A study of value speculative execution and misspeculation recovery in superscalar microprocessors

Huiyang Zhou, Chao-ying Fu, Eric Rotenberg, Thomas M. Conte
Department of Electrical and Computer Engineering
North Carolina State University

Abstract

Recent research has shown that value prediction is a promising way to collapse the true data dependencies. To fully exploit the potential of value speculation, however, a highly accurate value predictor and efficient architectural support for value speculative execution are both necessary. In this paper, we address in detail value speculative execution in a generalized superscalar model based on MIPS R10000. In particular, in order to minimize misprediction recovery penalties, the selective reissuing mechanism is discussed and the necessary support at each pipeline stage are described. In this study, three important design issues are highlighted including: 1) when to resolve/verify the prediction; 2) where to keep the dispatched instructions after they are issued speculatively to enable reissuing; 3) how to dynamically construct the data dependence chain. The critical paths in the possible implementation are also identified. Then, potential benefits on the performance are discussed with different reissue latencies and value misprediction rates. Our experimental results show that there is a great speedup potential, on average up to 12% in our 4-way issue superscalar model, once highly accurate value predictors are available, and also it is shown that value speculation can achieve much higher speedups with better branch predictors and wider issue bandwidth.

1. Introduction

As the modern superscalar microprocessors become deeply pipelined with wider window sizes, speculative execution is crucial for performance enhancement. In particular, great performance gains can be obtained by collapsing the true data dependencies with the use of value prediction, i.e., value

speculative execution. In order to fully exploit the potential of value speculation, however, a highly accurate value predictor and efficient architectural support for value speculative execution are necessary.

Previous research [1, 2, 12] has shown that the outcome of many dynamic instructions is highly predictable, and several value predictor designs [1, 2, 3, 12, 13, 14, 15, 16] have been proposed to exploit the characteristics of the dynamically produced data sequences so as to produce high prediction accuracies. However, as is the nature of speculative execution, there are misprediction recovery penalties whenever a value misprediction occurs and the mispredicted value has been consumed. To balance the benefits of value prediction with misprediction recovery penalties, some selective value prediction techniques, either hardware-based dynamic techniques [4, 9, 12] or compiler-based static techniques [5, 6], have been proposed to predict only the values whose benefits of speculation outweigh the potential risk of misprediction. It is also important to note that value prediction can improve the branch prediction accuracy, as discussed in [7]. With all the previous research, it is desirable to perform a detailed study on superscalar pipeline support for value speculative execution, highlighting the important design issues, and stressing the potential benefits from value speculative execution.

In this paper, we present our study of value speculative execution in a generalized superscalar model based on MIPS R10000 [8]. Different misprediction recovery schemes are discussed, and in particular the selective reissuing mechanism is studied in detail. It is shown that in order to design an efficient selective reissuing mechanism, three important issues should be considered. The first is *when to resolve (i.e., verify) the prediction* since the execution results of the instruction may depend on another predicted value. The second is *where to keep the dispatched instructions after they are issued speculatively* to enable the reissuing. And the third is *how to dynamically construct the data*

dependence chain in order to reissue only the dependent instructions. We then discuss the necessary support at each pipeline stage to enable such a selective reissuing scheme and point out the potential critical paths. Our experimental results show that 1) there is a great speedup potential to exploit by using value speculative execution; 2) high value prediction accuracy can help tolerate some latency in the reissue critical path, and the latency is more critical when the misprediction rate is significant; 3) wider issue bandwidth and larger window sizes provide higher potential for being exploited by value speculative execution; 4) higher speedups can also be achieved from value speculation with better branch prediction accuracies. In any case, a high-accuracy value predictor is essential to achieve performance gains.

The remainder of the paper is organized as follows. Section 2 briefly describes the microarchitecture of our generalized superscalar model based on MIPS R10000. Value speculative execution and the recovery schemes are discussed in Section 3. Simulation methodology and result analysis are presented in Section 4, and Section 5 discusses the related work. Finally, Section 6 concludes the paper and discusses the future work.

2. The generalized superscalar model based on MIPS R10000

In order to examine value speculative execution and its impact on the superscalar pipeline, we use a generalized superscalar model based on MIPS R10000 in this paper. The structure of such a microprocessor can be modeled as shown in Figure 1.

The left hand side in Figure 1 includes an active list, a free list, an architectural map, a speculative map, and several shadow maps. This can be viewed as the control mechanism in the microarchitecture as there is no actual data movement. The free list maintains a list of unused physical registers, available for renaming the destination registers of newly fetched instructions. The map table keeps the speculative renaming information and the architectural map reflects the architectural register map state,

which is used to provide the precise state at the time of exception handling. The shadow maps are used to recover the renaming state from a branch misprediction. The active list records all instructions that are active within the processor. For each instruction, it keeps the current mapping and the previous mapping of the destination registers, which are used to update the architectural map and free the previously mapped physical register when the instruction retires. In the right hand side of Figure 1, the instruction moves down from the frontend (the fetch engine and I-cache) into the issue queue and the actual data moves into/out of the physical register file.

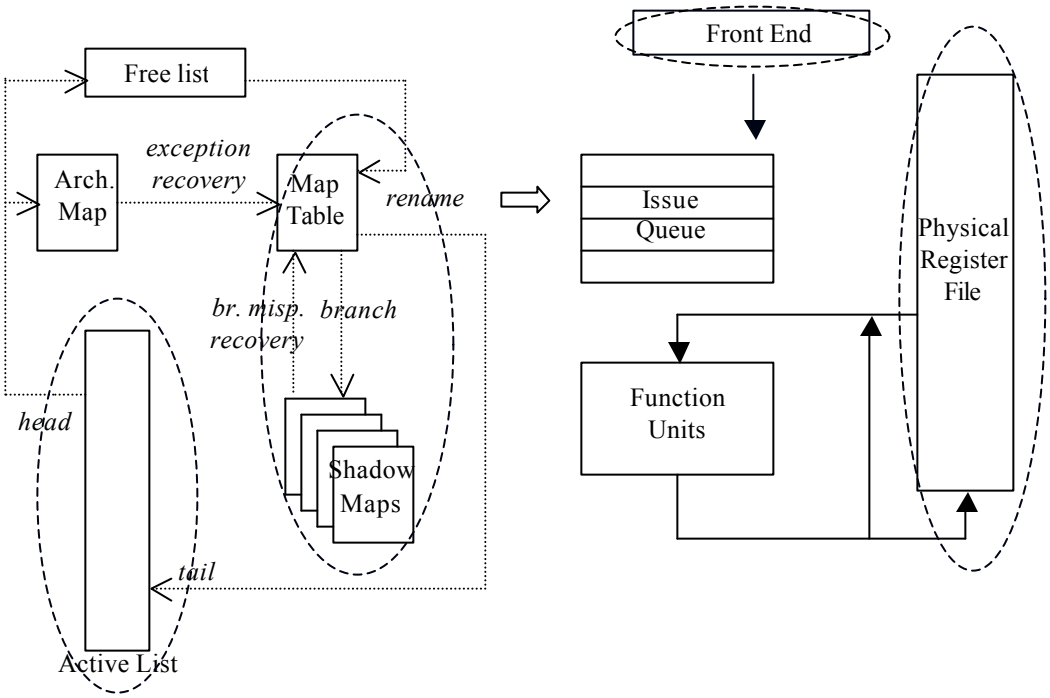

Figure 1. The microarchitecture of our generalized superscalar model based on MIPS R10000 (the parts in dotted lines are to be modified to support value speculative execution, described in Section 3.4)

In the case when there is no value speculative execution, the pipeline stages and their operations can be summarized as follows:

a) Instruction fetch (IF) stage: fetch instructions from L1 instruction cache (I-Cache); generate the next program counter from the branch predictor and the return address stack (RAS).

b) Instruction dispatch (ID) stage: decode the instructions and rename the registers to remove false data dependencies, dispatch the instruction into the issue queue and append the instruction into the active list.

c) Instruction issuing (IS) stage: check the source operands, if they are ready, the instruction can be issued when there is available issue bandwidth.

d) Register read (RR) stage: read the data from the physical register file.

e) Execution (EX) stage: execute the computation. For the memory operations, this stage can be further divided into address generation (AGEN), memory stage 1 (MEM 1), memory stage 2 (MEM 2) to perform the load and store operations.

f) Write back (WB) stage: write the computation result back into the physical register file.

g) Retire (RET) stage: commit the changes to the architectural state by updating the architectural map, freeing the physical register, and moving the instruction out of the active list if the instruction reaches the head of it.

These pipeline stages are decoupled by the issue queue and the active list, (or called a reorder buffer-ROB), whose size determines the dynamic window size used for superscalar execution.

3. Value speculative execution and misspeculation recovery schemes

In this section, value speculative execution and misspeculation recovery schemes are discussed in detail. We begin with the easy case of perfect prediction. The misprediction and recovery schemes are then introduced and several important design issues are highlighted in order to design an efficient recovery scheme.

3.1 Value speculative execution with perfect prediction

If we assume the perfect confidence in our value predictor (i.e., the predicted value is known to be correct), there is little additional work to do to support value speculative execution.

Considering the pipeline stages listed in Section 2, we need to add a value predictor in the front-end and access this predictor at the IF stage. If the prediction is made, then we put this value into the physical register file and mark the destination register as ready so that its dependent instructions (called *value prediction consumer instructions* [4]) can be issued without waiting for the completion of the current instruction (called the *value prediction producer instruction*). In other words, the consumer instructions can be executed speculatively while the producer instruction still takes its normal path in the pipeline. When the value prediction producer instruction completes at the WB stage, it compares the result with the prediction and updates the value predictor accordingly. Since we assume the perfect prediction model, this verification result is always true. Note that in this discussion, the predicted value is stored into the physical register file at the IF stage and compared with the actual result at the WB stage, this complicates the register file design since it requires reading the stored value and comparing it to the value that is to be written at the same WB stage. One way to alleviate this problem is to use an additional storage area for the predicted value. This method may result in a simpler register file design at the cost of area since the additional storage is virtually a duplicated physical register file keeping the speculative value information.

3.2 Resolving the value prediction

In the real cases when it is impossible for the value predictor to provide the perfect prediction, misprediction recovery is necessary. There are two recovery mechanisms considered in this paper: *complete squashing* and *selective reissuing*.

However, before we discuss the recovery schemes, we must first consider when do we resolve the value predictions to catch the potential misspeculations. This problem arises because the execution result may not be able to resolve a value prediction since the execution may depend on another predicted value (i.e., the execution result may be speculative itself). There are two ways to solve this

problem: *non-speculative resolution* and *speculative resolution*. Non-speculative resolution resolves the prediction until its source operands are not *value speculative*. Here, we use the term value speculative to describe the value that is either predicted or computed using a predicted value. Speculative resolution puts confidence in previous predictions and resolves the prediction once the execution is complete even if it uses some value speculative operands. These two methods are both possible for resolving the prediction but choosing one of them affects the processing in the case of correct predictions. In the non-speculative resolution scheme, if a prediction is verified to be correct, the corresponding dependent instructions' source operands need to be marked as non-speculative in order to enable the further resolution of those dependent instructions. In speculative resolution scheme, such processing is not necessary since it is assumed that the previous predictions are correct and the incorrect predictions will cause re-execution anyway. But the potential problem with speculative resolution is that the re-execution may happen more than once and this increases the misprediction recovery penalties. In this study, non-speculative resolution is chosen to minimize the recovery penalties and processing in the case of correct predictions is discussed.

3.3 Value misprediction recovery by complete squashing

When a value prediction is made at the IF stage, the producer instruction will not be able to resolve the prediction until it reaches the WB/RET stage and its own source operands are not value speculative. When a misprediction is detected and the mispredicted value has been consumed by dependent instructions, a recovery is inevitable to ensure the correct execution.

Complete squashing works by flushing all the instructions following the misspeculated instruction from the issue queue and the active list, and then re-fetching those instructions from the cache. This process is identical to the branch misprediction recovery mechanism. Similarly, a shadow map needs to be allocated to maintain the correct renaming state when a value prediction is made. Multiple shadow

maps are required in order to make multiple value predictions in the current instruction window. One way to eliminate the requirement of multiple shadow maps is to back up to the latest branch instruction when a value misprediction occurs. This would result in more recovery penalties since it requires more instructions to be flushed and refetched.

Since the complete squashing method takes the same approach as the branch misprediction recovery, it has a similar high recovery penalty in terms of the clock cycles. However, this high recovery penalty is not necessary since value misprediction is different from the branch misprediction. Branch misprediction will result in the wrong path of the execution; therefore it should be backed up to take the right path. Value misprediction, on the other hand, does not necessarily change the execution path or the renaming information. This can be further explained with the example shown in Figure 2.

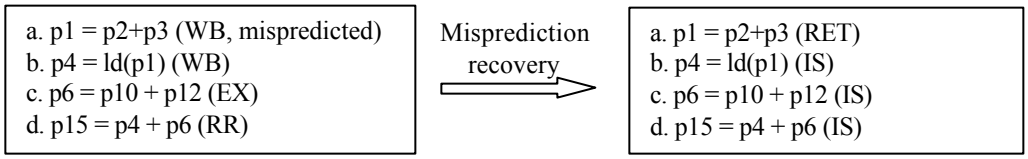

Figure 2. An example of complete squashing (The register index starting with 'p' denotes the physical register)

In Figure 2, instruction *a* makes a misprediction on register p1, which is consumed by instruction *b* directly and instruction *d* indirectly. When instruction *a* finishes the execution and resolves the misprediction, the instructions *b*, *c*, and *d* are flushed out of the window, re-fetched and re-dispatched into the issue queue. From this example, it can be seen that there are two reasons for the unnecessarily high recovery penalty. First, data independent instructions, like instruction *c* in our example, are unnecessarily flushed out. Second, the re-fetching increases the workload of the frontend and the re-dispatch repeats the renaming process, which is unnecessary as well.

Also if we use non-speculative resolution discussed in Section 3.2, there is additional processing in the case of correct prediction. This will require hardware support similar to what will be discussed in Section 3.4 for selective reissuing. Since the idea of the complete squashing is to simplify the

hardware at the cost of recovery penalties, speculative resolution may be more appropriate for the complete squashing scheme.

3.4 Value misprediction recovery by selective reissuing

From the analysis of the previous example in Figure 2, it can be seen that the ideal recovery scheme must only put the dependent instructions back into the issue queue and must not stall the independent instructions after a value misprediction. Using the same example as Figure 2, the recovery using selective reissuing is shown in Figure 3.

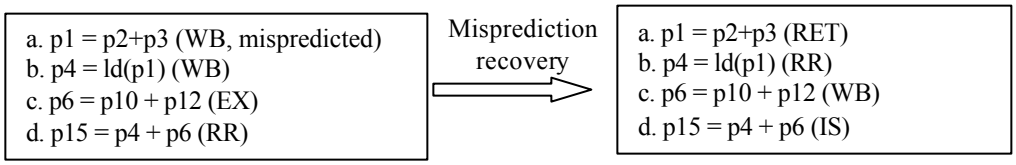

Figure 3. An example of selective reissuing

After the misprediction is resolved, instructions *b* and *d* are put back into the issue queue and the independent instruction *c* continues execution in the pipeline. Three problems are observed from this example: 1) the decoded and renamed instructions must be kept after being issued in order to enable selective reissuing; 2) there must be enough empty entries in the issue queue for the instructions to be reissued; 3) the data dependence must be identified dynamically to select only the dependent instructions. One way to handle the first problem is to add a storage field in the ROB/active list to retain the dispatched instructions, so an instruction leaves the issue queue once it is issued, speculatively or non-speculatively. This method requires no changes in the issue logic and it is the approach used in this study. Another method is to keep the speculatively issued instructions in the issue queue until it is no longer speculative [17, 21]. This approach enables quick reissuing for multiple times and solves the second problem easily, but it limits the dynamic window size especially when a predicted value requires long latencies to resolve. For example, a load instruction with its destination register predicted will prevent all its dependent instructions from leaving the issue queue in

the case of a miss in the data cache. When the predicted value is verified, there is also a structural hazard since many completed instructions will compete to write back their execution results into the register file.

The second problem depends on the issue queue architecture. For processors with a single, shared issue queue, this is not a problem as long as the size of the ROB is the same as the issue queue. But for those processors with separate issue queues (e.g., separate issue queues for load/store and integer operations), there is a chance that some issue queues may not have enough empty entries for the instructions to be reissued since the size of the separate issue queues is less than the size of the ROB. One possible solution is to stall the fetch engine until all instructions that need to be reissued enter the issue queue. This may introduce additional penalties for recovery. In this study, a shared issue queue model is used. For the third problem, two possible solutions are considered. One is using an expanded ROB entry to maintain the producer information of each source operand and the other is to use a data dependency matrix. Both will be discussed in the following subsection.

a. Dynamic construction of data dependence chain

In order to get the producer information of source operands, the ROB entry number is used, as it is a unique identifier for the instructions in the window. Here, it needs to be noted that any other identifier can be used for the same purpose as long as it uniquely identifies the instruction. The ROB entry number is chosen in this paper since it helps to explain both methods of dependence chain construction. By keeping this number in the register map, the processor knows which instruction produces the current version of the architectural register. Then, the producer-consumer dependency can be constructed from the ROB entry number in the map table corresponding to the source operands of the consumer instructions. This can be illustrated using the earlier example in Figure 2, as shown in Figure 4.

As shown in Figure 4, the data dependence relationship between instructions can be determined through a match between source operands' ROB entry numbers and the earlier instructions' destination ROB entry numbers.

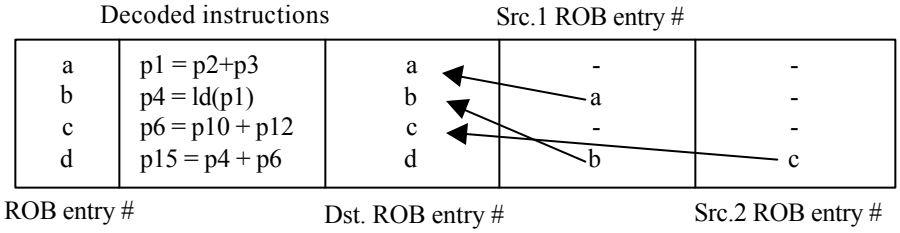

Figure 4. Data dependence chain construction

To facilitate this data dependence chain construction method, some modifications are made in the processor architecture shown in Figure 1. First, for each entry in the ROB/the active list, the fields are included to store the decoded instruction, the source operands' ROB entry numbers, and the flags to show whether the source and destination operands are speculative, which is shown in Figure 5. Then a field is added in the map table for the ROB entry number of the instruction that produces the register. The same change is also made in the shadow maps to restore not only the renaming state but also the data dependence state. With such modifications, the data dependence information can be constructed at the renaming stage, as described below. Another change in the architecture is the physical register file. We may either keep the speculative value in the same register file or use a different storage, as discussed in Section 3.1. In addition to the value, a flag is included to show that the value is speculative (i.e., predicted or computed using a predicted value), or non-speculative. All those structures that need to be modified in the processor are enclosed with a dotted line shown in Figure 1.

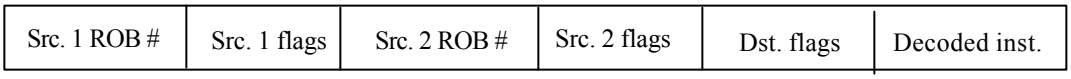

Figure 5. Fields added in the ROB entry

The dependency matrix approach uses an N-bit by N-bit matrix where N is the length of the ROB/active list. The rows and columns correspond to the ROB entries. Each row represents a bit mask

showing the instructions that the instruction in the ROB entry is dependent on. For the example in Figure 4, the matrix is shown in Figure 6. The bit in row 2 column 1 shows that the instruction *b* is dependent on the instruction *a*.

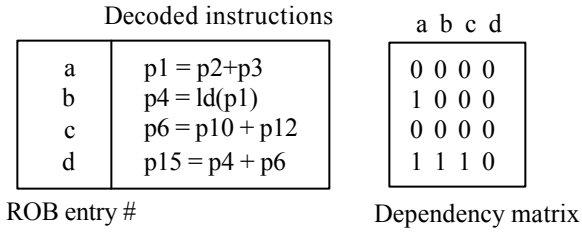

Figure 6. The data dependency matrix

This dependency matrix can be constructed in the dispatch stage using the ROB entry number in the map table. The current row in the dependency matrix can be obtained by setting the bits in the columns corresponding to the producer instructions of the source operands and then OR-ing it with the rows of the producer instructions. In Figure 6, the row corresponding to instruction *d* is constructed by setting bits in the columns corresponding to instructions *b* and *c*, and then OR-ing the rows corresponding to instructions *b* and *c*. This method simplifies the process of finding the dependent instructions, however, it requires additional work at the dispatch stage to construct such a matrix and also it is hard to tell from the matrix whether a dependency is direct or indirect. Such information may be useful when selecting which dependent instructions are to be put back into the issue queue first.

b. Pipeline execution support for selective reissuing

After such modifications are made in the processor microarchitectural model, additional operations are considered at each pipeline stage to support the selective reissuing. At the IF stage, the value predictor is accessed and a choice is made whether or not to predict the target value. At the ID stage, it reads the source operand's ROB entry number from the map table, and puts the ROB number of the current instruction into the entry of its destination register. Then, it stores the data dependence information either in the expanded ROB entry or the data dependency matrix. If the target value is

predicted, then the predicted value is stored, the ready flag of the target register is set and the flag of the value is set as speculative.

Based on the availability of the source operands (i.e., by checking the ready flag of the source registers), the instructions in the issue queue can be issued if issue bandwidth and the function unit are available. Here, it needs to be noted that the ready flag can be set speculatively at the ID stage with a value prediction or at the EX stage by the wake up and select logic when the result is computed using a speculative value. Instructions such as store, branch, function call, and return are not issued with a value speculative source operand. The reason is that value speculative execution of a store will require the dynamic identification of memory dependent load instructions to perform selective reissuing and value speculative execution of branches may result in the wrong execution path, which would require a complete squash. At the RR stage, when the speculative value is read out as the source operand, the destination value must also be marked as speculative. At the EX stage, there is no additional operation except for the load instruction. If the target of the load instruction is predicted, the ready flag of the destination register should be maintained in order to enable its dependent instructions to be issued speculatively when the data cache miss happens, since the data cache miss would reset the ready bit of its destination register and stop waking up its dependent instructions.

At the WB stage, a validation flag is set to show whether the real computation result is the same as the prediction if the target of the instruction is predicted. Then the instruction leaves the execution pipeline and enters the RET stage.

When the instruction is at the WB/RET stage, if its source operands are non-speculative, it processes the validation flag that is set at the WB stage. If the validation flag shows the correct value prediction, the instruction needs to broadcast its ROB entry number to find its dependent instructions. Then, the source operand of its dependent instructions would be set as non-speculative. After the setting, if the

dependent instruction's source operands are all non-speculative, the dependent instruction must verify its own result if it is at the RET stage. If the validation flag shows there is a mismatch between the predicted value and the non-speculatively computed result (i.e., a misprediction occurs), in addition to broadcasting its ROB entry number to find the dependent instructions, it requires a mechanism to put those dependent instructions back into the issue queue if they have already been issued. Here, finding the dependent instructions is similar to the process in the case of a correct prediction. The difference is that for correct predictions, the dependent instruction will broadcast its ROB entry number only if all of its source operands are non-speculative, and for the misprediction case, any source operand misprediction causes the instruction to broadcast its ROB entry number to invoke reissuing.

As we discussed earlier, there are two ways to keep the data dependence information. So there are two corresponding ways to find the dependent instructions. One is to broadcast the ROB entry number recursively and the other is to use a dependency matrix. If we use the recursive method (i.e., the instruction broadcasts its ROB entry number to find its immediate dependents and the dependent ones will broadcast their ROB entry number to invoke indirect dependent instructions, etc.), it is serial in nature. This part will then be the *critical path* for the selective reissuing mechanism. The use of the dependency matrix can alleviate this critical path penalty. Also, since multiple instructions need to be put back into the issue queue when a misprediction occurs, this will increase the pressure on the bandwidth of getting into the issue queue (i.e., how many of instructions can be put into the issue queue simultaneously). However, this pressure is not so critical since the issue bandwidth is more likely to be a bottleneck than the bandwidth to get in the issue queue. This surmise is based on the fact that the reissued instructions inherently have true data dependencies. So the performance would not degrade if we put the immediate (direct) dependent instructions back into the issue queue earlier than the indirect dependent instructions. Using this method, the pressure on the critical path of finding the

dependent instructions can be also relieved if the misprediction occurs infrequently so that there is no overlapping conflict on the broadcast buses.

Also, in the case of a misprediction, upon identifying the dependent instructions, the allocated resources of those instructions need to be freed, such as the result shift register and the memory ports. After the instruction processes the validation flags, it remains at the RET stage of the ROB until it is at the head of the ROB. When the instruction is to be retired, it checks the map table for its destination register's ROB entry number. If the rob entry number keeps unchanged after the instruction is dispatched, this number in the map table is invalidated since the producer is leaving the current window.

4. Simulation methodology and experimental results

In order to examine the impact of value speculative execution with the selective reissuing mechanism, our superscalar timing simulator [10, 11] is extended according to the microarchitectural modification and pipeline execution discussed in Section 3.4. Table 1 shows the base processor configuration. The entire SPECint95 benchmark suite is used in the simulation. Table 2 lists the input sets and all benchmarks were run to completion.

Table 1. Base Processor configuration

Instruction cache	Fetch bandwidth: 2-way interleaved to fetch full cache block
	Size/assoc/replacement = 64kB/2-way/LRU
	Line size = 16 instructions
	Miss penalty = 12 cycles
Data Cache	Size/assoc/replacement = 64kB/4-way/LRU
	Line size = 64 bytes
	Miss penalty = 14 cycles
Superscalar Core	Re-order buffer: 64 entries
	Dispatch/issue/retire bandwidth: 4-way
	4 fully-symmetric function units
	Issue mem bandwidth: 4
Execution latencies	Address generation: 1cycle
	Memory access: 2 cycle (hit in cache)
	Integer ALU ops = 1cycle
	Complex ops = MIPS R10000 latencies

Table 2. Benchmarks

Benchmark	Input dataset	Instr. Count
Compress	Compress95.ref	24 million
Gcc	-O3 genrecog.i –o genrecog.s	117 million
Go	9 9	133 million
Jpeg	Vigo.ppm	166 million
Li	Test.lsp (queens 7)	202 million
M88ksim	-c < ctl.in (dcrand.big)	120 million
Perl	scrabble.pl < scrabble.in (dictionary)	108 million
Vortex	vortex.in (persons.250, bendian.*)	101 million

In order to separate the impact of the selective reissuing from other factors such as value predictor designs and selective value prediction, we chose to predict all the register-defining instructions and vary the misprediction rate from 0% to 15%. Then, in order to simulate the confidence mechanism to find those highly predictable instructions, we randomly choose 70% of register-defining instructions to predict and vary the misprediction rate from 0.0% to 2% (70% is chosen since it is the average value prediction accuracy of the current value predictors [3, 6, 20]). The misprediction rate is simulated using a probabilistic approach by using a uniform distributed random number generator. If the result is in a certain range, then we deliberately use a wrong value as the predicted value, otherwise we use the correct value as the prediction. Also in our simulation, we choose not to issue the store, branch, function call and return instructions if any of the source operands is value speculative. The base results in IPC where no value speculative execution is allowed is shown in Table 3.

Table 3. The base model results in IPC

Benchmarks	compress	gcc	go	jpeg	li	M88ksim	perl	vortex	Average
IPC	1.53	1.87	1.62	1.97	2.09	1.74	1.91	2.30	1.85

In our simulator, we modeled the selective reissuing mechanism with different reissue penalties. In the ideal case when we assume that the reissue can be done in one clock cycle (i.e., the instructions to be reissued get into the issue queue in 1 clock cycle), the simulation results are shown in Figure 7. The average IPC is computed using the harmonic mean.

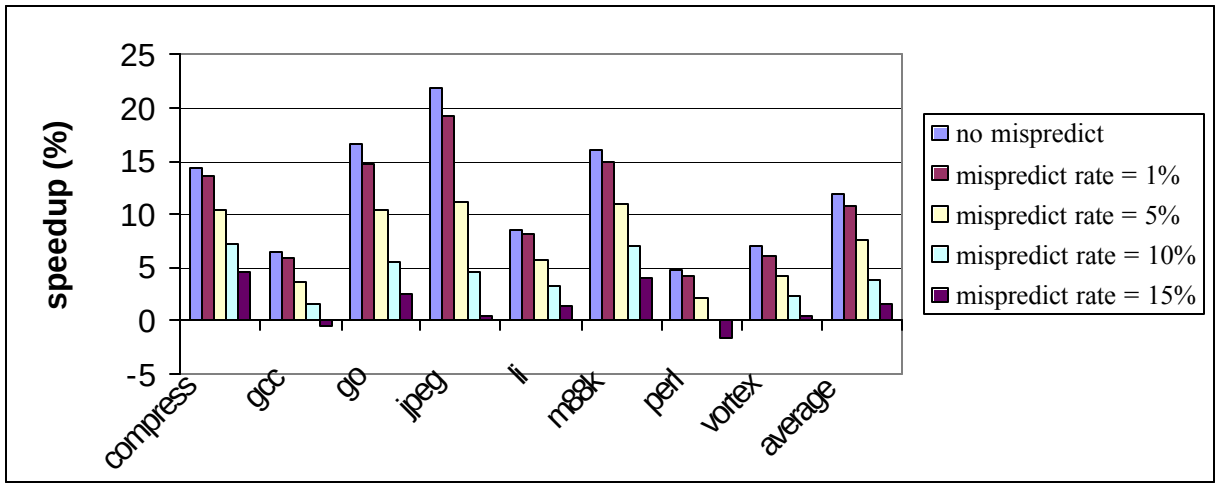

Figure 7. The speedup with the selective reissuing latency of 1 cycle

From Figure 7, it can be observed that pure data dependence has different impacts on different benchmarks. For benchmark *jpeg*, when the data dependence is broken down by prefect value prediction, there is 22% speedup. The benchmark *perl*, on the other hand, just has around a 5% speedup. This shows that the data dependence is more dominant in benchmark *jpeg* than *perl*. Compared with the speedup of an ideal prediction, the performance is pretty good when the misprediction rate is 1%. However, when the value misprediction rate increases further, the performance speedup decreases more dramatically. This means that the high prediction accuracy is crucial for value speculative execution even with the selective reissuing mechanism. Among the benchmarks, *compress* is most robust against the change of misprediction rate with a speedup of 14.4% for ideal prediction and a speedup of 4.6% for the misprediction rate of 15%. Overall, the average speedup drops from 11.9% to 1.6% when the misprediction rate increases from 0% to 15%.

When we increase the latency of selective reissue from 1 cycle to 2 cycles, the performance of value speculative execution depends more on the accuracy of value prediction, which is shown in Figure 8.

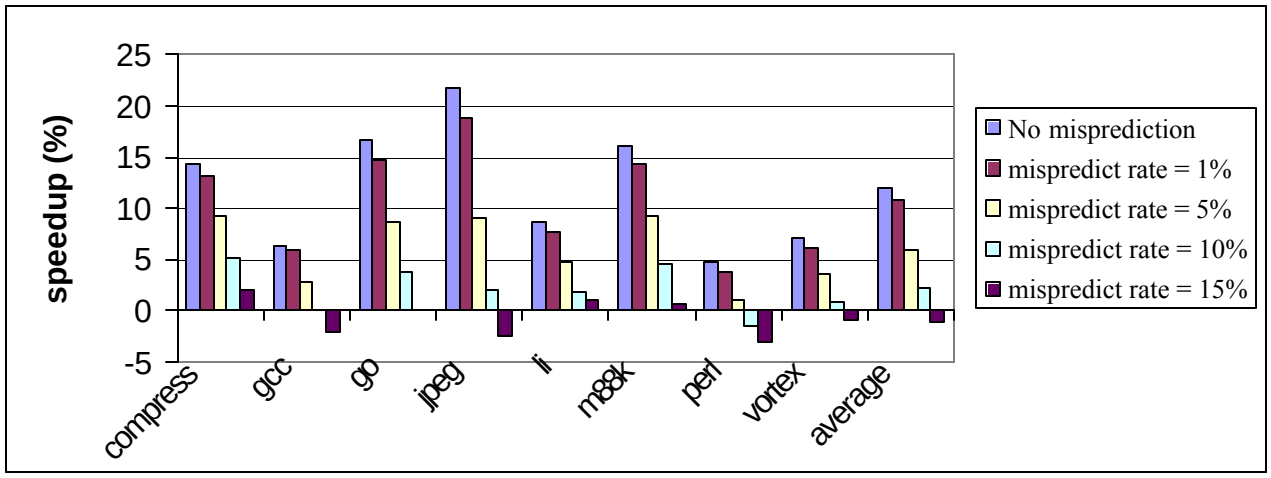

Figure 8. The speedup with the selective reissuing latency of 2 cycles

In Figure 8, when the misprediction rate is low, e.g., 1%, there is not much difference in performance when compared to the speedup shown in Figure 7. This means that with high value prediction accuracy, more delay along the critical path of the selective reissue mechanism can be tolerated. However, compared to the results shown in Figure 7, the speedup drops more significantly when the misprediction rate increases further. On average, the speedup drops from 11.9% to –1.1% when the misprediction rate increases from 0% to 15%.

We also investigate the relationship between the performance gain from branch prediction and value prediction. Here, the reissue latency is set to 2 cycles and the simulation results are shown in Figure 9.

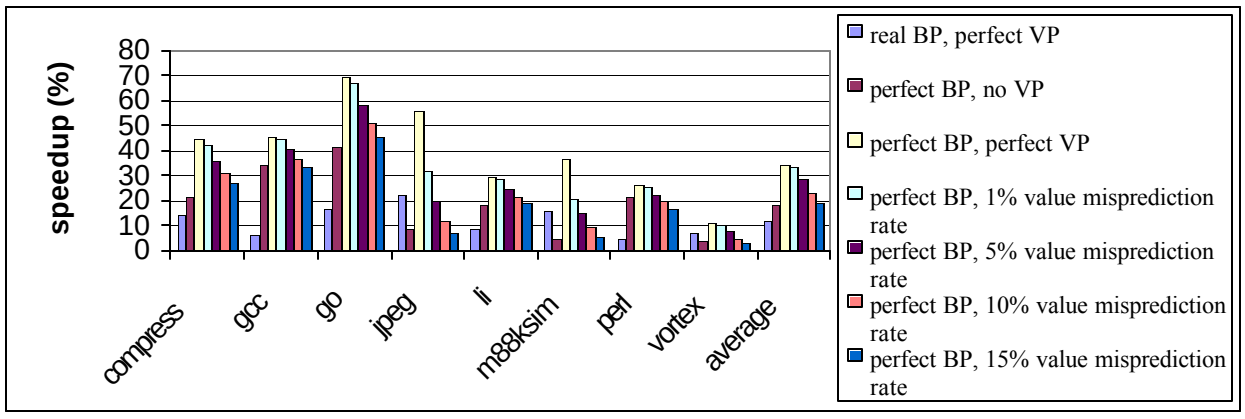

Figure 9. The speedup using branch prediction (BP) and value prediction (VP)

From Figure 9, it can be seen that in all benchmarks except *jpeg*, *m88ksim*, and *vortex*, control dependence is more dominant than data dependence. However, when we assume perfect value prediction and perfect branch prediction, the speedup is higher than the sum of the speedup resulted from a single perfect prediction. For example, the average speedup is 11.9% for ideal value prediction and 18.4% for perfect branch prediction. And the combined prefect prediction results in the speedup of 34.6%. This means that with more accurate branch prediction, higher speedup can be obtained from value prediction, which in turn may tolerate more value mispredictions for the same speedup. When we compare the speedup of cases with perfect branch prediction and different value misprediction rates, we can see that with perfect branch prediction, the speedup of a value misprediction rate of 15% is still slightly better than that without value prediction. The reason is that branch misprediction will squash value speculative execution result on the mispredicted path regardless of whether it is a correct value prediction or not. So a more accurate branch predictor will utilize value speculative execution more efficiently, resulting in the ability to tolerate higher value misprediction rates.

Another investigation in our study is the impact of the larger window size and the wider issue bandwidth. We increased the window size to 256 and increased the dispatch, issue, and retire bandwidth to 16. With the reissue latency set to 2 cycles, the simulation results are shown in Figure 10.

From Figure 10, it can be seen that with a larger window size and wider issue bandwidth, the speedup of value speculative execution is higher, which can be expected since the data dependence is more dominant and more issue bandwidth can be filled with the wider issue processors. The same conclusion is shared in [18]. Also, comparing to the results in Figure 8, it can be seen that the wider machine can tolerate a higher misprediction rate because there are more benefits from value speculative execution in wider issue machines with the same misprediction rate.

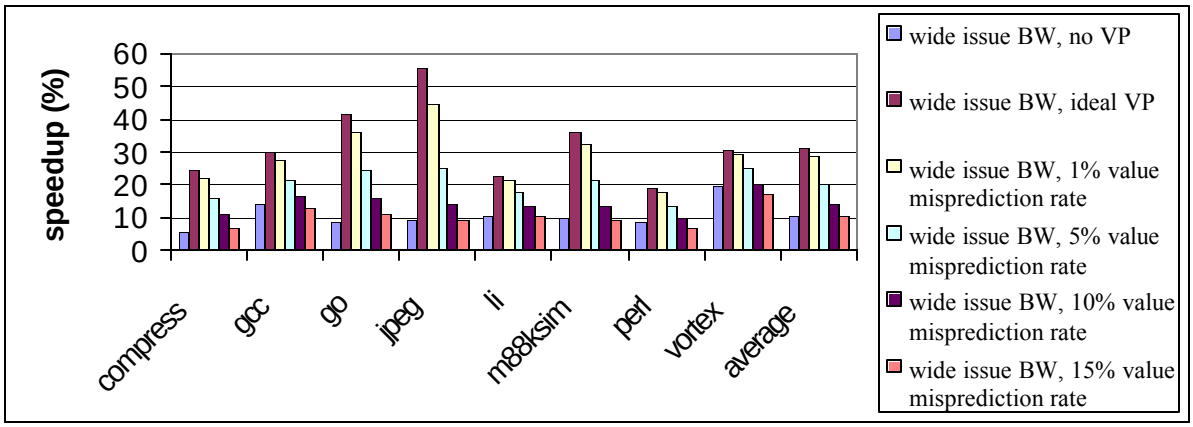

Figure 10. The speedup using a wider issue machine with value speculative execution

As the results show, the misprediction rate is highly crucial to attain speedup. Considering the current proposed value predictors, the accuracy is not high enough if all the instructions are to be predicted. So the confidence mechanism is necessary to filter out those hard-to-predict instructions. In our simulations, we randomly choose 70% of the instructions to predict and the results are shown in Figure 11 with the reissue penalty of 2 cycles. Comparing this to the results in Figure 8, it can be seen that the speedup of the confidence mechanism with misprediction rate less than 2% is greater than the speedup when predicting all instructions with a 10% misprediction rate and is less than the speedup when predicting all instructions with a 5% misprediction rate. This result shows that although the confidence mechanism is very useful, we still need *value predictors with better accuracies and larger coverage* to fully exploit the potential of the value prediction scheme and we expect this to be a great challenge in the research of this area.

5. Related work

As discussed in Section 1, several schemes have been proposed to design highly accurate value predictors and research has been performed for either hardware-based or compiler-driven selective value prediction to exploit the potential of value prediction more efficiently.

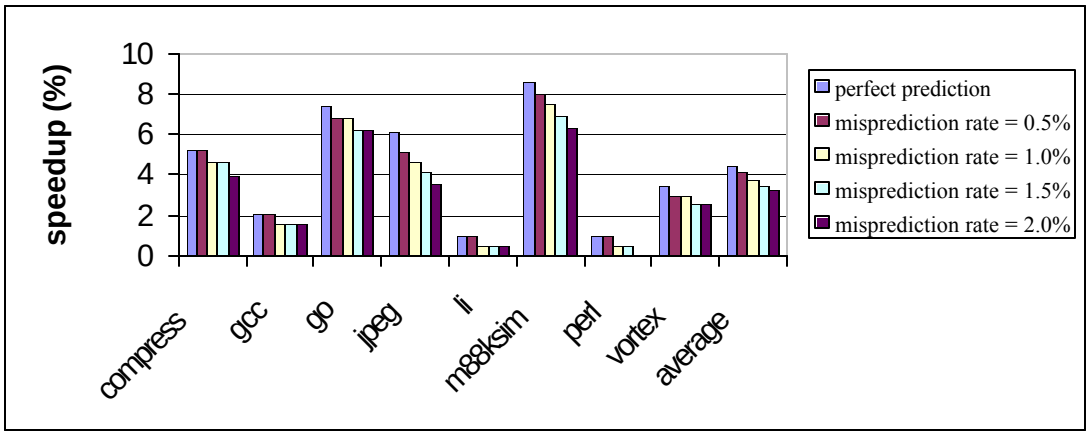

Figure 11. The speedup with the selective reissuing latency as 2 cycles and predicting 70% of instructions

For the misspeculation recovery, the idea of complete squashing and selective reissuing has been described briefly in [4, 12]. In [17], a selective reissuing scheme called selective reissue core (SRC) is developed. In the SRC scheme, a PowerPC style of superscalar processor is used as the baseline architecture and the instruction remains in the reservation station until it reaches the final (non-speculative) state, which enables quick reissue for multiple times. The SRC uses the non-speculative resolution method to validate the prediction and the data dependence chain is maintained through the register names contained in the reservation station. The SRC also introduces the speculative execution path and the promotion path to support speculative execution. The instruction issued to the speculative execution path will not broadcast its result in the result bus and if the prediction is correct, it will be issued to the promotion path to broadcast its result on result bus. The potential problem with SRC is the structural hazard. It is likely that the normal execution results and the validation results compete for the result bus. Also the dynamic instruction window size is limited as discussed in Section 3.4. Another selective reissue scheme (RUU) is described in [19], which expands the entry of the instruction window to support value speculative execution, and also depends on the result bus to find the dependent instructions to reissue in the case of misspeculation.

6. Conclusions

In this paper, a detailed study of value speculative execution is performed using a generalized superscalar model based on MIPS R10000 and possible recovery mechanisms including the complete squashing and the selective reissuing are examined. Three important issues are highlighted in designing an efficient recovery scheme: when to resolve the prediction, speculatively or non-speculatively, where to keep the dispatched instructions after they are issued speculatively so as to enable reissuing, and how to construct the data dependence chain dynamically. Necessary support at each pipeline stage is then discussed to enable such a selective reissuing scheme and the potential critical path is pointed out as finding the dependent instructions and putting them back into the issue queue. From our experiments, it shows that 1) there is a great speedup potential to exploit by using value speculative execution; 2) high value prediction accuracies can tolerate some latencies in the reissue critical path and the latency is more critical when the misprediction rate is significant; 3) wider issue bandwidth and larger window sizes provide more opportunities to be exploited by value speculative execution; 4) higher speedups can also be achieved from value speculation with better branch prediction accuracies. In any case, however, a high-accuracy value predictor is essential to the possible performance gains.

Our future work will investigate the effect of value speculation on different processor paradigms from superscalar processors, such as slipstream processors [22]. With the microarchitecture features of the slipstream processor, there exist chances to use value speculation for different purposes in addition to break the true data dependency in the program. Also, as it is pointed out in the paper, the design of a highly accurate value predictor is worthwhile to investigate. The associated issues including the prediction bandwidth and updating mechanisms also need to be exploited further.

References

[1] Y. Sazeides and J. E. Smith, "The Predictability of Data Values", In *30th International Symposium on Microarchitecture (MICRO)*, 1997

[2] M.H. Lipasti, C. B. Wikerson and J. P. Shen, "Value locality and load value prediction", In 7th International Conference on Architectural Support for Programming Language and Operation Systems (ASPLOS), Oct, 1996.

[3] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors", In *30th International Symposium on Microarchitecture*, 1997

[4] B. Calder, G. Reinman, and D. Tullsen, "Selective value prediction", In *26th Annual International Symposium on Computer Architecture (ISCA)*, May 1999

[5] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Value speculation scheduling for high performance processors", In *8th International Conference on Architectural Support for Programming Language and Operation Systems*, 1998

[6] E. Larson and T. Austin, "Compiler controlled value prediction using branch predictor based confidence", In *33rd International Symposium on Microarchitecture*, 2000

[7] T. Heil, Z. Smith, and J. E. Smith, "Improving Branch Predictors by Correlating on Data Values", In *32nd International Symposium on Microarchitecture*, Nov. 1999

[8] K. C. Yeager, "The MIPS R10000 superscalar microprocessor", *IEEE Micro*, 1996

[9] E. Tune, D. Liang, D. Tullsen, and B. Calder, "Dynamic prediction of critical path instructions", In *7th International Symposium on High Performance Computer Architecture (HPCA)*, 2001

[10] D. Burger and T. M. Austin, "The Simplescalar Tool Set Version 2.0", *Technical Report*, Computer Science Department, University of Wisconsin-Madison, 1997

[11] E. Rotenberg, Superscalar simulator with Simplescalar ISA, ECE dept, NC State

[12] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit via Value Prediction", In 29th *International Symposium on Microarchitecture*, 1996

[13] F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction?", In *30th International Symposium on Microarchitecture*, 1997

[14] M. Burtscher and B. G. Zorn, "Exploring Last n Value Prediction", In International Conference on Parallel Architectures and Compilation Techniques (PACT'99), 1999

[15] T. Nakra, R. Gupta, M. L. Soffa, "Global context-based value prediction", In *5th International Symposium on High Performance Computer Architecture*, 1999

[16] D. M. Tullsen, and J.S. Seng, "Storageless Value Prediction Using Prior Register Values", In *26th Annual International Symposium on Computer Architecture*, 1999

[17] B. Rychlik, J. Faistl, B. Krug, and J. P. Shen, "Efficacy and performance impact of value prediction", In *International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, 1998

[18] R. Sathe and M. Franklin, "Available parallelism with data value prediction", *5th International Conference on High Performance Computing (HIPC'98)*, 1998

[19] T. Sato, "Data Dependence Speculation Using Data Address Prediction and its Enhancement with Instruction Reissue", *24th Euromicro Conference*, 1998

[20] S. Lee and P. Yew, "On Some Implementation Issues for Value Prediction on Wide-Issue ILP Processors", In *International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, 2000

[21] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. "Trace Processors". In *30th International Symposium on Microarchitecture*, 1997

[22] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. "A Study of Slipstream Processors". In *33rd International Symposium on Microarchitecture*, December 2000