dependent, requiring three cycles to execute them; the chain is committed by cycle 6. In the pipeline with VP, the dependence between instructions is broken by predicting the outputs of I and J (alternately, the inputs of J and K). This enables the three instructions to execute simultaneously (in cycle 3); the chain is committed in cycle 4. In the pipeline with IR, the previous results of these instructions are reused in parallel with decoding them (in cycle 2). The data dependence is alleviated because the dependent instructions are reused at the same time. Since the results are known, the instructions skip over the execute stage and are committed in cycle 3. Thus, we see that both VP and IR enable dependent instructions to complete simultaneously, which otherwise would have completed one at a time.

3. Impact of Differences: Qualitative

The pipelines in Figure 1 bring forth the key difference between the two techniques: IR verifies the results before

Pipeline	Base Superscalar						With VP				With IR		
	1	2	3	4	5	6	1	2	3	4	1	2	3
Fetch	I, J, K						I, J, K				I, J, K		
Dec & Ren		I, J, K						I, J, K				I, J, K	
Execute			I	J	K				I, J, K				
Commit				I	J	K				I, J, K			I, J, K

Figure 2: Flow of a dependent chain of instruction, I, J, K, (where J is dependent on I, and K is dependent on J) on (i) a base superscalar, (ii) a superscalar with VP, and (iii) a superscalar with IR. For both VP and IR, the true dependences are collapsed because the instructions in the chain do not execute one at a time as in the base superscalar.

using them (*early validation*), while VP uses the results speculatively and verifies them later (*late validation*). This feature of early and late validation leads to two differences in the way VP and IR function: (i) VP is speculative while IR is non-speculative, and (ii) reused instructions do not execute, while the value predicted instructions need to execute to verify the prediction. Due to these differences, the two techniques vary in (i) the amount of redundancy they can capture, and (ii) the way they interact with other microarchitectural features. We elaborate on them next.

3.1 Amount of Redundancy Captured

Since IR validates results early based on inputs, it may be conservative. For example, if the inputs of an instruction are not ready at the time it is tested for reuse then it will not get reused; or, an instruction that produces the same result but with different inputs (e.g. logical operations, loads) will not get reused. However, VP can make the correct predictions for each of the above cases since it neither depends on inputs being available nor on them being the same. Thus, IR may not capture as much redundancy in programs as VP. However, validating results early makes IR non-speculative and has other advantages that we discuss next.

3.2 Impact on Performance

In this section, we describe the interactions of VP and IR with other microarchitectural features and qualitatively reason about their likely impact on performance.

- **Effect of value misprediction:** When a value is mispredicted, instructions dependent on that value are re-executed. Since mispredictions are detected during the verification stage, the execution of these instructions is delayed by the VP-verification latency. On the other hand, IR does not incur any misprediction penalty.

- **Impact on branch prediction:** Typically, when a branch is mispredicted, all instructions following the branch are discarded and instruction fetch resumes from the correct address. The branch misprediction penalty, which includes cycles spent executing the discarded instructions, can be reduced if the misprediction is detected earlier in the pipeline; this way the machine can start executing on the correct path sooner, saving cycles that would have otherwise been wasted doing the wrong work. Both VP and IR can resolve branches (and thereby detect mispredictions) earlier by collapsing the dependent chains of operations leading to the branches, and thereby, reduce the branch misprediction penalty.

  VP and IR also interact with branch prediction in other ways. IR further reduces branch misprediction penalty due to two reasons. First, when a mispredicted branch is reused, the misprediction gets detected earlier (at decode) than it would have if the branch had to execute. Second, due to convergent code sometimes processors perform useful work on the wrong path. Since IR buffers work done on the wrong path, it can recover useful work from the work squashed due to misprediction, resulting in faster recovery.

  VP, on the other hand, may increase the branch misprediction penalty in two ways: (i) by causing more mispredictions, and (ii) by delaying branch resolution. Which of the two effects occur depends on how the branches that execute with value-speculative operands are handled. For the purpose of explanation we semantically divide a branch into two operations: (i) *executing a branch* — which means determining its outcome, and (ii) *resolving the branch* — which means taking the action based on its outcome (e.g. squash). Branches with speculative operands can be handled in two ways: (i) they are resolved while their operands are still speculative, or (ii) their resolution is delayed until their operands become non-speculative. The first option may cause spurious branch mispredictions, because now correctly predicted branches may also get mispredicted when they produce the wrong outcome (due to wrong inputs). In the second option, where the action is taken only after the branch inputs are known to be non-speculative, branches have to wait till their source instructions have been verified. This delays the branch resolution by the latency of VP-verification, delaying branch misprediction detection and, thereby, increasing the misprediction penalty.

- **Impact on resource contention:** As an instruction flows through a pipeline it contends for different resources (e.g. functional units, cache ports) at different stages in the pipeline. VP and IR may influence resource contention by changing both the pattern in which the resources are requested and the demand for them. By collapsing true dependences both VP and IR can make instructions ready to execute sooner. This changes the schedule of the instruction execution, which may result in clustering or spreading of the requests for resources, thereby, increasing or decreasing resource contention.

  Since a reused instruction does not execute, IR tends to reduce the demand for resources. This frees up resources for use by other contending instructions. VP, on the other hand, may increase the demand for resources, since instructions which execute with wrong inputs need to re-execute. These instructions may execute multiple times if they see wrong values repeatedly (placing further demand on the resources).

- **Impact on execution latency:** IR decreases execution latency of individual operations, since reusing an operation effectively reduces its execution time (from possibly multiple cycles) to the 1 cycle (latency of performing reuse). Unlike IR, VP does not bypass execution — the instructions still have to execute to verify their prediction — hence, it does not impact the execution latency of the operations. Thus, in VP, the completion time of an instruction will still be limited by its execution (and verification) latency.

Instruction fetch	4 insts per cycle. Only one taken branch per cycle. Cannot fetch across cache line boundaries in the same cycle.
Instruction cache	64K bytes, 2-way set assoc., 32 byte line, 6 cycles miss latency.
Branch predictor	Gshare [6]. 10-bit history register, 16K entry counter table.
Speculative execution mechanism	O-o-O issue of 4 operations/cycle, 32 entry reorder buffer, 32 entry load/store queue. Max of 8 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)	8-integer ALUs, 2 load/store units, 4-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.
Data Cache	64K bytes, 2-way set assoc., 32 byte line, 6 cycles miss latency. Dual ported, non-blocking.

Table 1: Details of the base simulator

4. Impact of Differences: Quantitative

In this section, we first quantitatively evaluate the various microarchitectural interactions of VP and IR, and determine their impact on net performance. In the latter part of this section, we estimate amount of total redundancy present in programs that can be captured by IR.

4.1 Impact on Performance: Experimental Setup

Since we are more concerned with the differences between the paradigms of value prediction and instruction reuse, we would, ideally, like to remove implementation specific effects from the evaluation. But, using oracle schemes or schemes with unbounded buffers (resulting in very high prediction and reuse rate) would mask the "real life" effects of these techniques that we wish to highlight. Thus, we choose two comparable and realistic schemes to implement each of the technique. We describe these schemes below.

4.1.1 Value Predictor

We implement VP using a scheme, which we call VP_{Magic}. This scheme is like the *last value predictor* [4], except that instead of saving only the last result of an instruction we save its last 'n' unique results. With each result we also store a 2-bit confidence counter, which is incremented or decremented depending on whether prediction is right or wrong. Only *confident* results (which have the counter value above certain threshold) are used for prediction. We obtain the prediction for the result of an instruction as follows: if the correct result of the instruction is present among the last 'n' results, then that result is selected as prediction; otherwise, the most confident result is selected as the prediction. The reason we use such an oracle selection policy (which gives the scheme its name) is to make the VP scheme comparable to the IR scheme (we describe it next). The IR scheme is capable of buffering different instances of an instruction and selecting the correct instance for reuse.[1] Since we did not want this difference to bias our evaluation, we choose to make VP scheme equally powerful. However, this VP scheme is still quite realistic; [11] describes a value prediction scheme, which buffers 'n' results per instruction and accurately selects the prediction value from these 'n' values.

We also simulate the *last value predictor*, V_{LVP}, which uses the last result of an instruction as a prediction for its future result. This predictor incurs higher value mispredictions than VP_{Magic}. We simulate this predictor because it permits us to observe how the various interactions we wish to study will shape up for programs where the value prediction performance is not high.

4.1.2 Reuse Scheme

The IR scheme we simulate is S_{n+d},[2] described in [9]. In this scheme, results of instructions are stored in the RB along with two pieces of information needed for establishing the reusability of the result: (i) the operand register names and (ii) pointers to the RB entries of the instructions which produced values for the operands. The pointers link the RB entries to form a dependent chain. A reusable entry is detected as follows. The start-entries of dependent chains are invalidated when their operand registers are overwritten; only valid start-entries are reused. Other entries in a dependent chain are reused if the entries on which they are dependent have been reused. We handle loads in a special way. Load entries are invalidated when a store writes to their memory address. In this paper, we augment this scheme in two ways. First, we also save operand values with the RB entries. A start-entry is invalidated only if the new operand value is different from the old one. Second, if the operand values for an invalidated entry become current again then the entry is reverted to the valid state.

4.1.3 Microarchitecture

Our microarchitectural simulator is built on top of the *SimpleScalar toolset* [1], an execution-driven simulator

[1] One way to select the correct instance from among several instances is to read all instances out of the RB, and then perform the reuse test on each of them in parallel. The instance that succeeds the reuse test is selected.

[2] Letters 'n' and 'd' in S_{n+d} stand for *name* and *dependences* between instructions. These are the two pieces of information used for establishing the reusability of the results.

based upon MIPS-I ISA. The simulator models in detail a 4-way dynamically-scheduled processor with its first level of instruction and data cache memory. The parameters for the out-of-order simulator are listed in Table 1.

A VPT and an RB are incorporated in this pipeline as shown in Figure 1. We use a 16k-entry VPT and a 4k-entry RB. Both structures are 4-way set associative (*i.e.*, they can store a maximum of 4 instances per instruction), with LRU replacement policy. Since an RB entry is effectively 4 times the size of a VPT entry (an RB entry stores the operand values and dependency information with the result), we provide the VPT with 4-times as many entries as the RB so as to assign same amount of hardware storage to both techniques. Both structures support four reads and four writes per cycle, which allows four instructions to be value predicted or reused per cycle. In addition, the RB supports invalidations based on four different register names. Both VP and IR can collapse dependence chains up to four instructions long in a cycle.

We employ an aggressive value misprediction recovery policy. In case of mispredictions, only the dependent instructions are re-executed. Only the first instruction in the dependent chain pays the misprediction penalty; other instructions issue as they see new values. This ensures that the misprediction penalty is charged only once for the entire dependent chain (as opposed to charging it for every instruction in the chain).

4.1.4 Configurations Studied

As we described in Section 3.2, VP can interact with branch prediction differently depending on how branches with value-speculative operands are handled. To evaluate the impact of these interactions, in our simulations we handle branches in two ways: (i) *speculative branch resolution* (SB) — where branches are resolved as soon as they execute (even if their operands are value-speculative); and (ii) *non-speculative branch resolution* (NSB) — where branches are resolved only after their operands become non-value-speculative.

Also, as described Section 3.2, VP may cause instructions to re-execute several times. To evaluate its impact, we handle re-executions in two ways in our simulations: (i) *multiple executions allowed* (ME) — where we allow an instruction to execute as many times as it sees new input values; and (ii) *no multiple executions allowed* (NME) — where we re-execute instructions once after their correct operands are known.

A combination of the above variations results in four different configurations been simulated for VP: ME-SB, NME-SB, ME-NSB, and NME-NSB.

To see the effect of VP-verification latency, we run the VP experiments with both 0- and 1-cycle verification latency. For IR experiments, we assume that the reuse test can be performed in parallel with instruction decode, hence it does not incur any extra latency.

4.1.5 Benchmarks

We used seven programs from the SPEC95 integer benchmarks suite for our study. The benchmark programs are listed in Table 2 along with their inputs, the number of dynamic instructions executed on the timing simulator, and branch and return prediction rates. We simulate all benchmarks for 200 million cycles or until completion, which ever occurs earlier. For *go*, *m88ksim*, *ijpeg*, *vortex*, and *gcc* we skip the initial 1 billion instructions, and for *compress*, we skip the first 2.5 billion instructions, executing them on a functional simulator (so that we don't make all our measurements in the initialization phase). The exact number of instructions simulated in a fixed number of cycles is dependent on the microarchitectural enhancement applied. Thus, the number of instructions shown in table are approximate numbers. All 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 -funroll-loops -finline-functions).

4.1.6 Value Prediction and Reuse Rates

In Table 3, we show the percentage of reuse and value prediction rates obtained for various benchmarks using the schemes described earlier. As expected, more results and addresses get value predicted correctly (VP_{Magic}) than reused, except for *compress*, where more addresses get reused. This is because for many loads in *compress* IR reuses only addresses (not results), but VP is able to predict their results and hence does not need to predict addresses. We also show the prediction accuracy obtained for VP_{LVP}. Again, as expected, VP_{LVP} makes less correct predictions than VP_{Magic} (except *ijpeg*, where misprediction rates are also higher), since it buffers only one instance per instruction.

Bench	Input	Inst. Count (mil.)	Br. Pred Rate (%)	Ret. Pred Rate (%)
go	null.in (ref)	354.7	75.8	99.9
m88ksim	ctl.in (ref)	491.4	94.6	100
ijpeg	vigo.ppm(train)	439.8	88.8	99.9
perl	scrabble.in (train)	479.1	95.6	100
vortex	vortex.in (train)	507.6	97.8	99.9
gcc	reload.i (ref)	420.8	92.0	100
compress	bigtest.in (ref)	421.2	89.3	100

Table 2: Benchmark programs, their inputs, inst. committed (after skipping), branch and return prediction rates.

Benchmarks	IR		VP_{MAGIC}				VP_{LVP}			
	result (%)	address (%)	result		address		result		address	
			pred (%)	mispred (%)	pred (%)	mispred (%)	pred (%)	mispred (%)	pred (%)	mispred (%)
go	24.3	19.9	38.4	3.3	26.8	4.7	30.4	4.5	25.6	4.0
m88ksim	48.5	33.9	54.8	0.6	42.0	4.6	42.0	2.7	31.2	1.3
ijpeg	11.2	24.0	16.7	0.9	19.4	2.2	17.4	4.4	18.1	2.2
perl	19.8	28.1	35.4	1.2	35.6	2.0	26.8	1.7	32.0	1.2
vortex	20.9	16.2	36.7	1.1	26.9	4.4	33.8	3.3	24.7	3.3
gcc	18.6	19.4	36.5	1.9	23.9	5.2	29.2	3.9	18.9	2.9
compress	16.5	65.1	20.5	0.2	43.4	0.03	17.3	0.6	41.7	0.1

Table 3: Percentage IR and VP rates for various benchmarks. The result percentages are over the total number of dynamic instructions simulated, while the address percentages are over the total number of memory operations.

4.2 Results

4.2.1 Early Validation

As pointed out in Section 2, a key feature of IR is that it validates results before using them. This early validation has several benefits: it makes results available sooner in the pipeline, resolves branches early, and reduces the demand for execution resources (since reused instructions do not execute). In this section, we isolate the importance of early validation to performance. In Figure 3, we show the percentage speedups obtained with IR over the base case for the two experiments: *early* — where results are validated at decode stage (as in IR); and *late* — where results are validated at execute stage (this is as if the reused instructions where predicted correctly). We see that more than half of the performance improvement is lost if the validation is deferred to the execution stage.

4.2.2 Interaction with Branch Prediction

In this section we quantify the interactions of VP and IR with branch prediction.

- **Spurious branch mispredictions:** In Table 4, we show the increase in the number of branch squashes due to spurious branch mispredictions. The numbers shown are for configurations SB; for configurations NSB (where the branches are not resolved value-speculatively) the number of branch squashes is not affected. From the table we see that spurious mispredictions can increase the number of branch squashes significantly for some benchmarks (e.g., *go*, *perl*, *vortex*). Although, the increase in mispredictions is large, the impact of these extra mispredictions on overall performance will be determined by how much misprediction penalty they incur. Since, these spuriously mispredicted branches use speculative values, they get executed (and squashed) early in the pipeline, thereby incurring less misprediction penalty than branches which resolve late in pipeline.

In Table 4, we also show the increase in branch mispredictions for VP_{LVP}. Since the value misprediction rate for this scheme is higher than VP_{Magic}, the increase is more pronounced in this case (e.g., *m88ksim* and *vortex*).

- **Recovering useful work from squashes:** One way IR reduces branch misprediction penalty is by recovering useful work from the control-squashed instructions. In Table 5, we show the percentage of executed instructions that are squashed due to branch misprediction, and percentage of

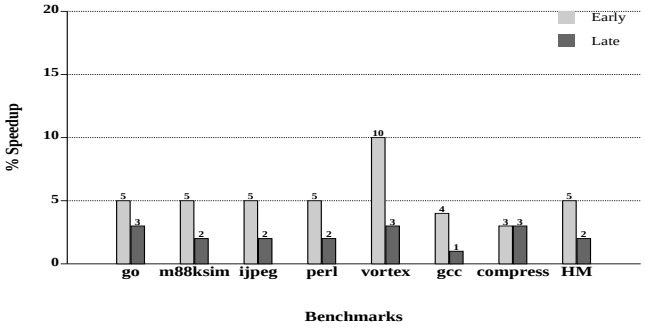

Figure 3: Performance benefits of early validation. Bars HM show the harmonic mean over all benchmarks.

Benchs	Increase in Branch Squashes due to Value Misprediction (%)			
	VP_{Magic}		VP_{LVP}	
	ME-SB	NME-SB	ME-SB	NME-SB
go	20.0	17.1	37.8	37.2
m88ksim	3.4	2.9	102.9	99.8
ijpeg	3.3	3.1	31.9	31.8
perl	30.3	22.0	39.4	37.9
vortex	54.4	51.8	164.5	160.4
gcc	16.4	14.1	50.9	49.5
compress	1.5	1.5	30.6	30.6

Table 4: Percent increase in the number of control squashes due to spurious branch mispredictions.

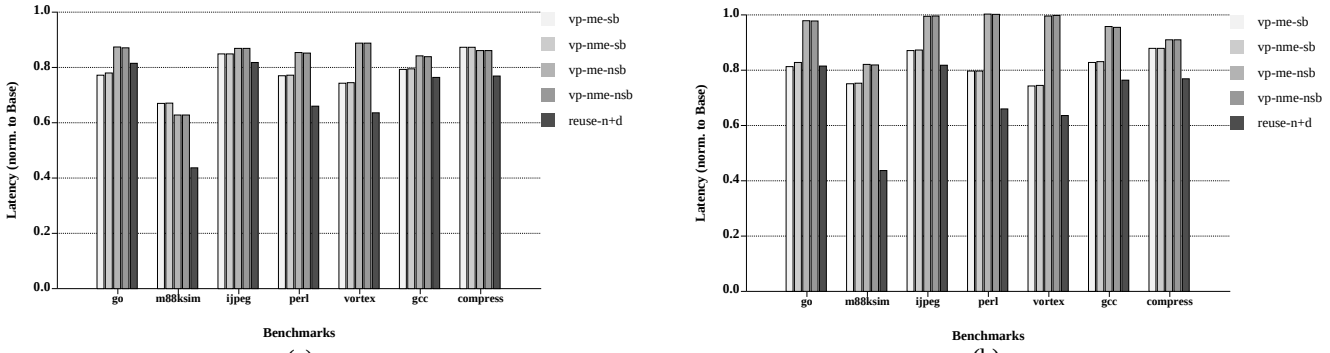

Figure 4: Branch resolution latency (normalized to the base case) for different configurations of VP and IR. (a) 0-cycle VP-verification latency. (b) 1-cycle VP-verification latency. The bars for reuse-n+d are same in (a) and (b).

such instructions that are recovered using IR. We see that a significant amount of squashed work is recovered for all benchmarks (for most benchmarks more than 30% of the squashed executed instructions are recovered). This contributes towards reducing the branch misprediction penalty.

- **Branch resolution latency:** In Figure 4, we show how VP and IR affect branch resolution latency. We define *branch resolution latency* as the time between when a branch is decoded and when it is finally resolved (*i.e.*, action on its outcome taken). Smaller this latency, the better, because then mispredicted branches can be detected sooner, reducing the misprediction penalty. In IR, if a branch is reused, it gets resolved immediately, resulting in a resolution latency of zero cycle. On the other hand, in VP with NSB configuration, a branch is not resolved till its operands become non-value-speculative, thus delaying branch resolution by the latency of verifying the value prediction. In Figure 4, we show the branch resolution latency normalized to the base case. We show results for 0- and 1-cycle VP-verification latency in figures (a) and (b); the bars for IR are same in both graphs.

As seen from the figure, both VP and IR reduce branch resolution latency; but IR does so to a greater extent. As would be expected, the SB configuration reduces this latency more than the NSB configuration. We also note that with 1-cycle VP-verification latency (Figure 4(b)), for several benchmarks (*ijpeg*, *perl* and *vortex*) the reduction in branch resolution latency is negligible.

Benchs	Inst Executed (millions)	Exec Inst Squashed (% of Inst Exec)	Squashed Inst recovered (% of Inst squashed)
go	450.4	15.0	36.6
m88ksim	543.5	4.9	53.9
ijpeg	454.8	2.5	49.4
perl	530.7	4.7	33.8
vortex	560.9	1.2	29.8
gcc	466.8	5.7	35.3
compress	490.8	9.8	27.7

Table 5: Percent of executed instructions squashed due to branch misprediction, and percent of such squashed instructions recovered by IR.

4.2.3 Impact on Resource Contention

In this section, we quantify impact of VP and IR on the contention for resources (e.g. functional units, data cache ports, writeback bus etc.). We estimate resource contention by counting the number of times resources are not available for executing the ready instructions, and dividing this by the total number of requests made for resources. The bars shown in Figure 5 are normalized to the base case. The results shown are for 0-cycle VP-verification latency; the results for 1-cycle VP-verification latency are similar. As pointed out in Section 3.2, VP and IR may affect contention in both ways — they may increase or decrease contention. We see that in most cases IR reduces resource contention. It increases contention slightly for *go* and *perl*. On the other hand, VP increases contention in all the cases. The increase is specially significant in *compress*, *go*, *perl* and *vortex*.

We also observe that resource contention is unaffected by multiple re-executions (resource contention for ME and NME are same). This result would be expected from the percentage of dynamic instructions that execute multiple

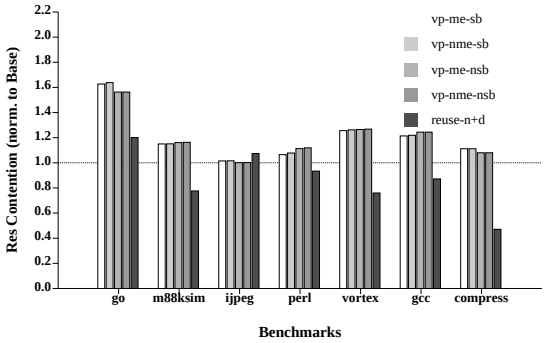

Figure 5: Resource contention (normalized to base). The line at 1.0 stands for the resource contention in the base case.

Benchmarks	% inst # times executed		
	1	2	3
go	94.4	4.9	0.7
m88ksim	97.6	2.3	0.1
ijpeg	98.9	1.0	0.1
perl	98.3	1.6	0.2
vortex	98.5	1.5	0.0
gcc	96.3	3.3	0.4
compress	99.6	0.4	0.0

Table 6: Percent of dynamic instructions that execute once, twice, and thrice. The numbers are for VP_{Magic} configuration ME-SB with 1-cycle VP-verification latency.

times, shown in Table 6. We see that very few instructions (< 0.5% in most cases) execute more than twice (similar results are also presented in section 5 of [7]), and hence, restricting the number of executions per instruction to 2 (as done by NME configuration) does not benefit much.

4.2.4 Net Performance

VP_{Magic} and IR

In Figure 6, we show the impact on performance of the two techniques. In the figure, we show the speedups over the base case (IPC/IPC_{base}) for the VP and IR schemes. Figures (a) and (b) show speedups for 0- and 1-cycle VP-verification latency respectively; IR speedups shown in both the figures are the same.

We make some observations from the results in Figure 6. First, the VP performance is sensitive to when branches are resolved. The configurations SB perform better than configurations NSB for all benchmarks, except *perl*, because SB resolves branches earlier than NSB (Figure 4). Also, since the value prediction rates for VP_{Magic} are high (and misprediction rates are low), the negative effects of spuriously mispredicted branches in SB (which are small for most benchmarks) are more than offset by the benefits of resolving branches sooner (except in *perl*, where the spurious mispredictions are high, and configuration NSB performs slightly better).

Second, as expected from Table 6, we observe that the impact of the multiple executions due to value mispredictions is negligible. The slight benefit seen in the case of *go* is because the NME-SB configurations reduce the number of spurious mispredictions (Table 4) by restricting the number of re-executions per instruction.

Third, as would be expected, increasing the VP-verification latency from 0-cycle to 1-cycle decreases the performance benefit for VP. But, interestingly, the increase affects the NSB configurations more than the SB configurations. This happens because in the NSB configurations the branches have to wait longer (due to 1-cycle VP-verification latency) for their operands to become non-speculative (Figure 4 (b)).

Fourth, we observe that for some benchmarks (e.g., *ijpeg*, *perl*, *vortex*), even though the reuse rate is less than the prediction rate, IR performs better than VP. This is due to the combined effect of early validation (Figure 3), recovering work from control squashes (Table 5), and reducing the branch resolution latency (Figure 4). Another advantage of IR is that it does not incur any misprediction penalty.

VP_{LVP}

To further study the sensitivity of VP to the way branches are handled, we show the performance numbers for VP_{LVP} in Figure 7. We show results for both 0- and 1-cycle misprediction penalty in figures (a) and (b) respectively. We note that, since in VP_{LVP} we only store one instance per instruction, its results should not be compared against the IR results presented earlier in this section (where up to 4 instances per instruction are stored). From Figure 7, first, we see that with prediction accuracies of VP_{LVP}, VP may actually perform worse than the base case. For all benchmarks,

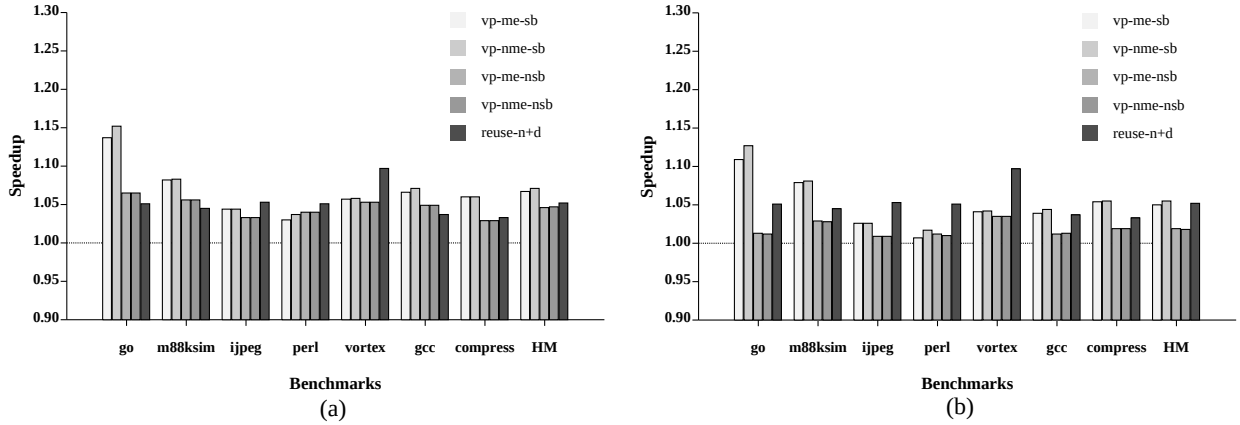

Figure 6: Speedups with VP_{Magic} and IR scheme S_{n+d}. (a) 0-cycle VP-verification latency. (b) 1-cycle VP-verification latency. Bars HM are the harmonic mean over all the benchmarks.

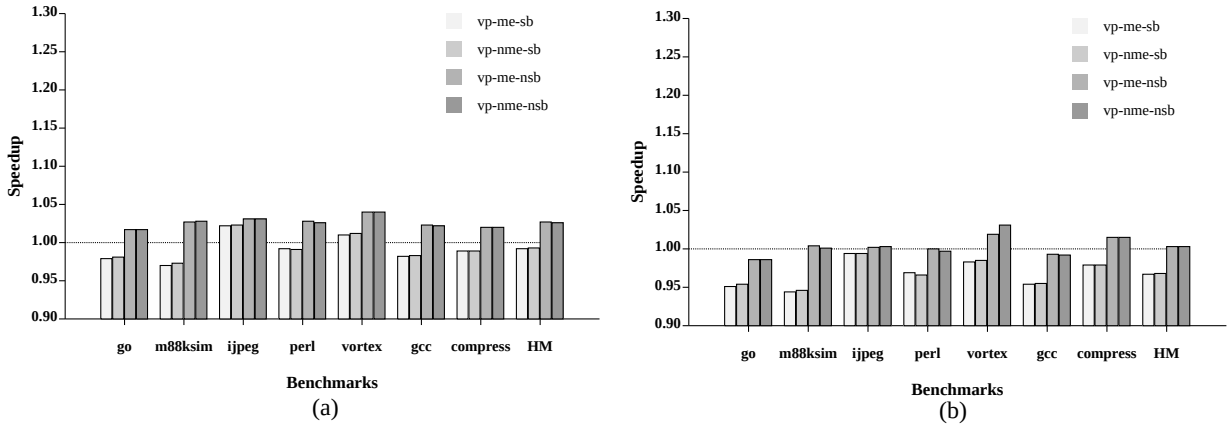

Figure 7: Speedups with VP_{LVP}. (a) 0-cycle VP-verification latency. (b) 1-cycle VP-verification latency. Bars HM are harmonic means over all benchmarks.

with configurations SB we see a degradation in performance. This is because the negative effects of spurious branch mispredictions (Table 4) are not offset because of the low value prediction accuracies of VP_{LVP}. The benefits of VP are further reduced when the VP-verification latency is increased to 1-cycle (Figure 7(b)).

Second, unlike for VP_{Magic}, in this case the configuration NSB works better than the configuration SB, implying that for cases where value mispredictions are higher it is more beneficial to delay branch resolution until the operands become non-value-speculative than to resolve them value-speculatively. Results in Figure 6 and 7 also indicate that a particular way of handling branches may not be the best policy for all cases; high and low value misprediction accuracies may warrant different treatment for branches.

4.3 Amount of Redundancy Captured

In this section, we try to get a feel for how restrictive is IR. To do so, we first estimate the total redundancy in programs and then determine what fraction of that redundancy can be captured by IR.

To estimate the redundancy, we buffer dynamic instances of every static instruction generated during a program execution (limited to 10K instances per static instruction), and classify each result-producing dynamic instruction into one of the following three categories: (i) *unique* — if it produces a result for the first time, (ii) *repeated* — if it produces the same result again, and (iii) *derivable* — if it produces a result that can be determined from the results it had produced earlier (e.g., instructions whose results fall on a stride. Once the stride size is known their subsequent results can be derived). We define *redundancy*, or *redundant instructions*, as the sum of the repeated and the derivable instructions. This measure of redundancy also provides a rough upper bound on the number of instructions that can be value predicted.[3]

In Figure 8, we show the above instruction categories for the benchmark programs.[4] We see that few (< 5%) instructions produce unique results, most (80% to 90%) produce repeated results, and few (< 5%) produce derivable results.

[3] This upper bound does not include the cases where VP may correctly predict some unique instructions by chance. Anyhow, the number of such cases will be small because number of unique instructions is small.

[4] In the figure, we have a category *Unaccounted*, which represents the instructions that could not be buffered (and hence could not be accounted for) because we only cached 10K instances per static instruction.

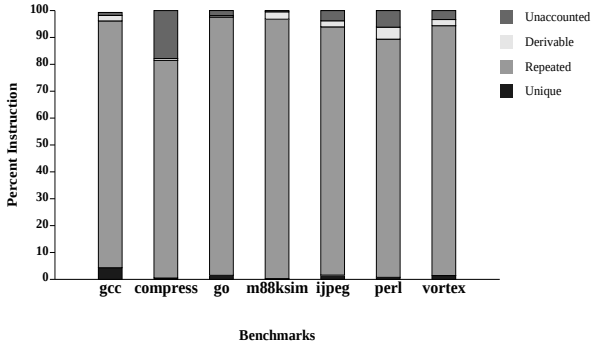

Figure 8: Classification of instruction results into *unique*, *repeated*, and *derivable* results. *Unaccounted* are the values which could not be accounted for due to limited buffering.

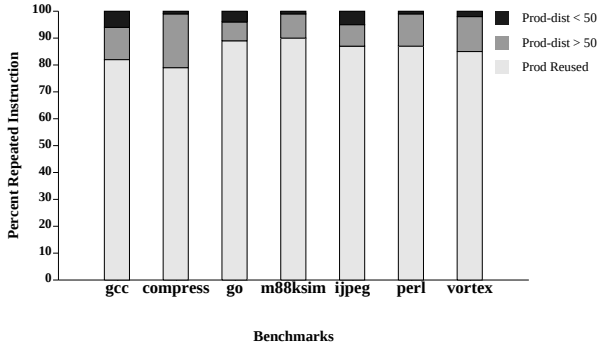

Figure 9: Categorizing the repeated instructions for determining whether their inputs are ready.

Next, we estimate how many of the redundant instructions are reusable. We note that, IR can reuse repeated instructions, but not derivable instructions. However, not all repeated instructions are reusable; as mentioned earlier, instructions are not reused if (i) their input operands are not ready at the time reuse test is done, or (ii) input operands are different. To estimate the number of reusable instructions, we subtract the number of times the above two cases occur from the total repeated instructions.

We assume that the inputs of an instruction are not ready if their producer instructions are less than 50 instructions ahead, unless the producer instructions themselves are reused. In Figure 9, we categorize repeated instructions into three categories: (i) instructions whose producers are reused (inputs are ready), (ii) instructions with unreused producers greater than 50 instructions ahead (inputs are ready), and (iii) instructions with unreused producers less than 50 instructions ahead (inputs are not ready). As shown in the figure, for most repeated instructions the inputs become ready because their producer instructions are reused. Only for less than 10% of repeated instructions the inputs are not ready (Prod-dist < 50). This is contrary to the expectation that most of the times inputs may not be ready early in a pipeline (where the reuse test is done).

Due to lack of space we do not present separately the number of repeated instructions that are not reused because of different inputs. In Figure 10, we show the net of the redundant instructions that can be reused. The bar labelled "redundant" is the sum of the bars *repeated* and *derivable* in Figure 8. As seen from the figure, most (84-97%) of the redundant instructions in programs are amenable to reuse. Thus, the approach that IR uses for detecting reuse does not significantly restrict its ability to capture redundancy present in programs.

5. Summary and Conclusions

In this paper, we attempted to understand the differences between the two recently proposed hardware techniques — Value Prediction (VP) and Instruction Reuse (IR) — that exploit the redundancy present in programs to collapse the critical path of the computation. The purpose of this work was to gain insight into the working of these techniques and to understand their interactions with other microarchitectural features. We believe that a better understanding would help in designing other mechanisms (which may be hybrid of VP and IR) that exploit redundancy in programs more effectively.

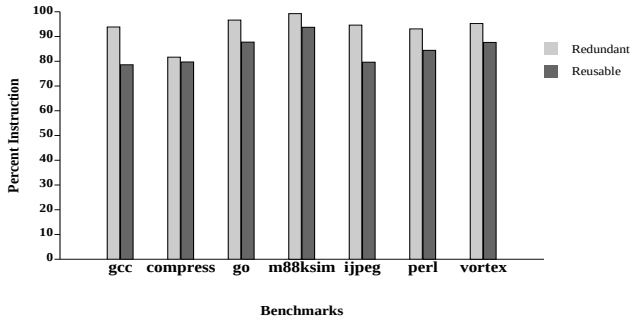

Figure 10: Amount of redundancy that can be reused.

We identified the key difference between the techniques: IR validates results before use (*early validation*), while VP validates results after use (*late validation*). We highlighted how these techniques differ in their interaction with other microarchitectural features and attempted to understand the differences in performance based on these interactions.

Our results showed that the performance obtained using VP is sensitive to the way branches that execute with value-speculative operands are *resolved* (*i.e.*, action taken based on their outcome). We evaluated two ways of resolving branches: (i) resolving them immediately after they execute, and (ii) resolving them only after their operands become non-value-speculative. In the first case, the branches get resolved sooner, but they also get mispredicted spuriously. In the second case, the branch resolution is delayed, which reduces the benefits gained by VP. Our results showed that a particular way of resolving branch may not be the best policy for all cases; for lower value misprediction rates the first policy works better, while for higher value misprediction rates the second policy works better.

Although IR captures less amount of redundancy (for the same amount of hardware storage), it performed better than the VP schemes studied for some benchmarks. The performance advantage of IR stems from, early validation of results, recovering useful work from branch misprediction, and reducing the branch resolution latency. Another advantage of IR is that it is non-speculative, and hence it does not incur any misprediction penalty.

Finally, we estimated a limit on how much redundancy present in programs can be captured by IR. We found that most (84-97%) of the redundancy can be reused. Thus, the approach of detecting redundant instructions based on their operands, non-speculatively, does not significantly restrict IR's ability to capture redundancy present in programs.

Acknowledgments

We thank Andreas Moshovos, Eric Rotenberg, Yiannakis Sazeides, and the referees for their helpful comments on this work. This work was supported in part by NSF Grants MIP-9505853, the U.S. Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order no. D346, and a donation from Intel Corp. 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.

[2] F. Gabbay and A. Mendelson. Speculative Execution based on Value Prediction. Technical Report EE Department TR 1080, Technion - Israel Institute of Technology, Nov. 1996.

[3] F. Gabbay and A. Mendelson. Using Value Prediction to Increase the Power of Speculative Execution Hardware. *ACM Transaction on Computer Systems (TOCS)*, Aug. 1998.

[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 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Sept. 1996.

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

[7] E. Rotenberg, Q. Jacobsen, Y. Sazeides, and J. E. Smith. Trace Processors. In *Proc. of 30th Annual International Symposium on Microarchitecture*, pages 138–148, Dec. 1997.

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

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

[10] A. Sodani and G. S. Sohi. An Empirical Analysis of Instruction Repetition. In *Proc. of 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

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