Dynamically Allocated Dependence Vector for Data Value Speculation Recovery

Kevin Volkel, SakthiKumaran Sambasivam

I. Project Description

This project is a selective re-execution approach for data value speculation recovery. Our approach is to use dynamically allocated dependence vectors in order to keep track of which instructions are directly and indirectly dependent on one another. With this indirect and direct dependence information, we replay instructions from their Issue Queue entry whenever it is detected that the instruction may have executed with an incorrect source operand. We also study the squashing from head recovery technique, which is defined as squashing and flushing the pipeline whenever a mispredicted instruction reaches the head of the Active List. We also identify the strengths and weaknesses of both squashing recovery and selective recovery, and combine the two approaches to provide a hybrid recovery strategy. In order to evaluate our recovery strategies, we also implement a differential finite context method (DFCM) value predictor paired with a confidence mechanism that is a resetting counter that is incremented probabilistically. This report will provide a description of the microarchitecture of our design, and will provide an analysis of some performance results obtained for multiple data value speculation recovery strategies.

II. Simulator Accomplishments

TABLE I: Table of simulator accomplishments for different recovery strategies, **committed instruction numbers are the same for perfect and real branch prediction**.

Major Microarchitecture Feature/Simulator Configuration	**(# of committed instr.)**				
	astar	bzip	mcf	perl	hmmer
real value pred w/ perfect recovery	100M	100M	10M	100M	100M
real value pred w/ squash from head recovery	100M	100M	10M	100M	100M
real value pred w/ selective re-execution (small GDV)	100M	100M	10M	100M	100M
real value pred w/ selective re-execution (large GDV)	100M	100M	10M	100M	100M
real value pred w/ hybrid recovery	100M	100M	10M	100M	100M

III. Microarchitecture Description

This section provides a detailed description of the microarchitecture of our project. First we will present the architecture of the value predictor that we implement, and then we will present the microarchitecture of our selective re-execution recovery approach that uses dynamic dependence vectors to keep track of indirect and direct dependence between instructions.

A. Value Predictor Architecture

The DFCM value predictor (Figure 1) consists of three different tables, a First Level Table, a Second Level Table, and a Last Value Table [1]. The First Level Table is indexed by the PC of an instruction, or a combination of the PC of the instruction and the global branch history. This table stores a confidence counter and a hash of past *strides* between values produced by an instruction. This is the major difference between DFCM and FCM predictors as FCM predictors store a hash of past *values* that the instruction has had rather than *strides*. The hash of the stride history is used to index the Second Level Table which produces a predicted stride. Because DFCM is a stride predictor, there also needs to be a table that holds the last value that the instruction produced so that a predicted value can be calculated using the predicted stride and the last value.

The value predictor tables in our implementation are updated at commit so that wrong path/mispeculated instructions do not update the tables. Although this may help predictor accuracy as non-committed instructions will not thrash the predictor tables, implementing this from a design perspective is difficult due to the possibility of multiple instances of the same static instruction being in flight. We do not consider the complexity of this part of the predictor, as we update the predictor tables in an oracle manner. That is, we update the predictor tables immediately for an applicable instruction if that instruction will ultimately retire later.

The confidence mechanism that we pair with our predictor is a resetting counter that is probabilistically incremented whenever there is a correct prediction, and cleared upon an incorrect prediction [2], [3]. We also use this probabilistic updating idea in order to easily implement a method that determines if a value prediction is moderately confident or very confident. We do this by having two separate counters, and each counter is updated based on separate probabilities. The counter that conveys higher confidence will be incremented at a lower probability than the other. This method comes in handy for our hybrid recovery approach.

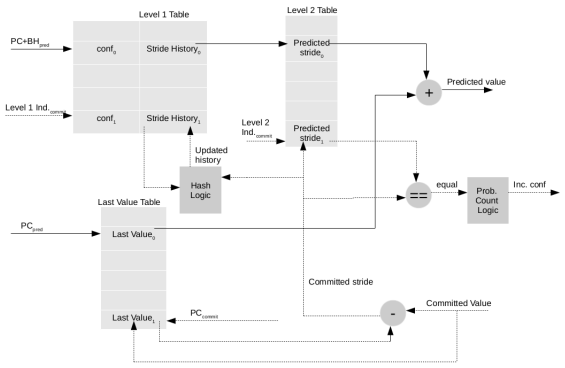

Fig. 1: Value predictor architecture implemented. Dashed lines represent a committed instruction accessing the predictor, solid lines indicate an instruction making a prediction. Note, all reads of the predictor's tables in the commit process happen before writes to the same locations.

.

B. Misprediction Recovery Architecture

In the Rename stage (Figure 2), based on the confidence of the prediction, we will choose either squashing from head or selective re-execution recovery mechanism. If the confidence is between T_L and T_H, we will choose selective re-execution. But if the confidence is higher than T_H , we opt for squashing from head. We have a structure called the Global Dependency Vector (GDV) which indicates if there is a dependence vector slot available to be used for selective re-execution. If there is an open slot, and this instruction has a prediction that will be recovered selectively, we create a one hot bit vector, and then OR this with DVs of the source operands read out from the Dependency Matrix (DM). The result of this operation is the resultant DV of the instruction and it is used to update the DM entry that is pointed to by the instruction's physical destination register. This is needed to propagate dependency information. In order to track whether an instruction has a valid VP_ID, we use a flag called VP_ID_VAL. If there is no open bit in the GDV, we assign VP_ID_VAL as false and the value prediction is dropped. For branch misprediction handling, we checkpoint the GDV like the GBM at the point of a branch. In the dispatch stage, if the instruction has a valid value prediction, the predicted value will be injected into the pipeline by writing the predicted value into the instruction's physical destination register.

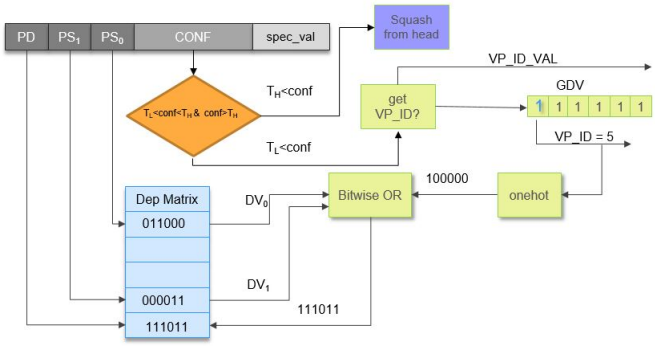

Fig. 2: Selective Re-execution-Rename Stage

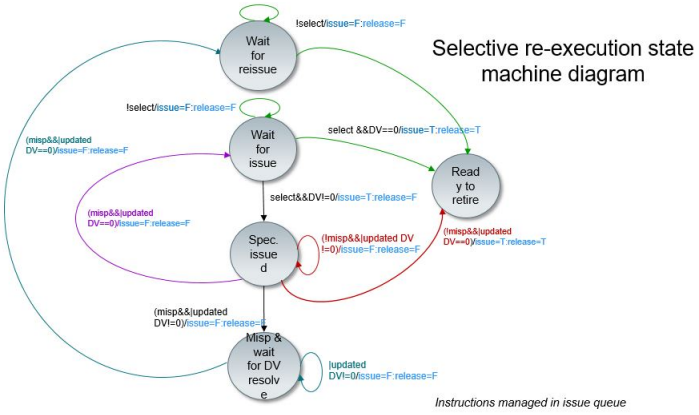

Fig. 3: Selective re-execution state machine diagram

When an instruction enters the Issue Queue, both of its source operand DVs are written to its entry, and the instruction's state (represented by 3 bits) is initialized to the WAIT_FOR_ISSUE state (Figure 3). Once the instruction is selected, it is either speculatively issued (combined DV of the instruction is not equal to 0), or non-speculatively issued (combined DV is equal to 0). If it is speculatively issued, the instruction waits in the SPEC_ISSUED state until the DV resolves to 0, or until there is a misprediction detected. If a misprediction is detected, the instruction enters the MISP_AND_WAIT_FOR_DV_RESOLVE state and stays there until the instruction's combined DV is 0. Upon reaching 0, the instruction will enter the WAIT_FOR_REISSUE state where it is issued for a second time, and thus reaching the READY_TO_RETIRE state. If no misprediction is detected, or if an instruction is initially issued non-speculatively, the instruction will reach the READY_TO_RETIRE state. Upon reaching this state we release the IQ entry and set a *released* bit in the AL. Whenever an instruction re-issues its *completed* bit in the AL is reset.

Dependence vector bits are resolved (cleared) whenever an instruction with a valid VP_ID is in writeback. At this moment, the VP_ID will be broadcasted to both the IQ and the DM. The DM will clear each bit in each entry that corresponds to the VP_ID. The same thing will happen in the IQ, but each IQ entry will also asses whether or not the broadcasting instruction is mispredicted and whether or not if it depends indirectly or directly on that broadcasting instruction. When the broadcasting instruction is mispredicted, we reset the ready bit of indirectly dependent source operands in the Issue Queue to maintain data dependency order among instructions. Instructions that must re-execute must also reset their ready bits in the ready bits table.

IV. Simulator Configurations

In the following two tables we present configurations used to evaluate our recovery strategies. Real branch prediction was not used for evaluation, but was used to test whether or not our recovery mechanisms work under real branch predictor conditions. We opted for perfect branch prediction for evaluation in order to gain a clear understanding of the performance impact of various recovery policies. In the real branch prediction scenario, we use the standard 721sim parameters. All figures mentioned in Table IV are results obtained from 100M committed instruction runs (except mcf which is run at 10M instructions due to its low IPC).

TABLE II: Table of superscalar simulator configurations.

Superscalar Structure/Feature	Configurations
Issue Queue	8/256/1024 entries
Active List	1024 entries
LSQ	1024 entries
Fetch Queue Size	128 entries
Fetch Width/Dispatch Width/Issue Width/Retire Width	8/8/8/8
L1 D$ assoc/line size/sets	4/64 bytes/256
L1 I$ assoc/line size/sets	8/64 bytes/128
L2$ assoc/line size/sets	8/64 bytes/512
Perfect Branch Prediction?	Yes
LQ/SQ Size	1024 entries
Physical Register File Size	1088 entries

TABLE III: Table of Configurations for added features.

Added Structure/Feature	Configurations
DFCM First Level Table Size	32K entries
DFCM Second Level Table Size	32K entries
DFCM Last Value Table Size	32K entries
Inverse of Probability to Increment Conf. Counters	1-32 (varied by powers of 2)
Recovery Policy	perfect/squash/selective/hybrid
GDV Size	1-320
Confidence Counter Threshold (Count at which Predictions are made)	3

TABLE IV: Table of Figures.

Figure Label	Explanation
Figure 4	IPC for squashing and ”perfect” recovery, normalized to superscalar performance w/o value prediction. IQ is set to 8 entries, other superscalar parameters follow Table II. Inverse of probability is swept from 1-32, using only powers of 2.
Figure 5	IPC for squashing and ”perfect” recovery, normalized to superscalar performance w/o value prediction. IQ is set to 1024 entries, other superscalar parameters follow Table II. Inverse of probability is swept from 1-32, using only powers of 2.
Figure 6	IPC for selective recovery, normalized to superscalar performance w/o value prediction. IQ set to 8 entries, other superscalar parameters follow Table II. GDV size swept from 1-64 bits, using only powers of 2. Hybrid recovery data uses 2 confidence counters, a squash counter and selective counter. Both saturate at 3, and the squash counter has an inverse probability of 32 and the selective counter has an inverse probability of 8, GDV is set to 16 bits.
Figure 7	IPC for selective recovery, normalized to superscalar performance w/o value prediction. IQ set to 256 entries, other superscalar parameters follow Table II. GDV size swept from 1-64 bits, using only powers of 2. A run was also made with GDV set to 320 bits.

V. Results and Analysis

A. Main Results

Figures 4 and 5 provide insight into two different utilizations of value prediction. Figure 4 shows performance benefits that can be gained from unblocking the Issue Queue. Even with a high penalty recovery approach like squashing from the head

of the AL, up to a 50% performance increase can be achieved. This performance increase is primarily due to the unblocking capabilities of squashing from the head. On the other hand, squashing from the head for a large IQ (Figure 5) results in large slow downs for each benchmark and each confidence measure. With a large IQ, squashing from the head can no longer leverage its unblocking property. Instead, performance gains will only be a result of executing instructions early. For both graphs, squashing from the head increases in performance for each benchmark as the confidence required for a prediction to be used increases. This is due to less mispredictions being experienced, and since mispredictions have such a heavy penalty, the performance will increase with less mispredictions. For some benchmarks in Figure 4, like perl and hmmer, the performance gains level off after the inverse of the probability reaches 16. This may be due to the limitations of increasing the conservatism of confidence, that is, even as the confidence measure is made more conservative there will likely be a natural amount of mispredictions. For example, a loop counter that resets may be mispredicted on the reset each time, and removing these mipredictions will ultimately lead to not being able to use any predictions.

”Perfect” recovery provides insight into what kind of performance gains can be achieved when a misprediction is not allowed into the pipeline in an oracle manner. In this case, instructions that would have used a mispredicted value will simply wait for their actual value to be ready. For both a small and large IQ, the performance decreases with increasing confidence. This is expected, as no price is paid for a misprediction, so there is always an incentive to use more predictions. An interesting result of the ”perfect” recovery is shown in Figure 5. Even when there is no penalty associated with mispredictions, the relative increase in performance is less than that of a small IQ. This indicates that the non-blocking property may be a larger contributor to performance gains than simply just executing early.

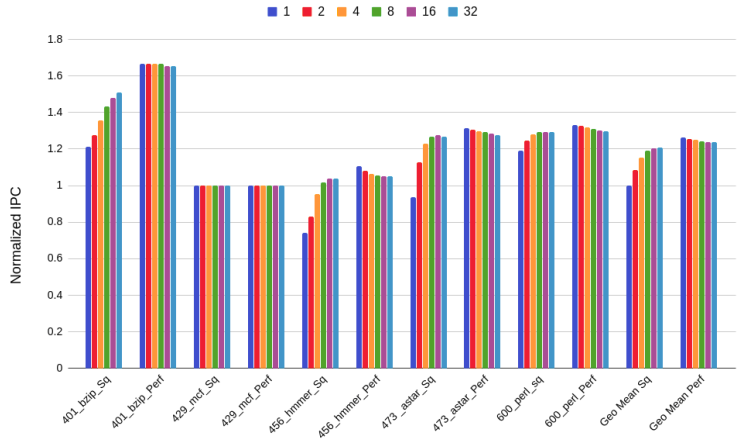

Fig. 4: Normalized IPC results for squashing and perfect recovery. IQ is set to 8 entries, and each bar represents the inverse of the probability that the confidence counter is incremented.

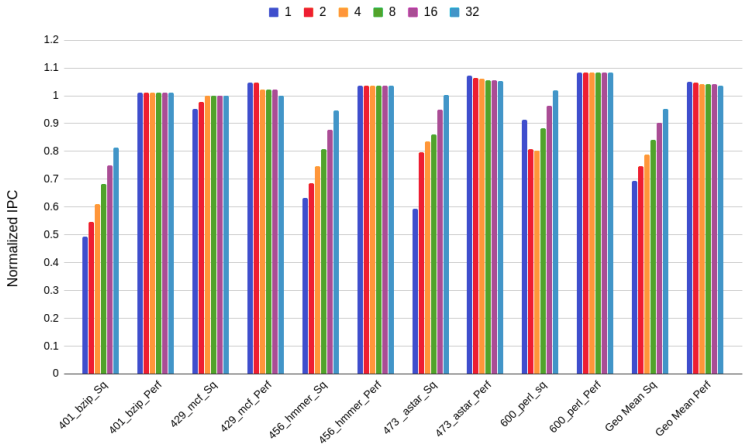

Fig. 5: Normalized IPC results for squashing and perfect recovery. IQ is set to 1024 entries, and each bar represents the inverse of the probability that the confidence counter is incremented.

Figures 6 and 7 show the results from using our selective re-execution method. From Figure 6, it is shown that our selective recovery method actually results in a slow down when compared to the baseline that does not have value prediction. This can be due to the fact that our selective re-execution recovery may increase the pressure on IQ resources, which are already a

bottleneck in this case. The extra pressure is a result of instructions being required to wait in their respective IQ entry until they are proven to be no longer speculative. This pressure can be exacerbated under cases where mispredictions result in long data dependence chains to be re-executed because each instruction in the chain will need wait for its combined DV to resolve to 0 before re-executing, and thus vacating its IQ entry. This means that each instruction in the chain will need to wait until all VP_IDs that it depends on to resolve. In other words, a new data dependence chain between instructions in the mispredicted slice is created, but now it is a VP_ID dependence. This new dependence chain can become detrimental to performance due to VP_IDs not being broadcasted until the writeback stage, ultimately ruining back-to-back execution of instructions in some cases. The performance degradation eventually levels out in Figure 6 at a GDV size of 16 bits because once this size is reached, there are no more predictions dropped due to no spot being available in the GDV.

It should be noted that the implementation of this selective recovery scheme is essentially a perfect implementation when viewed from a hardware design standpoint. For example, consider the action of setting the *released* bit in the AL whenever an IQ entry is to be released. In the current implementation, any number of instructions can write to their *released* bit at any time. From a hardware design standpoint, the bandwidth for writing to the *released* bit would need to be limited. This would translate to only allowing so many instructions to set their released bit each cycle. This could delay the re-issuing process of an instruction even longer since instructions set their *released* bit whenever they are re-issued. Not only does bandwidth need to be limited for writing this bit, but also bandwidth needs to be limited to the *completed* bit since instructions must reset this bit upon re-execution. This also translates to limiting the possible number of instructions that can be re-issued each cycle. This bandwidth analysis indicates that if this ideal selective re-execution does not provide a performance benefit, then the non-idealities in actual hardware may make the performance even worse.

Performance for selective recovery with a large IQ does not degrade or improve performance by a large amount generally. This indicates that even though we may alleviate the IQ pressure caused by this selective re-execution strategy, there is not enough performance gains from early execution to outweigh the possible performance penalties that result from the re-serialization of the data dependence graph that results from a misprediction. The performance for our hybrid recovery in Figure 6 performs better than any of the pure selective re-execution configurations, but if the normalized IPC is compared for its squash from head counter part (inverse probability of 32 in Figure 4), the performance is actually worse.

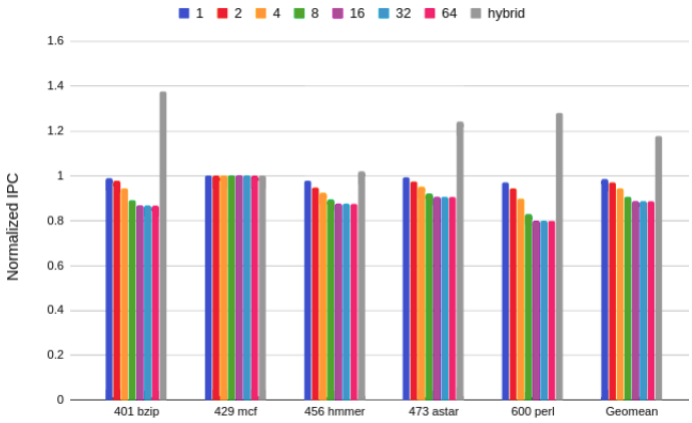

Fig. 6: Normalized IPC results for selective and hybrid recovery. IQ is set to 8 entries, and each bar besides the gray one (hybrid prediction) represents the GDV size in bits.

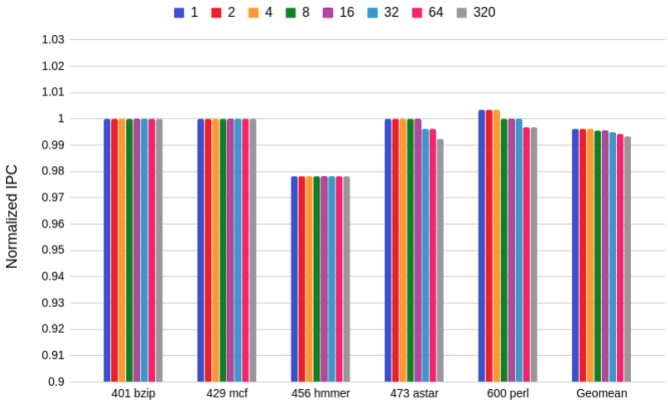

Fig. 7: Normalized IPC results for selective recovery. IQ is set to 256 entries, and each bar represents the GDV size in bits. **Graph starts at 0.9**

VI. Future Work

Retaining speculative instructions inside the Issue Queue can cause structural hazards more frequently, especially when mispredictions occur early on in a data dependence chain. To alleviate this problem, we can use a replay buffer in the hopes that DV resolution can be done in a buffer outside of the IQ. Reducing the impact of the serialization caused by a misprediction may be another area to look into. For example, once an instruction mispredicts, instructions should not be made to wait for the VP_IDs to be resolved that are after the mispredicted instruction in the data dependence graph. This technique would effectively bulk release any VP_ID that is after a value misprediction in the data dependence graph. Another possible avenue to research further would possibly be to characterize performance benefits from other non blocking techniques, such as early retiring long latency loads.

VII. Related Work

A. Value Predictors

In [4] predicting the value of register producing instructions is first introduced. In this paper they use a last value predictor due to their observation of value locality, where instructions are likely to produce the same value repeatedly. Since this paper, many efforts have been made to develop various value predictors in hopes of gaining better prediction accuracy. For example, predictors like stride predictors, have been developed in order to predict value patterns that exhibit constant strides [5] . Other predictors that use context of value history have also been developed in order to predict more complex value patterns. Such predictors include [6], which uses history of past values to make a prediction, and the DFCM predictor [1] which uses past stride history in order to predict a stride that will ultimately be used for making a prediction. Other predictors have been proposed that use branch history, such as the VTAGE predictor in [7] which uses branch history to index into many different tables that provide predictions. The table that has a hit with the longest branch history is used for making a prediction. Some predictors, like [8], have attempted to combine global branch history with stride predictors in an aim to get a per path stride predictor because different paths may result in different strides being used for a certain instruction.

B. Confidence Mechanisms

Besides simply predicting the values that are produced by instructions, a confidence measure is important in order to separate out mispredictions from correct predictions. Lipasti *et al.* in [4] use a Classification Table that has a *pred history* field that is incremented and decremented whenever there is a correct prediction or misprediction, respectively. This field is used to determine whether or not a prediction produced from the predictor should be used or not. Since this paper, different confidence mechanisms have been studied. For example, a resetting counter was investigated in [2]. This confidence mechanism was studied in the context of separating mispredicted branches from correctly predicted branches. Perais *et al.* in [7] utilize this type of clearing counter for separating value mispredictions from correct predictions, and combined this counter with probabilistic incrementing in order to devise a confidence mechanism that achieves performance increases even with a high penalty recovery approach like squashing from the head. Probabilistically incremented counters are introduced in [3]. Incrementing counters probabilistically means that whenever a counter is to be incremented, a random number generator is used to generate a random number. If this random number is a certain value, then the counter will be incremented. In [3], they show that a linear feedback shift register can be used to generate pseudo random numbers, which in turn can be used to determine if the counter is to be incremented or not. Ultimately, their goal is to show that a small counter (in terms of numbers of bits) can be incremented probabilistically in order to approximate a wider counter, thus saving storage space on confidence counters.

C. Recovery Mechanisms

Recovery mechanisms also play a large role in value prediction. Zhao *et al.* explore a selective re-execution with replay buffer style recovery mechanism. In this approach, instruction execution information is stored in the AL [9]. Whenever a misprediction is detected, indirect and direct dependent instructions are re-inserted into the IQ using the information that was stored in their respective AL entries. In order to track dependencies between instructions they introduce an approach that we built upon, which is the Dependence Matrix approach. In [9], dependencies are tracked amongst AL entries, but in our approach, the Dependency Matrix is used to track dependencies amongst physical registers. Zhao *et al.* also show that as misprediction recovery increases in latency, performance gains from value prediction go away quickly. Selective re-execution that keeps instructions waiting in their IQ/Reservation Station entries until they are no longer speculative have been studied in both [4] and [10]. Squashing from the head has also been studied in [7], and they show that this simple but heavy-handed recovery mechanism can be viable under the condition that a relatively conservative confidence mechanism is employed.

REFERENCES

[1] B. Goeman, H. Vandierendonck, and K. d. Bosschere, "Differential FCM: increasing value prediction accuracy by improving table usage efficiency," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pp. 207–216.

[2] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning confidence to conditional branch predictions," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pp. 142–152.

[3] N. Riley and C. Zilles, "Probabilistic counter updates for predictor hysteresis and stratification," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pp. 110–120.

[4] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pp. 226–237.

[5] R. J. Eickemeyer and S. Vassiliadis, "A load-instruction unit for pipelined processors," vol. 37, no. 4, pp. 547–564.

[6] Y. Sazeides and J. E. Smith, "Implementations of context-based value predictors."

[7] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 428–439.

[8] T. Nakra, R. Gupta, and M. L. Soffa, "Global context-based value prediction," in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pp. 4–12.

[9] H. Zhou, C.-y. Fu, E. Rotenberg, and T. M. Conte, "A study of value speculative execution and misspeculation recovery in superscalar microprocessors."

[10] B. Rychlik, J. Faistl, B. Krug, and J. P. Shen, "Efficacy and performance impact of value prediction," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, pp. 148–154.

VIII. Appendix

This section provides some auxiliary results and analysis that did not make it into the main report. Section VIII-A provides some extra results regarding value prediction, Section VIII-B provides some insights into how to size the GDV, and Section VIII-C provides more discussion on the implementation issues that were touched upon in the results section.

A. Value Predictor Data

From Figure 8, it is shown that as the confidence required to make a prediction is increased, the number of useful correct predictions that are used decreases. Decreasing the amount of correct predictions that are used may lead to decreasing the possible performance gains from value prediction, but also a higher confidence required to make a prediction may filter out many mispredictions, thus avoiding misprediction recovery penalties. The benefits from filtering out more mispredictions was shown in Figure 4 for the squashing recovery method, but it was also shown that there is a limit to the benefits attained from increasing the confidence required for a prediction. For example, for the benchmarks perl and hmmer, the performance no longer increases with increasing confidence required for a prediction once an inverse probability of 16 is reached. This is most likely due to limitations of reducing the number of mispredictions, that is, there will always be some natural amount of mispredictions no matter how conservative the confidence is. Figure 9 provides some evidence of the limitations of increasing confidence. As the confidence required is increased, the average number of mispredictions first drops off quickly, but after reaching a probability inverse of 8, there is not much benefit in increasing the confidence required for a prediction.

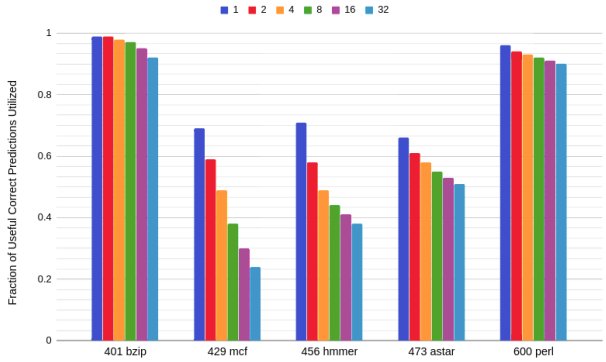

Fig. 8: Fraction of useful correct predictions used. Useful means that the correct prediction is for an instruction that will be committed. This fraction is computed by dividing the number of correct predictions classified as high confidence by the total number of correct predictions that the value predictor produces. Each bar represents a different confidence measure

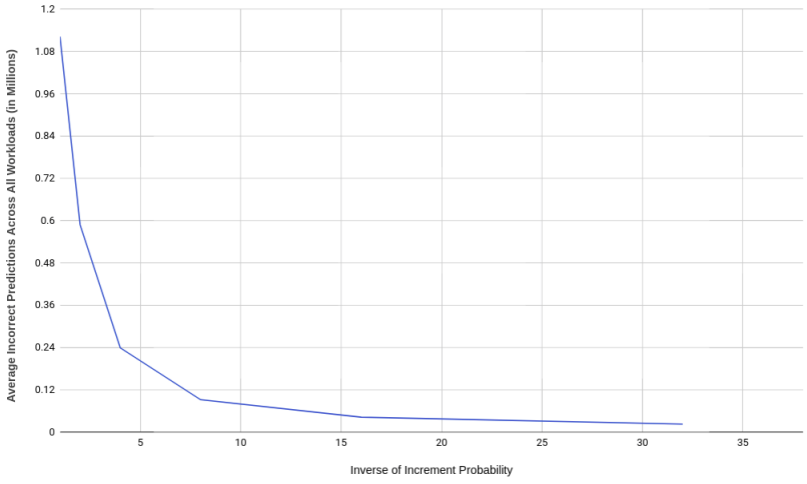

Fig. 9: Average number of mispredictions across all workloads (in millions) for each inverse of probability studied.

TABLE V: Table of total useful correct predictions for each benchmark, each benchmark is evaluated for 100M committed instructions, except mcf with is 10M committed instructions.

Benchmark	Total Number of Available Correct Predictions
bzip	70105574
mcf	2825174
hmmer	33707026
astar	50088429
perl	63285830

B. GDV Data

One important detail about our recovery method is the size the GDV should be made in order to not drop any predictions due to no GDV bit being available. Figure 10 shows the number of predictions that are dropped for different GDV sizes. One important characteristic of this graph is the number of GDV bits required to achieve no dropped predictions for different IQ sizes. For an IQ size of 8, a GDV size of only 16 bits is required in order to avoid dropping any predictions, but for a large 256 entry IQ a GDV size of more than 64 bits is required in order to avoid dropping predictions. Thus, there is a relationship between the IQ size and the GDV size required for not dropping any predictions. This is because a large IQ buffers more instructions, and if a lot of these instructions are value predicted, there will be a lot of valid VP_IDs in flight at the same time. This will result in high pressure on the GDV resource. Though, it is worth noting, for smaller IQs the required GDV is rather small compared to the number of bits required for a static dependence vector. A static dependence vector size will always be equal to the number of physical registers in the processor.

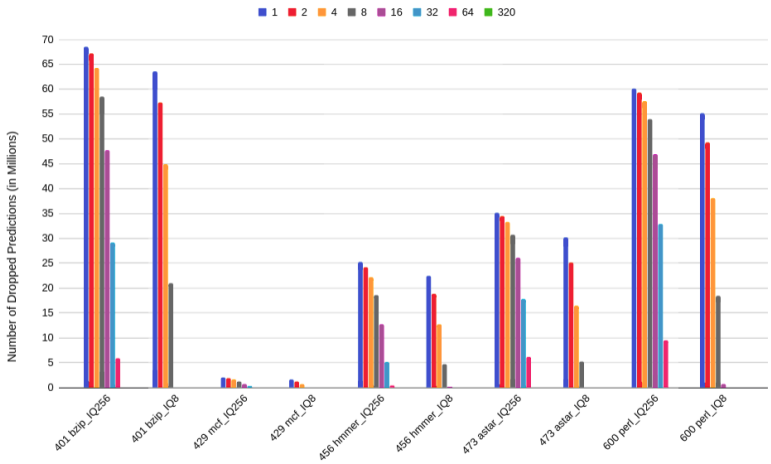

Fig. 10: Number of dropped predictions (in Millions) for each benchmark, and for both 8 and 256 IQ entries. Each bar represents a different GDV size.

C. Implementation Challenges

Previously, in the results section, it was stated that there was some implementation issues regarding our dependence vector recovery technique. There are three major implementation issues regarding our design, i.e. the number of write ports required for the AL *completed* bit, the AL *released* bit, and the ready bits table. The AL *completed* bit needs to be cleared for an instruction whenever it re-issues because an instruction may have already set this bit speculatively. Thus, added bandwidth to this bit for our selective re-execution method would need to be as much as the issue width, since it may be possible that all issue lanes are used to re-issue instructions that were speculative. If this is undesirable, the bandwidth can be limited by limiting the number re-issued instructions allowed each cycle. For the *released* bit, this bit needs to be set whenever an instruction moves to the READY_TO_RETIRE state. The transition to this state can either happen after the instruction is issued non-speculatively, or the instruction's DV resolves to 0 after it has been speculatively issued. Thus, there may be a scenario where many instructions may try to set their *released* bit in a cycle. In order to limit this bandwidth, a limited number of instructions that are in the READY_TO_RETIRE state would be allowed to write to their respective *released* bits. The bandwidth issue for the ready bit table is a little more complex. This is because this table needs to be written to by possibly many instructions whenever a misprediction occurs. Every instruction that depends on the misprediction must reset their ready bit in this table. Unlike the other bits, writing to this table cannot be delayed until a later point because instructions coming through dispatch will need to know the readiness of their source operands. Thus, if the bandwidth is to be limited to this table, it may be required that the dispatch stage be stalled while the ready bit table is repaired over multiple cycles.

D. Tables of Baseline Results

TABLE VI: Table of baseline IPCs for a 1024 entry IQ, superscalar parameters follow Table II.

Benchmark	**IPC**
bzip	2.88
mcf	0.43
hmmer	2.28
astar	2.66
perl	3.00

TABLE VII: Table of baseline IPCs for an 8 entry IQ, superscalar parameters follow Table II.

Benchmark	**IPC**
bzip	0.90
mcf	0.15
hmmer	1.77
astar	1.27
perl	1.78