Value Prediction in VLIW Machines *

Tarun Nakra, Rajiv Gupta and Mary Lou Soffa
Department of Computer Science
University of Pittsburgh
{nakra,gupta,soffa}@cs.pitt.edu

Abstract

The performance of VLIW architectures is dependent on the capability of the compiler to detect and exploit instruction-level parallelism during instruction scheduling. To exploit the detected parallelism, instructions are reordered to reduce the length of the code schedule and minimize the cycle count for execution. Code reordering is limited by the dependencies among instructions arising from both control flow and data flow. In this paper, we present the design of a VLIW architecture that uses value prediction to remove data dependencies and improve the instruction schedule. Our architecture consists of two execution engines, one for executing the original VLIW code, and the other for executing compensation code after a misprediction. Any code executed due to mispredictions is executed in parallel with the VLIW instructions. The instruction set and hardware of a traditional VLIW machine are modified accordingly to support this type of concurrent execution. The efficacy of the proposed architecture is demonstrated by implementing the prediction model in the Trimaran compiler infrastructure and studying the speedups that result due to the parallel execution of compensation code.

1 Introduction

The effectiveness of VLIW architectures depends on the capability of the compiler to reorder instructions to extract instruction-level parallelism. The operations in the code that can execute in parallel are packed together by the compiler into long instructions with the goal of minimizing the length of the instruction schedule. Code reordering, which is necessary to achieve this goal, is limited by both control and data dependencies. These dependencies can prevent the compiler from filling all the operation slots in a long instruction. One approach used in VLIW scheduling to reduce the instruction schedule length is to perform **control speculation** [3, 14] by moving code above conditional branches. Although control speculation is effective in removing control dependencies, true data dependencies are frequent, and long dependency chains become a bottleneck to the scheduler. This problem is severe especially in VLIW machines where conservatively computed data dependencies, especially for memory accesses, sequentialize the order of the operations for execution.

Value prediction has been shown to improve the performance of superscalar processors [5, 9, 10, 12, 13] by caching values and prediction confidences at run-time. However, VLIW machine code is scheduled statically and hence value prediction can only be applied to operations selected at compile-time. In [4], it was shown that executing operations with predicted values, **value speculation**, results in significant speedups for VLIW machines. A predicted value is eventually verified by executing the original operation that was predicted, and comparing the correct value with the predicted value. In case the prediction is found to be correct, code is executed as before. However if a misprediction occurs, all operations that were value-speculated using the incorrect predicted value are re-executed with the correct value. This re-executed code forms the **compensation code** for value speculation.

In [4], the compensation code for every prediction is scheduled in a separate block. Upon a misprediction, the control engine branches to the appropriate compensation code block. After the compensation code is executed, control is transferred back to the original code. While the compensation code is executed, the original code is not executing. Due to the lack of any parallel execution between these two code forms, the original VLIW code would wait for any prior compensation code to finish execution whenever a misprediction occurs. The VLIW machine may experience a

*Partially supported by National Science Foundation Grant CCR-98 08590 and a grant from Hewlett Packard to the University of Pittsburgh

slow down in performance in spite of the benefits from prediction. Another factor contributing to the slow down would be the frequent transfer of control due to additional branches in the code. The branches incorporated to perform recovery on a misprediction cannot be eliminated since the compensation code is executed only after verification of the prediction. Whenever control is transferred to compensation code blocks, the instruction cache would be affected by these blocks. In order to accommodate the compensation code blocks, the cache may evict other useful blocks depending on cache replacement policies.

Hence it is expected that value prediction would suffer some performance loss in VLIW machines when it is performed aggressively and consequently when compensation code is executed frequently. As we show later through experiments, the loss would be even more significant for wider machines. Although the effects of branch penalties and cache misses are ignored in [4], these issues may impact the benefits of value prediction sufficiently.

In this paper, we propose an architecture that reduces the impact of compensation code on performance as follows.

- Compensation code is generated dynamically during execution of the VLIW code with value-speculated operations. Whenever value-speculation is performed at run-time, the compensation code is produced and stored in a separate buffer, **Compensation Code Buffer (CCB)**. Hence the compensation code need not be scheduled statically with the rest of the VLIW code. This design avoids code growth and any impact on the instruction schedule. Also since the compensation code is not executed with the statically scheduled VLIW code, the instruction cache is not corrupted by compensation code blocks.

- A dedicated hardware mechanism is provided in our architecture for the execution of compensation code. A separate engine, the **Compensation Code Engine**, executes in parallel with the original **VLIW Engine**. The two engines are synchronized so that the VLIW Engine stalls upon a misprediction only if it requires a value that is yet to be computed by the Compensation Code Engine. Otherwise the VLIW Engine is allowed to continue execution while the Compensation Code Engine recovers from any misprediction by re-executing any mispredicted operations in parallel. In this way, the impact of the compensation code on the execution of scheduled VLIW code is minimized by performing synchronized execution of the two in parallel. We also avoid additional branches by having a separate hardware mechanism that does not function based on control transfer from the VLIW engine. Instead the execution is synchronized with the static VLIW code and, in this way, the instruction cache is not affected.

In Section 2, we introduce an overview of the proposed architecture. This overview is followed by extensions of the instruction set, the hardware modifications to the traditional VLIW machine and the design of the Compensation Code Engine. Section 3 outlines the experimental design considerations and presents performance evaluation results. Finally we conclude in Section 4.

2 Architecture

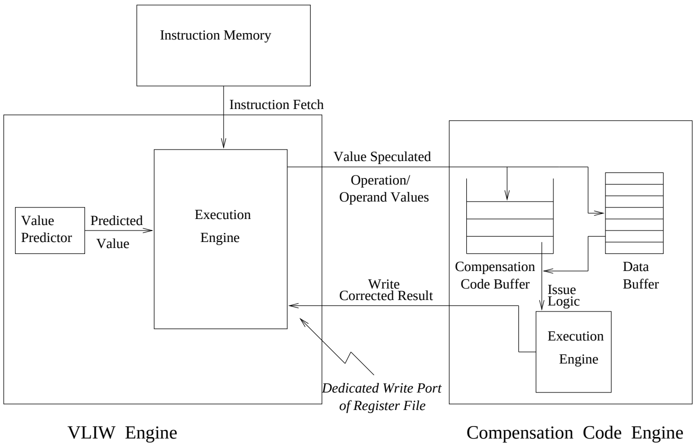

Figure 1. High-Level View of Proposed Architecture

Figure 1 shows a high-level view of the architecture consisting of two separate engines, executing in parallel. The VLIW Engine shown in the figure fetches instructions from memory, decodes them and then executes them on its execution engine. An operation within a VLIW instruction may have its destination operand predicted using some value prediction scheme [9, 12, 13, 15]. This operation simply accesses the value prediction table for the predicted value. The prediction enables other operations to use the predicted value. Such operations that are executed using predicted values are value-speculated. In the rest of the paper, we refer to such operations as simply being speculated. In Figure 1, when a speculated operation is encountered by the VLIW Engine, the operation is executed on the VLIW Engine. At the same time, the decoded operation is sent to the Compensation Code Engine for buffering and re-execution later if the speculatively computed value is found to be incorrect. A speculatively computed operation would be incorrect if the prediction is incorrect. A prediction is eventually verified by the VLIW Engine when the original operation, whose destination operand was predicted, is executed and the result is compared with the predicted value. If a misprediction is detected, the correct values of the speculated operations are computed by the Compensation Code Engine. The VLIW Engine may stall after detecting a misprediction if the correct values have not yet been produced by the Compensation Code Engine. This situation occurs when an instruction that is ready for execution consists of a non-speculative operation that depends on a value which was previously predicted and the prediction has not been verified yet. When all the operands of such an operation are verified to be correct, the instruction is issued for execution.

The Compensation Code Engine buffers the speculated operations for execution in case of a misprediction. These operations have already been decoded by the VLIW Engine. These decoded operations are executed on a simple pipeline engine in the order they were fetched. Execution of an operation on the Compensation Code Engine depends on whether or not the predictions made with the operands are verified as being correct by the VLIW Engine. Hence, an operation is stalled on the Compensation Code Engine pipeline until outcomes of all predictions related to its operands are verified. Information of a prediction outcome is sent by the VLIW Engine when it executes the original predicted operation and compares the result with the predicted value. If prediction of any operand was incorrect, the associated operation is executed on the Compensation Code Engine pipeline and results sent to the VLIW Engine. If all operands were predicted correctly, the operation is flushed out of the Compensation Code Engine pipeline since the VLIW Engine had already executed this operation with correct operands.

2.1 Instruction Set Extension

In order to achieve the above functionality, the instruction set of the VLIW architecture is extended. These extensions include an operation to perform prediction of values and additional forms of operations to verify the predicted values. The operation *LdPred* [4] loads the predicted value, computed by the value predictor, into a register. Verification of a predicted value involves re-execution of the original operation, whose value was predicted using *LdPred*, followed by a comparison of the correct value with the predicted value. The **check prediction** form of an operation is introduced to incorporate such verification.

Additionally, the VLIW engine should be able to distinguish speculative operations from the non-speculative ones. The **speculative** form of an operation utilizes a predicted value either directly or indirectly through some data dependency. A speculative operation is executed similar to a generic instruction except that predicted values are obtained either directly or indirectly through a previous *LdPred* operation. The **non-speculative** form of an operation utilizes only correct (verified) values. Execution of a non-speculative operation is stalled until all operands are verified as correct.

To understand the significance of these operation forms, let us consider an example of VLIW code scheduled on the modified machine. Figure 2 shows a dependence graph for a sequence of operations. Assume the add, move and multiply operations are of unit latency and the loads are of latency 3. The figure also shows a schedule that would result if a conventional list scheduler was used to schedule the code. This figure shows the schedule of the VLIW code without any value speculation. Operations 4 and 7, being loads and of high latency, are useful for prediction since they would allow several operations dependent on them (5,6,8,9,10,11) to be speculated.

Figure 3 shows the modified dependency graph and VLIW schedules that would result on the proposed architecture if loads were predicted. Within the schedules, the speculated operations are lightly shaded while the non-speculative operations, which depend on predicted values, are highlighted using a darker shading. Besides the VLIW schedule the figure also displays the code executed on the Compensation Code Engine pipeline. Let us consider each schedule one by one. For each case, we assume that the scheduler chooses all operations dependent on the loads to be speculated except for operations 10 and 11. Figure 3(b) shows the schedule for the case when both load operations were predicted correctly. In this case, no compensation code is executed. Instructions 10 and 11 are stalled until the values of the predicted operations they depend upon are verified. The schedule has an improvement over the original schedule, with no value speculation, by 3 cycles.

Figure 3(c) shows the schedule for the case when the value of $r7$ is predicted incorrectly. The misprediction is detected in cycle 6 when the *check prediction* operation for loading register $r7$ completes execution. The prediction for register $r7$ had resulted in speculation of other operations. These need to be recomputed by the Compensation Code Engine. Execution within the

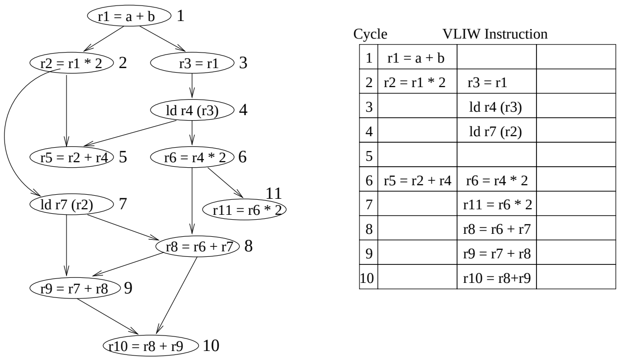

Figure 2. Example (a) Dependency graph (b) Schedule

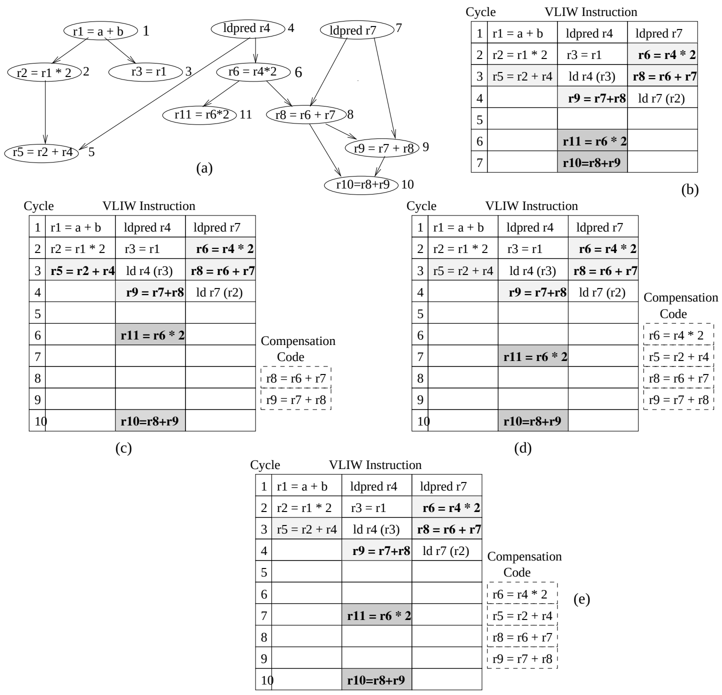

Figure 3. Modified example after prediction (a) Dependency graph (b) Schedule for both predictions correct (c) Schedule for operation 7 mispredicted (d) Schedule for operation 4 mispredicted (e) Schedule for both predictions incorrect

Compensation Code Engine does not begin until cycle 8 since the operations that were correctly speculated need to be flushed. Execution on the VLIW instruction is stalled until the Compensation Code Engine computes the values of register $r8$ and $r9$ (required by the non-speculative operation in cycle 10). Note that although the schedule of the VLIW code is statically computed, the stalling and issuing of VLIW instructions depends on run-time prediction and verification of values.

In Figure 3(d) the value of register $r4$ is mispredicted. The VLIW Engine is informed of this misprediction at the end of cycle 5, when the *check prediction* operation form of loading register r4 completes execution. The compensation code in this case is larger, since the number of value-speculated operations that are data dependent on $r4$ is greater than the ones dependent on $r7$. However the schedule length is same as that in the last case. This is because the compensation code starts executing two cycles earlier (cycle 6) and the subsequent VLIW instruction need not stall until the Compensation Code Engine completes execution of all operations. Hence, the instruction computing $r11$ can execute as soon as the first operation within the Compensation Code Engine gets executed. Figure 3(e) shows the code in case both the values were mispredicted. Here the code executed on both the engines is identical as in the previous case. The reason is since the compensation code is the same whether the load operation 4 was mispredicted or both the load operations were mispredicted. From the example, we observe that whenever prediction is performed, the total cycle count is never worse than the case when code is executed without predicting values (Figure 2). This result is true for the example even when all the predictions turn out to be incorrect. The reason for this occurrence is that compensation code gets executed in parallel with the VLIW engine.

Execution of a VLIW instruction consisting of non-speculative operations is stalled until all predicted operand values are verified. In order to stall such instructions, a mechanism is needed to specify which operand values are being predicted in every cycle. This mechanism is modeled as a register storing a bit for each value that is predicted. We call this register the **Synchronization register** since it needs to synchronize the definition and usage of predicted values. In order to model predictability, every predicted value is assigned a bit within the Synchronization register. A predicted value is computed either by an *LdPred* operation or an operation that is value-speculated. An operation computing the predicted value sets a bit within the Synchronization register to indicate the associated value is predicted. The register index associated with each predicted value is pre-determined statically by the compiler. The bit value is eventually cleared when the prediction is verified and the correct value is computed, if necessary.

In order to execute the non-speculative operations with correct operands, each VLIW instruction is associated with a set of bit indices of the Synchronization register. These bit indices correspond to the operand values of the non-speculative operations of the VLIW instruction. The instruction is not executed until the associated bits of the Synchronization register indicate these values are correct. Any predicted value is eventually verified (and corrected if necessary), and the associated Synchronization register bit is cleared. In case of a correct prediction, clearing of the bit is done by the VLIW engine after executing the *check prediction* form of the operation, whose computed value was predicted. For an incorrectly computed value, the Compensation Code Engine clears the bit for the value after computing its correct value.

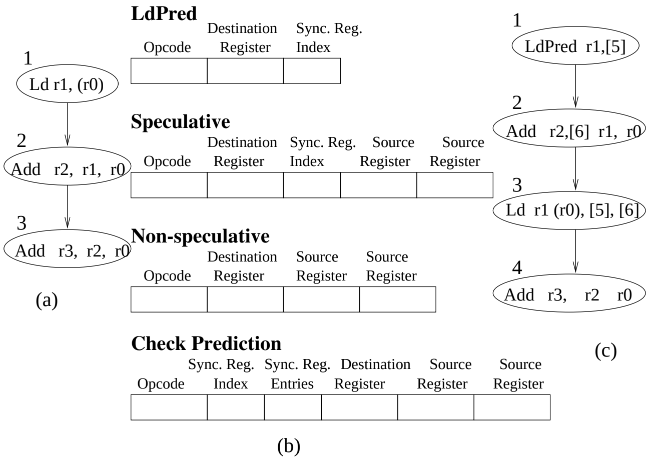

Figure 4. Operation format (a) Original code (b) Format of modified VLIW operations (c) Code with modified operations

The *LdPred* operation, besides loading the predicted value into a register, also stores a bit index of the Synchronization register. The bit referred to by the index is set by the *LdPred* operation to indicate that the value of the operation is predicted and not verified yet. The operation fields for *LdPred* are shown in Figure 4(b). The format of a *speculative* operation is shown in Figure 4(b). There is an additional field that stores an encoded number that holds a bit index of the Synchronization register. The bit of this index within the register is set when the operation finishes execution. The set value of the bit indicates the value computed by the associated operation is predicted and not verified yet. In Figure 2(c), operation 1 performs prediction on the value stored in register $r1$. This operation accesses the value prediction table and sets bit 5 of the Synchronization register. Operation 2 is speculative and uses the predicted value of operation 1, setting bit 6 of the Synchronization register.

The *check prediction* operation, upon computing the correct value, may clear bits within the Synchronization register that are related to the predicted value. These bits include one for the *LdPred* predicted value, which is always cleared after checking prediction regardless of the comparison result, since the correct value is computed along with the comparison. Besides this bit, the *check prediction* operation may clear bits that correspond to the operations speculated using the *LdPred* predicted value. These bits are cleared if the comparison is found to be successful. In case the comparison is not successful, the Compensation Code Engine would compute the correct values for these speculated operations and clear their bits. The instruction format stores the entry index for the *LdPred* predicted value as well as an encoded number for the bit indices for the rest of the predicted values whose bits are cleared conditionally. These bits are stored separately from the *LdPred* predicted value index since a *check prediction* operation needs to access the *LdPred* predicted value separately for comparison. In the example of Figure 4(c), operation 3 performs the load whose value was predicted in operation 1. After computing the correct value of $r1$, bit 5 of the Synchronization register is cleared and the correct value is compared with the predicted value. If the prediction is found correct, bit 6 is also cleared within the Synchronization register to indicate the value speculation was correct. Otherwise, the register file of the VLIW Engine is updated with the correct value of $r1$ and the Compensation Code Engine eventually clears bit 6 after computing the correct value of $r2$.

The *non-speculative* form requires correct operand values for execution. Thus operand values that were predicted prior to this operation must be verified. The instruction format of a non-speculative operation is the same as that of a conventional operation as shown in Figure 4(b). Any VLIW instruction containing one or more non-speculative operations stores bit indices of the Synchronization register corresponding to the operand values of the non-speculative operation that are predicted. For each non-speculative operation, two bit indices are stored related to the the most recent operations within the dependence graph (in topological order) that compute the operands of the operation. These bit indices are encoded together as a number and stored with the VLIW instruction. Before issuing the VLIW instruction, all the bits of the Synchronization register indicated by this number should be checked that none of these bits are set. In other words, the operand values of all the non-speculative operations are correct. Otherwise the instruction stalls. It is sufficient to check the most recent operation value that each source operand depends upon, for correct semantics, since any previous predicted values would have

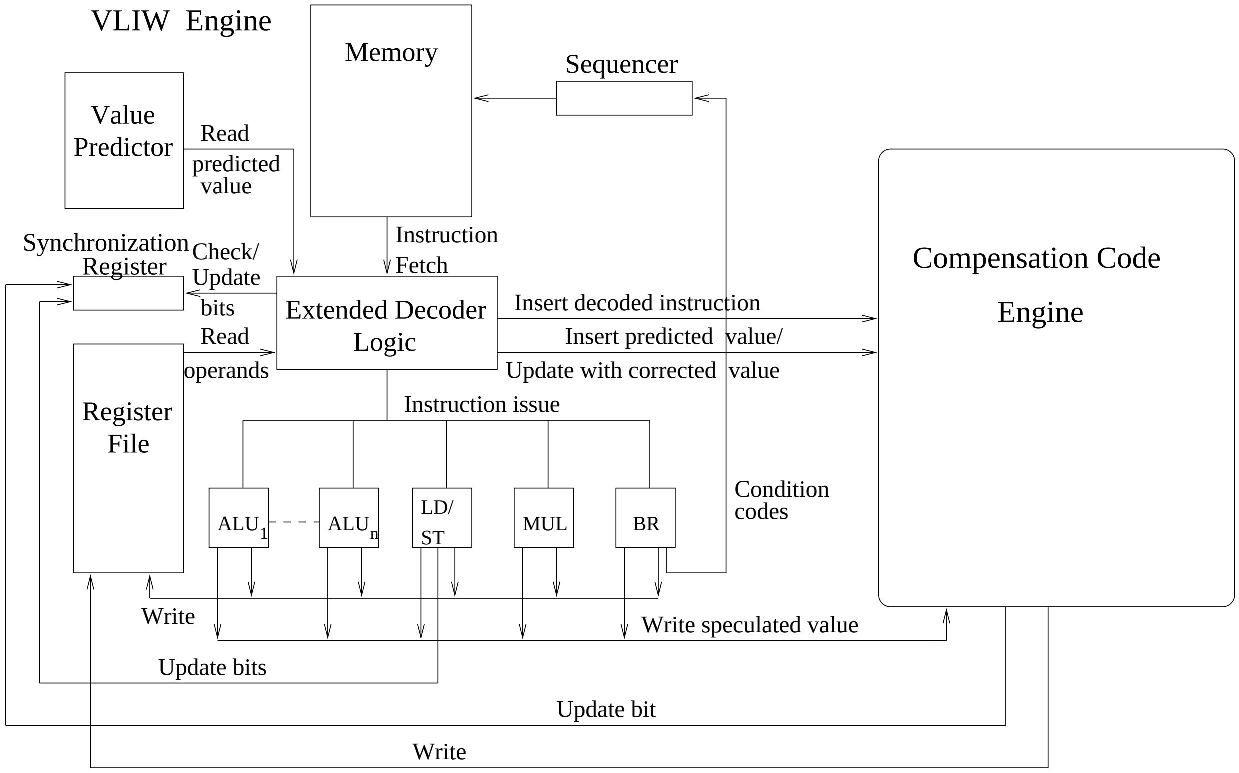

Figure 5. VLIW Engine Architecture

been correctly computed as long as the most recent one is verified. In the example of Figure 4(c), operation 4 is non-speculative and stalls on operand r2 until the value of r2 is correctly computed. Bit 6 of the Synchronization register which corresponds to value of r2 is not shown in the figure since it is encoded as part of the entire VLIW instruction rather than with an individual operation. This bit is cleared either by operation 3 if the prediction was correct or by the Compensation Code Engine after re-executing operation 2 with the correct value of r1. Once bit 6 becomes cleared, operation 4 is allowed to execute.

2.2 Primary VLIW Engine

The VLIW module of the architecture is similar to a conventional VLIW machine and is shown in Figure 5. The additional hardware includes the value predictor that returns a value whenever a VLIW operation uses value prediction (different value predictors that can be used are not discussed in this paper). Also, the register file incorporates the Synchronization register, shown separately in the figure.

Upon fetching a VLIW instruction from memory, the extended decoder logic decodes and executes the individual operations within the VLIW instruction. In case any operation is a *LdPred* operation, the predicted value is fetched from the value predictor. The *LdPred* operation is not issued to any functional unit since it uses a predicted value. The predicted value is sent to the Compensation Code Engine to be stored for any operation to use later. This *LdPred*'s predicted value is later verified by the corresponding *check prediction* operation and updated if the prediction was incorrect.

If there are operations besides *LdPred* in the current VLIW instruction, the decoder checks the Synchronization register field associated with the VLIW instruction before performing an issue. If this field is non-empty, there are some non-speculative operations within the VLIW instruction and these operations can be only be issued if their operand values are verified and correct. This field of the instruction is decoded and the non-zero bits of the decoded value are read as indices of the Synchronization register. The VLIW instruction is stalled until these indices of the Synchronization register have their bits cleared. Clearing these bits signifies that the operand values of all non-speculated operations within the current VLIW instruction have been verified. These bits are cleared either by a *check prediction* operation or by the Compensation Code Engine. Note that these encoded values are computed by the compiler statically for each VLIW instruction and

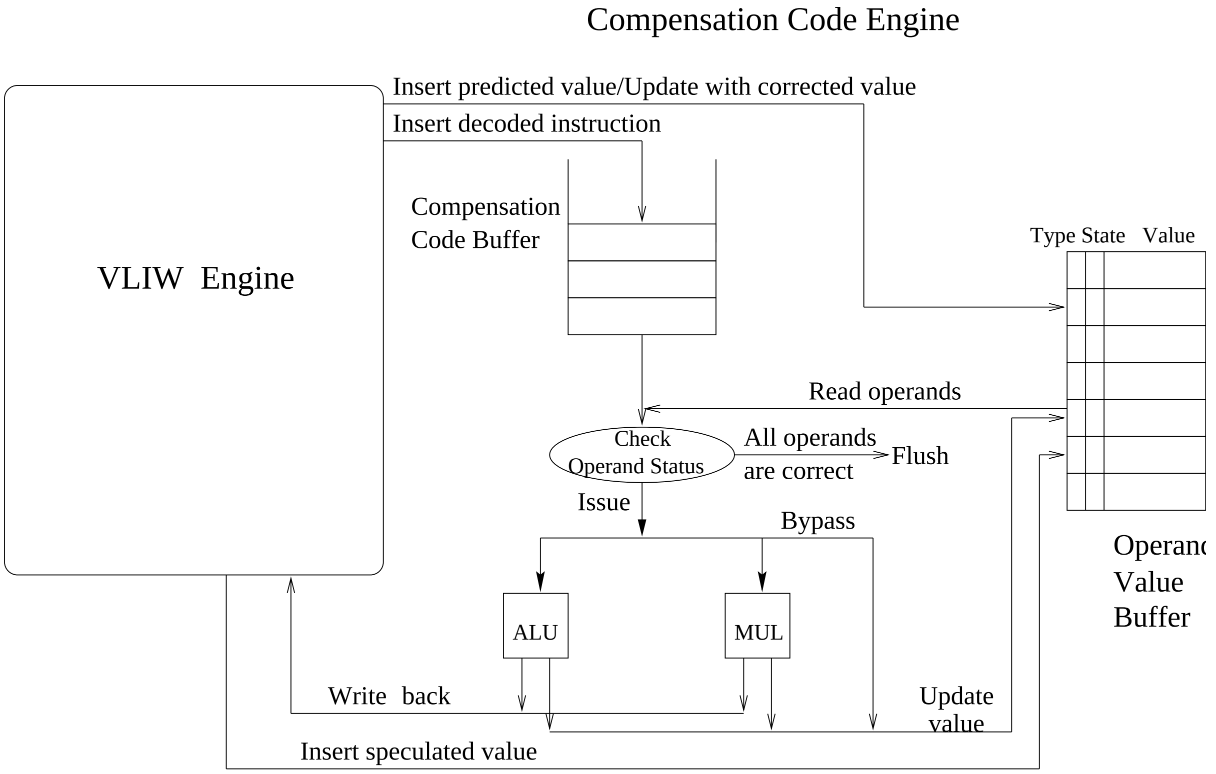

Figure 6. Compensation Code Engine Architecture

these values do not change dynamically.

If the decoder encounters an operation that has a *check prediction* form, the operation is issued to a functional unit. Upon completion of execution, the computed value is compared with the *LdPred* predicted value stored in the register file. If the comparison is successful, the Synchronization register bits specifying values computed by speculated operations, dependent on the *LdPred* predicted value, are cleared. If the comparison is unsuccessful, the register file is updated with the correct value. The bit of the Synchronization register corresponding to the *LdPred* predicted value is cleared in either case. Also the comparison result and correct value are sent to the Compensation Code Engine to update the predicted value stored there.

In case the decoder encounters a speculated operation, it sets a bit of the Synchronization register. The set bit value signifies that the value associated with that bit is predicted. Upon computing a predicted value, the VLIW Engine sends the result to the Compensation Code Engine. This allows the Compensation Code Engine to read predicted values locally without accessing the register file of the VLIW Engine. If the predicted value of a speculated operation is incorrect, the Compensation Code Engine computes the correct value and writes to the register file through a dedicated write port. Also the Synchronization register is updated after the value is recomputed within the Compensation Code Engine.

2.3 Compensation Code Engine

The Compensation Code module incorporates a simple engine that performs in-order execution of speculated operations. The order of execution of the operations is the same as the order in which they are received from the VLIW Engine. The components are shown in Figure 6. The engine consists of a Compensation Code Buffer (CCB) which is a First-In-First-Out buffer. The speculated operations are inserted in this buffer as they are executed on the VLIW Engine. Operations that are inserted in the CCB may not always get executed by the Compensation Code Engine. They are only executed in case they were incorrectly speculated. If the operand values of a speculated operation were correctly used within the VLIW engine, the Compensation Code Engine does not need to re-execute the operation. Instead, the operation is flushed as shown in the figure. Hence, the values of the operands are always sent along with the operation but these values may or may not be used by the Compensation Code Engine.

Each operand is assigned a state depending on how

Operand Type	*Initial State*	*Final State*	*Condition*
Predicted using *LdPred*	Prediction not verified	Recompute operand	Operand value is mispredicted
		Operand correct	Operand value is correctly predicted
Predicted by being speculatively computed	Recomputation not verified	Recompute operand	Associated prediction is incorrect
		Operand correct	Associated prediction is correct
Correct	Operand correct	Operand correct	Always

Table 1. State Table of Operand within Compensation Code Engine

it is computed and whether its value is correct or not. An operation is issued for execution based on the states of its operands. Table 1 shows an operand can be either predicted using *LdPred*, predicted by being speculatively computed, or correct. If the operand is predicted using *LdPred*, it is assigned an initial state of not being verified. Once the associated *check prediction* operation is performed, the operand's state is marked as either correct or needing recomputation, depending on whether the prediction is correct or not. If an operand is predicted by being speculatively computed, the initial state denotes that it is not known yet if the operand's value needs to be recomputed within the Compensation Code Engine. When the associated *LdPred* prediction is verified, the state is updated to either correct or marked for recomputation. The last case is when the operand does not involve any prediction, directly or indirectly through any dependence and its state is marked as being correct.

The issue logic of the Compensation Code Engine avoids execution of any operations other than those required to recover from mispredictions. To execute only the required operations, each operation is assigned a state of either being stalled, issued for execution or flushed. These states are derived directly from the states of the operands. If any of the operands is in a state of not being verified, the operation waits for the correct values before being issued for execution. This implies the Compensation Code Engine stalls if an operand's value is not verified yet. An operation is issued for execution if none of its operands is in the state of not being verified and if one or more of its operands is in a state requiring recomputing. The latter condition happens if any of the operand's value was mispredicted leading to the operand's state being marked for recomputing. In such a case, the operation would need to be executed locally with the correct operands. If all operands are in the correct state, the operation is flushed since the correct values would already have been computed by the VLIW Engine.

In order to store the values needed by the Compensation Code Engine, a buffer is provided in our architecture called the **Operand Value Buffer (OVB)**. It stores the operand values as well as bits for the type and state associated with each operand value. An entry is inserted into the buffer when the VLIW Engine sends the decoded operation along with the operand values. An entry is updated on different conditions for the different operand types. If the operand is predicted, its entry gets updated by the VLIW Engine when the prediction is verified. For an *LdPred* predicted value, the update is for both the value and state. For a predicted operand that was speculatively computed, the state update is similar but the value is corrected upon a misprediction only after the operation computing the value is executed within the Compensation Code Engine. An operand value that is correct does not need to be updated.

The operations are initially fetched from the CCB and the operand values are read from the OVB. The operands are checked to see whether the operation needs to be executed. The operation behaves according to the states of the operands, as mentioned above. After execution of an operation on the compensation code pipeline, if the computed value is required by a subsequent operation, it is written to the OVB (besides being always written back to the register file of the VLIW Engine).

The entries of the OVB can be traced for the Figure 3 example. Figure 7 considers the case of Figure 3(c), when the value in register r4 is correctly predicted and the value in register $r7$ is mispredicted. The scheduled VLIW code along with the compensation code is shown to the left in the figure. The contents of the OVB and CCB are shown next to each other for each cycle. Each entry of the OVB is labeled as v_n where n is the register number storing that value. Hence $v8$ is the value computed and stored in register $r8$. Note that there may be more than one speculated value for a particular register but this is not the case in the example shown. To make the figure more readable, we do not show the correct entries (having no predic-

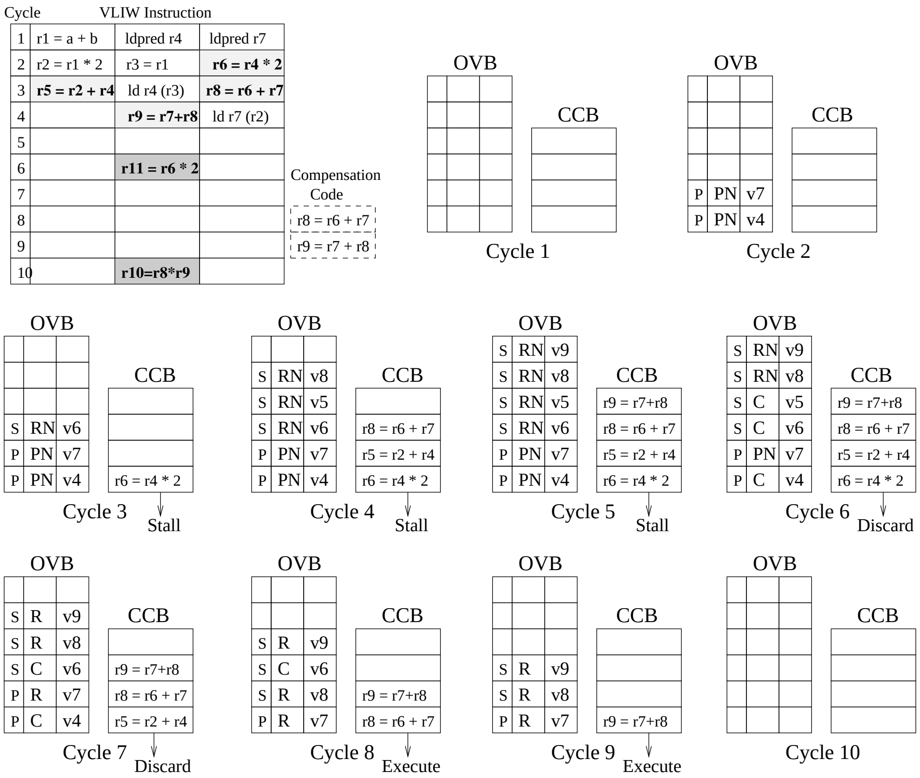

Figure 7. VLIW Schedule and Execution Trace

tion) in the OVB. The speculated operands have the state S while the predicted operands have the state P. Also the states from Table 2 are denoted as follows. State PN denotes the state of the operand's prediction to be not verified while state RN is the state of the operand's recomputation not verified. State C is the state of the operand's value being correct while R is the state requiring recomputation.

In the figure, the contents of the CCB and OVB are shown during each cycle. During the first cycle, there are no entries within these buffers. When the second cycle initiates, the predicted values of register $r4$ and $r7$ are sent to the Compensation Code Engine and stored in the OVB. These values are assigned the status of not being in the verified state (PN). After the second cycle, the value in register $r6$ is speculatively computed in the VLIW Engine and stored in the Value buffer (shown in cycle 3). A state value of RN relates to the default initial state for the speculated value. Similarly the speculated values $v5$, $v8$ and $v9$ are added to the OVB in the next two cycles.

Cycle 5 verifies the predicted value of register $r4$ and this updates the state of its value in the OVB in the next cycle. Since this value is correctly predicted, the dependent operations (computing $r6$ and $r5$) are discarded in cycles 6 and 7 respectively. The value of $r5$ is also removed from the OVB since it is not required by any subsequent operation. The value in $r7$ is found mispredicted in cycle 6 and its state is updated in the next cycle. The misprediction results in the execution of compensation code in the subsequent cycles. This results in the computation of the correct values $v8$ and $v9$ in cycles 8 and 9 as shown. These values are sent to the VLIW Engine to allow any stalled non-speculative operations to execute. In this example, the operation computing r10 is non-speculative and eventually executes in cycle 10 after receiving the correct values from the Compensation Code Engine.

3 Experimental Results

To evaluate the performance of the proposed machine, a simulation was implemented, and a number of experiments were performed. The run-time behavior of the two execution engines was simulated using an existing VLIW compiler operating on wide-issue machine descriptions. The simulation involved computation of the static schedule lengths of the basic blocks of the benchmarks. These schedule lengths were computed assuming the machine had the capability of performing value prediction. In order to perform accurate measurements, the machine description of the VLIW

Benchmark	*Best case*	*Worst case*
SPEC INT 95 programs		
compress	0.48	0.02
ijpeg	0.35	0.04
li	0.49	0.05
m88ksim	0.53	0.02
vortex	0.49	0.006
SPEC FP 95 programs		
hydro2d	0.63	0.002
swim	0.49	0.003
tomcatv	0.51	0.002

Table 2. Fraction of execution time used by blocks with speculated operations

Benchmark	*Best case*	*Worst case*
SPEC INT 95 programs		
compress	0.80	0.94
ijpeg	0.82	0.93
li	0.85	0.91
m88ksim	0.73	0.89
vortex	0.68	0.92
SPEC FP 95 programs		
hydro2d	0.80	0.93
swim	0.98	1.02
tomcatv	0.95	1.002

Table 3. Schedule lengths of blocks with speculated operations as a fraction of schedule length with no prediction

architecture was modified to incorporate the instruction semantics as proposed above. Since the *check prediction* operation form always loads from memory, this operation can be made to execute on a memory unit with the extra semantics of performing a comparison check. Also the *LdPred* operation, being similar to a move operation, can utilize an integer functional unit with the source operand coming from the Value Predictor instead of a register. Using these assumptions, we can avoid additional functional units for performing the simulations.

The simulations were performed on the Trimaran compiler [1] running on HP PA-RISC processors. Code was scheduled by predicting loads on the longest critical path for each block and speculating operations dependent on the loads. In our experiments, the basic blocks were optimized to the highest level of control. These blocks were initially value profiled [2, 6], based on stride [5] and FCM prediction [13]. The final value prediction rate for each operation, executed in the simulation runs, was chosen to be the higher value out of these two prediction rates. Besides value profiles, the generated code was also profiled to determine the frequency of execution of each block. The additional instructional semantics described above, were incorporated into the Playdoh architecture [8].

The profile parameters were used in the experiments to estimate the execution cycles needed to execute blocks with speculation. The performance of the proposed architecture was analyzed by determining the execution cycles that would benefit from the improvements obtained by prediction. Table 2 shows the fraction of the execution time used by blocks whose operations were speculated. The best case column indicates the fraction of overall execution time spent in executing blocks in which values were predicted and then found to have been all correctly predicted and the worst case column gives the fraction of total execution time spent in blocks in which all predictions made were found to be incorrect. As an example, for the benchmark *compress* benchmark, 48% of the total execution time was spent executing blocks where predictions were made and all the predictions were correct. We observe that on an average the benchmarks spent half of the overall time in blocks where all predictions were made correctly. It should be noted that in all these executions, the threshold of load prediction (from value profile) was kept at a fairly low percentage of 65% to analyze the mispredictions cases as well. The cases in which all the predictions within a block were incorrect are very infrequent and account for a very small fraction of the overall execution time, as can be seen from the third column of the table. Hence the effect of the compensation code on the overall execution time is quite small for our technique.

In order to analyze the performance improvements due to the execution of predicted code, we analyzed the improvements on the schedule lengths of the blocks. Table 3 shows the effective schedule lengths of the blocks incorporating prediction (calculated after incorporating compensation code), as a fraction of the schedule lengths of the original blocks (with no predictions). In the best case with all correct predictions, the schedule length reduces by about 20% on an average. In case when all the predictions within a block turn out to be incorrect, the schedule still manages to improve for most of the cases. This improvement is due to the the additional execution power of the compensation code engine.

The improvement in schedule length was analyzed by studying the distribution of changes in scheduled

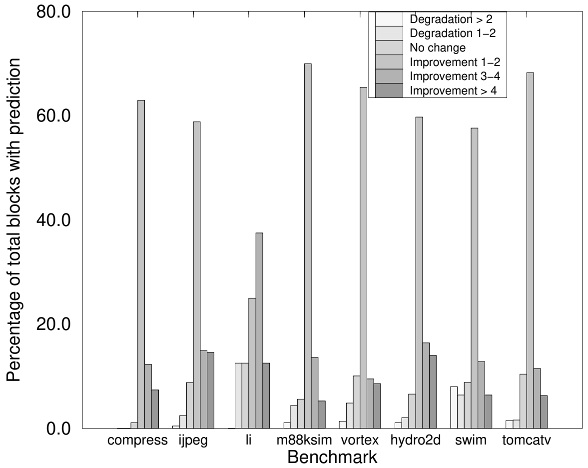

Figure 8. Distribution of change in schedule lengths due to prediction

lengths caused by value prediction (see Figure 8). This figure shows the percentage of the total blocks executed that experience varying degrees of changes in schedule lengths. The distribution is for the case when all predictions are correct and is expressed as percentages for different cases of cycle improvement or degradation. We observe that a large percentage of the blocks improve the schedule length by 1-4 cycles. This improvement is fairly significant for basic blocks. For larger regions such as hyperblocks [11] and superblocks [7], we expect to see a further improvement for the majority of blocks. Note that some blocks experience a degradation in length. This degradation is due to the additional *LdPred* operations that may result in additional cycles. However, the average schedule lengths for most blocks is reduced by value prediction.

The machine description used to observe the above results was a 4-wide VLIW architecture. We also compared the performance of a 8-wide machine having two of each type of functional unit as shown in Table 4. From the figure, we observe that for a higher issue width, the processor spends less time in executing the correctly predicted code. This result occurs because in order to create greater degrees of parallelism, an increased amount of speculation is performed and hence the likelihood of mispredictions and execution of compensation code increases. Moreover as the last two columns in the figure show, the improvement in block schedule length is higher for the wider machine. Hence, the impact of executing compensation code on overall performance is higher for a wider machine. This observation reinforces our earlier assertion that the effect of compensation code is more significant for wider machines.

To compare our technique with an existing approach for handling compensation code, we implemented a recovery scheme, based on the one proposed in [4]. For each predicted operation, a compensation code block was statically inserted in the program graph. This additional block consisted of copies of the instructions from the original block that were speculated based on the prediction. The compensation code blocks were scheduled along with the rest of the code, and execution cycles were estimated using profile data. In general, programs were observed to spend a significant portion of their total execution time recovering from mispredictions. For the worst case when all predictions within a block were incorrect, the average time spent in executing compensation code blocks ranged from 3-10% of the total execution time, compared to our scheme where this percentage was negligible. The increase was because the effective schedule length for a block (calculated after incorporating its compensation code blocks) was significantly higher than the block lengths observed with our technique. This marked difference indicates the usefulness of our technique in handling compensation code for aggressive prediction mechanisms.

4 Conclusions

In this paper we introduced the design of an architecture capable of supporting aggressive value prediction. The architecture consists of the conventional VLIW engine, and an additional engine to execute compensation code whenever a misprediction occurs. Compensation code is generated dynamically avoiding any overhead of the static code size and cache misses. The additional engine is a simplified in-order pipeline engine and executes compensation code in parallel with the VLIW code. The hardware design and instruction set architecture extensions required for this architecture are given in the paper. Experiments were performed to study the effectiveness of the architecture by simulating the run-time behavior of profiled code on the modified machine description. From the experiments, we conclude that compensation code is a significant factor when value prediction is performed on VLIW machines. The proposed architecture was observed to dramatically limit the impact of compensation code, especially in cases of severe mispredictions, benefiting the overall performance of value prediction. The proposed solution was shown to be more significant for wider issue machines, in which case the performance improvement is even higher. Also a comparison with an existing approach of handling recovery shows a marked

Benchmark	*Ratio for issue width=4*		*Ratio for issue width=8*	
	Ex. time fraction	*Schedule length fraction*	*Ex. time fraction*	*Schedule length fraction*
SPEC INT 95 programs				
compress	0.48	0.80	0.40	0.67
ijpeg	0.35	0.82	0.34	0.74
li	0.49	0.85	0.42	0.73
m88ksim	0.53	0.73	0.49	0.68
vortex	0.49	0.68	0.42	0.57
SPEC FP 95 programs				
hydro2d	0.63	0.80	0.57	0.71
swim	0.49	0.98	0.46	0.90
tomcatv	0.51	0.95	0.47	0.83

Table 4. Entries of previous two tables of best case for different issue widths

improvement with our technique.

References

[1] The trimaran compiler research infrastructure. *Tutorial Notes*, November 1997.

[2] B. Calder, P. Feller, and A. Eustace. Value profiling. *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[3] J. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Tranactions on Computers*, July 1981.

[4] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte. Value speculation scheduling for high performance processors. *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[5] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. *EE Department TR#1080, Technion - Israel Institute of Technology*, November 1996.

[6] F. Gabbay and A. Mendelson. Can program profiling support value prediction? *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[7] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haag, J. Holm, and D. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, May 1993.

[8] V. Kathail, M. Schlansker, and B. R. Rau. Hpl playdoh architecture specification: Version 1.0. *Hewlett-Packard Laboratories Technical Report HPL-93-80, Computer Systems Laboratory*, February 1994.

[9] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–237, December 1996.

[10] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[11] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[12] T. Nakra, R. Gupta, and M. L. Soffa. Global context-based value prediction. *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, January 1999.

[13] Y. Sazeides and J. E. Smith. The predictabilty of data values. *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–258, December 1997.

[14] M. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R.Johnson, S.Anik, and S. G. Abraham. Achieving high levels of instruction-level parallelism with reduced hardware complexity. *Hewlett-Packard Laboratories Technical Report HPL-960120, Hewlett-Packard Laboratories*, February 1997.

[15] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 281–290, December 1997.