Using Dataflow Based Context for Accurate Value Prediction

Renju Thomas
ECE Department
University of Maryland
College Park, MD 20742
renju@eng.umd.edu

Manoj Franklin
ECE Department and UMIACS
University of Maryland
College Park, MD 20742
manoj@eng.umd.edu

Abstract

We explore the reasons behind the rather low prediction accuracy of existing data value predictors. Our studies show that contexts formed only from the outcomes of the last several instances of a static instruction do not always encapsulate all of the information required for correct prediction. Complex interactions between data flow and control flow change the context in ways that result in predictability loss for a significant number of dynamic instructions. For improving the prediction accuracy, we propose the concept of using contexts derived from the predictable portions of the data flow graph. That is, the predictability of hard-to-predict instructions can be improved by taking advantage of the predictability of the easy-to-predict instructions that precede it in the data flow graph. We propose and investigate a run-time scheme for producing such an improved context from the predicted values of previous instructions. We also propose a novel predictor called dynamic dataflow-inherited speculative context (DDISC) based predictor *for specifically predicting hard-to-predict instructions. Simulation results verify that the use of dataflow-based contexts yields significant improvements in prediction accuracies, ranging from 35% to 99%. This translates to an overall prediction accuracy of 68% to 99.9%.*

1 Introduction

Recently, data value prediction was proposed for overcoming the effects of data dependences in extracting instruction level parallelism (ILP) [5]. The interest in data value prediction has been growing, and a wealth of literature is already available on designing different types of data value predictors [2] [3] [4] [5] [7] [11]. Some of the best predictors proposed so far use a hybrid approach, by using a combination of different predictors such as last value, stride and context predictors.

In a standard context-n predictor, the values from the n previous dynamic instances of a static instruction are used to form the context. A selection logic decides the value to be predicted for a particular context, based on a small number of saturating counters associated with each value. Thus, the traditional context-based predictor is a statistical predictor that relies on statistical correlation between the context derived from previous instances and the outcomes of future instances. Naturally, this prediction mechanism may not be able to give a correct prediction in many cases, because of the lack of correlation between such a context and the actual outcome of the instructions. Our studies confirm that contexts derived only from the previous dynamic instances of an instruction are not sufficient to accurately predict the outcome of many instructions. This is because of the following reasons:

1. The previous instances of the instruction need not be in the dynamic dataflow path that determines the current instance's result.
2. Additional information may have been injected into the dynamic dataflow path of the current instance after the previous instance was executed (due to events such as control flow changes), affecting the value produced by the current instance.

An earlier study [7] found that contexts derived from the history of the recent branch outcomes and the result produced by the last executed instruction have some statistical correlation with the result to be predicted. However, this modified context does not specifically consider dataflow, and therefore need not always have correlation with the outcomes of future instances.

The context used for predicting a dynamic instruction[1] should encapsulate its predictability information. We propose a scheme to generate such a context for instructions that are currently unpredictable. In our scheme, the context is derived from the closest predictable values in the instruction's dataflow path.

[1]In this paper, by predicting an instruction, we mean predicting its result.

Thus, our methodology of providing the "right context" involves correlating the unpredictable instructions to their predictable producer instructions in the dynamic dataflow graph.

An analysis of unpredictability characteristics given in Section 2 reveals points in the data flow path leading to the instruction to be predicted, where such closest predictable instructions can be identified to form the context. We introduce a novel predictor called *dynamic dataflow-inherited speculative context (DDISC) predictor*. This context based predictor uses the dataflow-inherited context only in cases needed, namely, for unpredictable instructions in a conventional context predictor. Also, the context used adapts more closely, tracking the "actual context" in which the instruction to be predicted is in.

The rest of this paper is organized as follows. Section 2 describes unpredictability and derives the required background for understanding the relevance of the proposed dynamic dataflow-inherited speculative context-based predictor. Section 3 explores the causes of unpredictability. Section 4 explains the concepts of this novel predictor and discusses the hardware considerations for DDISC. Section 5 gives the simulation results. We conclude the paper in Section 6.

2 Characteristics of Currently Unpredictable Instructions

A recent paper had modeled the predictability of data values, and had provided some useful insights into the prediction process and the predictability of data values [8]. We analyze the *unpredictable instructions* in a conventional predictor so as to obtain insights into the reasons why unpredictability occurs.

2.1 Classification of Unpredictability

To obtain some meaningful insights from the unpredictability observed in predictors, we do a classification for them. As our main goal is to understand the reasons why a particular instruction is unpredictable, our classification method focuses on the causes for unpredictability. In particular, we differentiate between whether an instruction is a generating point for unpredictability, or is simply passing on the unpredictability of its source operand producers. It would indeed be interesting to investigate what causes some instructions to be unpredictable when all of its operand producing instructions are predictable. In our study, we categorize unpredictable instructions into the following two categories:

- *Unpredictability generates:* An unpredictability generate instruction is an unpredictable dynamic instruction whose source operand producer instructions are predictable. They can be thought of as the points in the dynamic dataflow graph where unpredictability is started[2].

- *Unpredictability propagates:* An unpredictability propagate instruction is an unpredictable dynamic instruction that has at least one source operand whose producer instruction is unpredictable.

Figure 1 shows an example dataflow graph depicting producer-consumer relationships between several instructions, without explicitly representing the control flow. In the figure, dataflow through registers are indicated by straight lines with arrows, and dataflow through memory are indicated by dotted lines with arrows. The figure has one store instruction (I9), marked by an unshaded circle with the letter **S** inside. The instructions that are very lightly shaded—I1, I2, I3, I8, and I9—are predictable. Instructions marked by dark circles—I4 and I12—are unpredictability generates. Among these, I12 is an unpredictability generate load instruction, for which the load value is not predictable, although the source register producer instruction (I8) is predictable. In Figure 1, the instructions marked by moderate shade—I5, I6, I7, and I10—are unpredictability propagates.

The unpredictability that is generated at unpredictability generates and then propagated through the unpredictability propagates eventually terminates at some predictable instructions. That is, at some point in the dynamic dataflow graph, the consumer becomes predictable although one or both of its operand producers are unpredictable. In a dynamic dataflow graph, starting from an unpredictability generate instruction, if we consider all the unpredictability propagate instructions that are associated by a producer-consumer relation, we can think of the existence of an *unpredictability chain* up to the point where the unpredictability is terminated. For instance, in Figure 1, instruction I10 belongs to the unpredictability chain that originated at I4.

[2]Our set of unpredictability generates is only loosely comparable to the set of predictability terminates in Sazeides and Smith's model [8]. Some cases of predictability termination in [8], such as when one of the operand producers is unpredictable and the other is predictable, do not come into the category of unpredictability generates in our study.

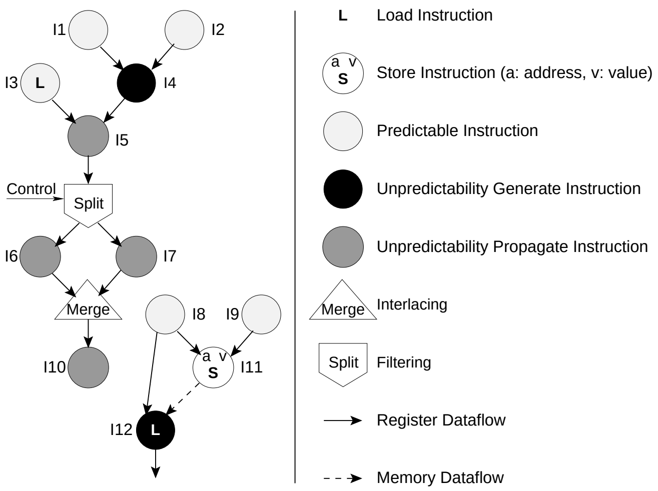

Figure 1: An example dataflow graph illustrating different types of unpredictable instructions and causes of unpredictability.

2.2 Measurement Framework and Results

We use a trace based MIPS ISA simulator for the measurements. All instructions producing a single register result except NOPs are considered for data value prediction. We use a total of 9 benchmark programs — 6 from the SPEC95 suite and 3 from the SPEC2000 suite. The programs, along with the inputs and flags used, are: (i) `compress95 in`, (ii) `m88ksim ctl.raw`, (iii) `li test.lsp`, (iv) `go 50 21 9stone.in`, (v) `gcc stmt.i`, (vi) `vortex vortex.raw`, (vii) `gzip input.compressed 2`, (viii) `mcf inp.in`, (ix) `twolf ref`. All programs are run for 500 million or up to completion, depending on whichever occurred earlier.

To give an optimistic treatment to conventional predictors, we use a hybrid data value predictor consisting of a stride predictor and an order 3 context predictor with a 256K entry table. The stride predictor table and the VHT of the context predictor are indexed using the lower truncated bits of the PC. The VPT is indexed by a shift and xor based hash of the 3 values obtained from the VHT entry. Each VPT entry keeps 4 values and selects one of them as the prediction, based on 3-bit confidence counters. A prediction is made only if the maximum counter value is above a threshold value of 4. The stride predictor predicts if the selected entry's state is in "predict". Otherwise, the context predictor is selected for prediction and either predicts, mispredicts or does not predict depending on its confidence estimator. The unpredictability and predictability characteristics of the hybrid predictor are shown in Figure 2(i). The rest of the paper focuses on the unpredictability portion of Figure 2(i). Our results in Section 5 show that it is indeed possible to predict these unpredictable instructions once we understand the characteristics of unpredictability and the nature of their causes.

The relative percentages of unpredictability generates, propagates, and terminates are shown in Figure 2(ii). Note that unpredictability terminates are predictable instructions. Figure 2(ii) shows that for a majority of the benchmarks, there are more unpredictability propagates present than unpredictability generates. We next look at the possible causes of unpredictability generation.

3 Causes of Unpredictability Generation

The previous section classified unpredictable instructions and presented a statistical breakdown of unpredictable instructions. In this section, we explore the causes for unpredictability generation. Such an exploration is of utmost importance in coming up with schemes for improving the prediction accuracy. The causes for unpredictability generation can be classified into the following taxonomy:

- Unpredictable register source operand (although the results produced by the operand's producer instruction(s) are predictable)
  - Interlacing due to control flow merges (i.e., there are multiple static producers for an operand)
  - Filtering due to control flow branching (i.e., only a subset of a static instruction's results are seen by a consumer instruction)
- Unpredictable result operand, given predictable source operands
  - Aliasing due to limited table size and context size
  - Algorithmic limitation
- Unpredictable memory source operand
  - Unpredictable store value
  - Unpredictable program input value

Often, a combination of the above factors also contribute to unpredictability.

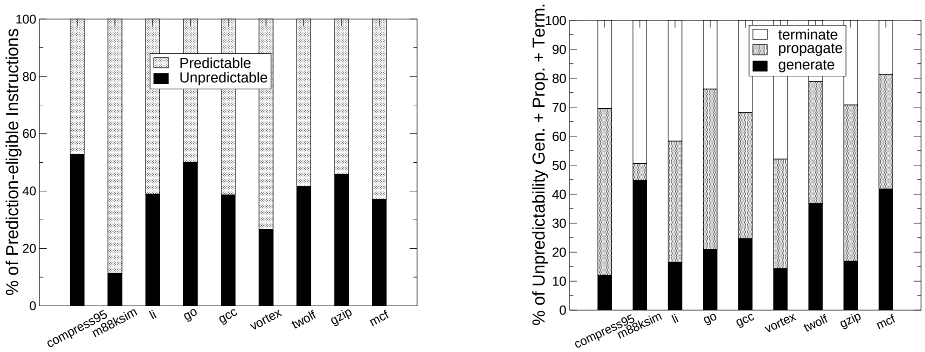

Figure 2: (i) Prediction characteristics of the Hybrid Predictor used for Unpredictability Measurement; (ii) Relative Percentage of Unpredictability Generates, Propagates, and Terminates

3.1 Unpredictable Register Source Operand

In this case, the value sequence seen at the input side of the instruction itself is not predictable. For the unpredictability generate instructions, the instruction(s) that produce the input values are predictable. This means that predictability is lost for these values in the process during which the predictable output of the producer of the register source operand was read at the input of the current instruction. The major cause for the information loss described above is control flow change. We use the terms "interlacing" and "filtering" to refer to these control flow induced information loss from a producer's output to a consumer's input.

3.1.1 Interlacing due to Control Merge

Consider the simple example given in Figure 3(a). Assume that instructions I1 and I2 produce one of the operand values used by instruction I3. Assume also that the results produced by I1 follow the pattern `a b c a b c`... . Similarly, assume that I2 produces the value sequence `p q r p q r`... . Both of these sequences are perfectly predictable using a conventional context predictor. Now, let the operand for I3 be produced by instructions I1 and I2 in the order I1 I2 I2 I1 I2 I1... . The corresponding value sequence at the input of I3 would be `a p q b r c a p`..., which is unpredictable. With this unpredictable sequence at I3's input, I3's output becomes unpredictable. Thus when the two streams interlace, the values become unpredictable because of losing the real context; all that is received by I3 is a somewhat random sequence of values. To make the unpredictable instruction predictable in this case, the context used for the prediction has to encapsulate information about the operand producer of the current instance.

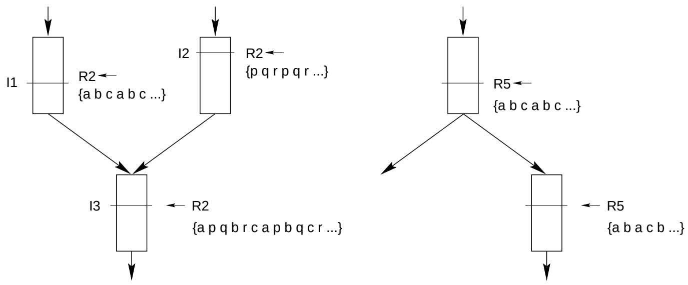

Figure 3: Example Control Flow Graph Illustrating (i) Interlacing due to control flow merge; (ii) Filtering due to control flow split

3.1.2 Filtering due to Control Split

If a conditional branch separates a predictable producer instruction and its consumer instruction, as shown in Figure 3(b), the consumer will not see the entire predictable sequence produced by the producer, if the branch direction keeps changing. This loss in information can cause the resultant output sequence produced by the consumer instruction to be unpredictable. To be able to correctly predict the unpredictable instructions affected by filtering, the predictor needs to be fed with the entire predictable sequences of the producers of the source operands. In other words, the predictor has to be provided with a "right context" that can correctly describe the value of the current instance.

3.2 Unpredictable Result Operand

3.2.1 Aliasing Due to Mapping and Physical Limitation of Predictor

In this case, the sequence of results would be "predictable" by an infinite-entry predictor, but there is information loss due to the limited representation of these values in finite sized prediction tables. An example of this is aliasing in which different instructions map to the same entry in a prediction table. This mapping can cause loss of information relevant to prediction. Information loss also happens due to partial representation of values in limited size prediction tables. Techniques are available to reduce aliasing and to reduce information loss in the prediction tables. To give one example, aliasing due to mapping can be alleviated using multiple predictors with different hash functions and then using a voting scheme to choose one of the predictions.

Another limitation of the predictor arises from the fixed context history depth of the context predictor used for prediction. This manifests itself either as some instructions not predicted correctly during the warm-up time of the predictor or the instruction never getting predicted correctly. Looking at it from the perspective of the given predictor, we can as well think that it is not using the "right context" for recording history and making predictions. Better prediction schemes need to generate/regenerate the "right context" for the instruction, so as to make correct predictions.

3.2.2 Algorithmic Limitation of Predictor

In this case, the source operands of an instruction are predictable, but the instruction operates on the source values in such a way that its result is unpredictable with a particular predictor, even if it has no size restrictions. This can happen for arithmetic/logical instructions, especially when one of the source operands is stride predictable and the other is context-predictable. The output sequence of such an instruction is likely to be unpredictable by a stride predictor as well as by a context predictor that derives the context from previous instances of the instruction. This unpredictability can be attributed to the inherent limitations of the algorithm implemented by the predictor. Here, the predictor context used might be weak so that the required predictability threshold is not reached and hence the instruction is unpredictable. Again, a "better context" that describes the actual context in which the instruction produces the current value should be able to predict such instructions better.

Algorithmic limitations of a predictor can also manifest in other ways, such as *warm-up* or *inertia*. When the predictor is encountering an instruction or a context for the first time, it is difficult to correctly predict the result. In fact, it takes several instances of the same instruction and the same context to set up the appropriate histories. This type of warm-up can be termed as *cold start*. Perhaps even more important are the cases affected by a multitude of other reasons that require the predictor to dynamically adapt in a better manner. For instance, when filtering occurs for an instruction once, the context gets corrupted. The effect of this corruption stays in the predictor for several future instances, although the impact on the unpredictability might vary. It is difficult to clearly separate these causes.

3.3 Unpredictable Memory Source Operand

Another possible cause for the generation of unpredictability is the partial information about the load value available at the input of the load instruction. In this study, we consider the load instruction to have only the address as the input. So, a load becomes an unpredictability generate if its address producer is predictable, but the loaded data sequence is not predictable. We consider two different cases for unpredictable load value, based on what caused the unpredictability.

Unpredictable Store Value: When the store value producer instruction is unpredictable, its unpredictability gets propagated to the corresponding load instruction that uses the value. In this case, the unpredictability is propagated through memory.

Unpredictable Program Input Value: There is also another situation that causes a load instruction to be unpredictable. This is the case when a program input value has been placed in a memory location by an I/O routine.

4 Improving Prediction Accuracy Using Dynamic Dataflow-Inherited Speculative Context

From the previous discussion, we see that the causes of unpredictability are varied. Hence, an approach to making the unpredictable instructions predictable by means of separate solutions for each of the unpredictability causes can become overwhelming. A unified approach that works for all cases is highly desirable.

From the analysis of the causes for unpredictability in the previous section, it becomes evident that the context to be used for the prediction of hard-to-predict instructions should not be based merely on the values produced by their previous instances. We propose to use the conventional context for the easy-to-predict instructions, and switch to a better context when encountering hard-to-predict instructions.

4.1 Dataflow-Inherited Speculative Context (DISC)

A trivial way to provide a better context for non-load instructions is to use the instruction's PC and operand values, all of which together uniquely identify its result. However, with such a scheme, the prediction of an instruction is delayed until all of its source operands become available, by which time the instruction can be executed in a functional unit to obtain the result!

We propose to perform the prediction earlier in the pipeline, by deriving the context from the *predicted* values of the producer instructions of the instruction to be predicted. Thus the context is derived from the *dynamic dataflow graph*, and is *data-speculative* in nature. For an unpredictability generate instruction, its producer instructions are predictable, and therefore those predicted values can be used to form the "right context".

For an unpredictability propagate instruction, however, using the predicted values of its producers will not help, because at least one of the producer instructions is unpredictable. For these instructions, finding the predictable values to use for signature calculation is not very obvious, and requires additional thought. These set of predicted values should have the following two properties:

- They should have a strong correlation to the value to be predicted, and must have components representative of as many dataflow paths that influence the value to be predicted.
- They must be predictable with the conventional predictor, permitting us to reliably use predicted values (instead of actual values) to form the context.

In the dynamic dataflow graph, the closest points where we can get predictable values that affect this instruction's result are the predicted values of the producers of the unpredictable generate(s) at the beginning of the unpredictability chain(s). For example, in Figure 1, for instruction I5, the predictable producers mentioned above are {I1, I2, I3}. Note that the same set can be obtained by taking the union of the set for I4 (a producer of one of the operands of I5), and I3 (the producer of the second operand of I5). Thus, we can say that in general, the context of an unpredictability propagate can be formed as a combination of the context(s) of its source operand producer instructions. Such an approach satisfies both of the criteria mentioned above.

To summarize, the improved context for each unpredictable instruction is formed only from the improved contexts of its producer instructions, irrespective of whether it is an unpredictability generate or propagate. One problem that remains to be tackled is that when instructions have multiple source operands, the contexts start growing in size, as we move down an unpredictability chain. In order to overcome this problem, we *compress* the contexts, as we move down an unpredictability chain.

4.2 Systematic Generation of DDISC: The Concept of Signatures

In this section, we propose one way of systematically deriving contexts from the dynamic dataflow graph. The context is determined by assigning a *signature* to each node in the dataflow graph. For signature formation, we classify the prediction-eligible instructions into the following three classes, from the conventional predictor's point of view: (i) predictable instructions, (ii) unpredictable non-load instructions, and (iii) unpredictable load instructions.

Signature of Predictable Instructions: We define the signature of a predictable instruction to be its value predicted by the conventional predictor. For instance, in Figure 1, the signatures of I1, I2, I3, I8, and I9 are their predicted values themselves.

Signature of Unpredictable Non-load Instructions: We let the signature of an unpredictable non-load instruction to be inherited from the signatures of its operand producers. When there are multiple operands, the signatures of the producers need to be compressed to keep their lengths manageable; thus, there can be aliasing in the signatures due to this compression. One way of doing this compression is to take the EXOR of the producers' signatures[3]. Thus, we define the signature of an unpredictable non-load instruction to be the EXOR of the signatures of their operand producers. For instance, the signature of instruction I4 in Figure 1 is formed by taking the EXOR of the signatures of its operand producers, I1 and I2.

Signature of Unpredictable Load Instructions: For each unpredictable load instruction, we can think of

[3]If two of the signatures to be EXORed are the same, then instead of taking their EXOR, one of the identical signatures is taken.

an immediately preceding store instruction that wrote the value into the same memory location (provided the load is not loading a program input data value). The signature of the load is inherited from the signature of the producer of the register value of this store instruction. For instance, the instruction I12 in Figure 1 is an unpredictable load. Note that its signature is not related to the signature of I8, its address register producer instruction, contrary to what might seem to be from a first look at Figure 1. Instead, I12's signature is inherited from that of I11's store value producer, I9.

Handling Control Split-Merger within Unpredictability Chain: In Figure 1, the result of I10 depends on whether the dataflow path taken for the current instance is via I6 or I7. With the above signature generation scheme, the signature of I6 and I7 are the same, namely the signature of I5. Hence, the signature of I10 will be the same, irrespective of the path taken through the dataflow graph. Therefore, information about the dataflow path taken inside the unpredictability chain must also be included in the signature. One way to track the unpredictability path followed dynamically is as follows. When a signature is computed for an unpredictable instruction, rotate the calculated signature by a value determined by the instruction's PC value. In this example, this rotation of signature ensures that I10's signature would be different when the path taken is through I6 and through I7.

4.3 Hardware Structures for Signature Calculation

A signature serves two important purposes: (i) as an improved context for predicting the corresponding instruction, and (ii) as a vehicle for conveying a better context from producer to consumer instructions. The signature to be used for predicting each unpredictable instruction has to be dynamically generated and recorded in an efficient manner. We therefore need an efficient hardware structure to store the signatures of past instructions. The main issues are: (i) how to limit the storage space (i.e., discard signatures when no longer needed), and (ii) how to link the producers and consumers efficiently?

4.3.1 Signature Register File

For storing the signatures of register-result producing instructions, and to effectively link register value producers and consumers, we propose to use a *signature register file (SRF)*, which is very similar to a register file. The SRF has as many entries as the number of registers defined in the instruction set architecture. Figure 4 shows the structure of the signature register file within a DDISC predictor. Each SRF entry can store one signature. For an unpredictable instruction whose destination register is `Rx`, and the source operand registers are `Ry` and `Rz` respectively, we define the SRF entries corresponding to register `Rx` to be composed of the hashed SRF entries of register `Ry` and `Rz`. For non-load instructions, if there is only a single source register, say `Ry`, then the SRF entry for `Rx` is copied from the SRF entry for `Ry`. When there are multiple source operands for an unpredictable instruction, its SRF entry is determined by a hash operation (such as EXOR) of the corresponding SRF entries of the operand registers. For an unpredictable load instruction, there is an exception to be made to the above rule, as explained below in Section 4.3.2.

4.3.2 Signature Memory Buffer

As discussed in Section 4.2, we need a hardware mechanism to act as the interface between stores and loads that have a producer-consumer relation, so as to speculatively communicate the signature values of stores to dependent unpredictable load instructions. The main issue is how to establish the producer-consumer relation between stores and loads. Two different possibilities exist here:

- *Case 1: Both the store and load addresses are available:* In this case, the linking of the store and the load can be done in an unambiguous manner.

- *Case 2: The store address and/or the load address is not available:* In this case, we have to do speculative store-load linking [1] [6] [9]. In these mechanisms, store-load bypassing enables the direct communication of the store's signature to a potentially data-dependent load. We can also do this linking by predicting the addresses of all stores and loads having unknown addresses, and by using the using the predicted addresses to speculatively pass signatures.

The hardware structure we use for store-load linking is called a *signature memory buffer (SMB).* There have been a plethora of schemes recently to successfully link the dependent stores and loads [1] [6] [9]. An SMB implementation could use one of the store-load linking techniques similar to store caches, store sets, or other store-load memory bypassing techniques. An efficient implementation is heavily dependent on the specific microarchitecture chosen, and will have to be tuned based on the hardware configurations used.

4.4 Signature Generation Algorithm and Example

The formal algorithm for generating the signature for each register-value producing instruction and memory-value producing instruction (i.e., store) is given below for a specific instruction `I`.

```
if (I is a store)
   SMB[*] ← SRF[a]; /* Mem[f(Ra)] ← Rx */
else if (I is predictable) /* Rx ← ... */
   SRF[x] ← Predicted value of I;
else if (I is a non-load) /* Rx ← Ry f Rz */
   SRF[x] ← SRF[y] ⊕ SRF[z] rotated left by
                              last 5 bits of PC;
else if (I is a load) /* Rx ← Mem[f(Ra)] */
   SRF[x] ← SMB[*] rotated left by
                              last 5 bits of PC;
```

`*`: depends on SMB implementation

Signature Generation Algorithm

Consider again the example dataflow graph given in Figure 1. For simplicity of explanation, assume that instruction Ix in Figure 1 writes its result into register `Rx`. Assume also that the last 5 bits of instruction Ix's PC is x. In this example, each signature is rotated by the number given by the last 5 bits of its PC value. Let SRF[x] denote the signature register file entry corresponding to register `Rx`. The signature determination for the example code of Figure 1 is given in Table 1.

Table 1: Illustration of Signature Determination for the Example given in Figure 1

Inst	T	Update of Signature Register File		
I1	P	SRF[1]	←	Predicted value of I1
I2	P	SRF[2]	←	Predicted value of I2
I3	P	SRF[3]	←	Predicted value of I3
I4	U	SRF[4]	←	SRF[1] ⊕ SRF[2] rotated left by last 5 bits of PC, i.e., 4
I5	U	SRF[5]	←	SRF[3] ⊕ SRF[4] rotated left by 5
I6	U	SRF[6]	←	SRF[5] rotated left by 6
I7	U	SRF[7]	←	SRF[5] rotated left by 7
I8	P	SRF[8]	←	Predicted value of I8
I9	P	SRF[9]	←	Predicted value of I9
I10	U	SRF[10]	←	SRF[6] or SRF[7] rotated left by 10
I11	S	SMB[*]	←	SRF[9]
I12	UL	SRF[12]	←	SMB[*] rotated left by 12

T: Type, P: Predictable instruction, U: Unpredictable non-load instruction, UL: Unpredictable load instruction, *: Depends on SMB implementation

4.5 Combination of Conventional and DDISC Predictors

Our signature-based scheme is a unified generic scheme that works for all causes of unpredictability, except those due to cold start. There will be some predictability information loss during signature determination due to hashing. Hence the predictability may be somewhat less for the unpredictability propagates, as they are farther down the unpredictability chain.

Figure 4 gives a block diagram of a DDISC predictor. The DDISC table structure is equivalent to that of a conventional last value predictor (LVP), except that it is indexed with the signature bits as opposed to the PC bits used in the LVP. Thus, each entry in the DDISC table stores the last value, along with a 2-bit saturating counter based confidence estimator [11]. The DDISC table is accessed by first reading the SRF entry corresponding to the destination register, and then using this signature value in folded form to index into the DDISC table. The DDISC predictor outputs a valid prediction only when the saturation counter is above a predefined threshold value. Otherwise, it makes no predictions. The table is updated when the actual value becomes available after execution. In our experiments, the saturating counter is incremented by two on a correct DDISC prediction and is decremented by one on a misprediction.

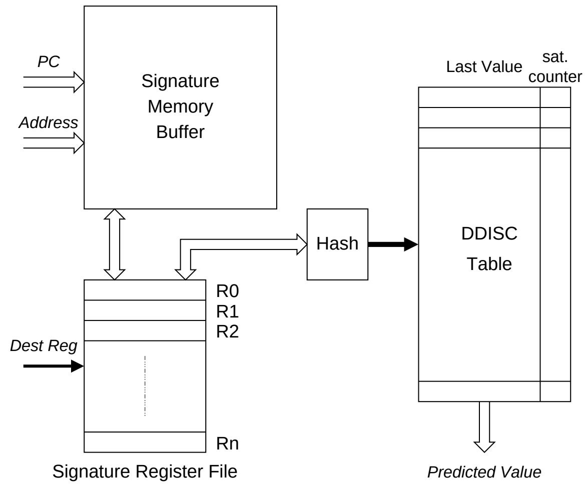

Figure 4: Hardware Components of a DDISC Predictor

Figure 5 gives a block diagram of a combined {conventional, DDISC} predictor. The confidence estimator in the conventional predictor acts as a selector for the DDISC predictor. Register-result producing instructions that are indicated as below the threshold for prediction by the confidence estimator of the conventional predictor are considered for prediction by the DDISC predictor. Note that the selection happens before the actual prediction.

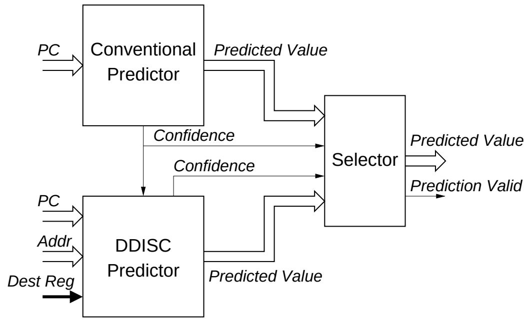

Figure 5: Block Diagram of Combined {Conventional, DDISC} Predictor

The DDISC predictor's performance is sensitive to several parameters such as the efficiency and accuracy of the conventional predictor on which the DDISC predictor relies on for predictable nodes, the information loss due to determining the signatures using limited hardware resources, and the aliasing in the DDISC table. When the prediction accuracy of the conventional predictor varies, the efficiency of the DDISC predictor will vary. Further, mispredictions in the conventional predictor affect the DDISC predictor in two ways. First, the DDISC predictor mistakes these mispredicted instructions as predictable and correlates on wrong values. Second, the DDISC predictor does not consider these mispredicted instructions for its prediction. Hence, improving the conventional predictor's confidence estimation will improve the DDISC predictor's performance.

The additional hardware cost for the combined {conventional, DDISC} predictor over the conventional predictor, is that for the DDISC table, the SRF, and the SMB.

5 Experimental Analysis

In this section, we perform an experimental evaluation of the DDISC predictor that we proposed in the previous section. Being the first paper on the concept, the experimentation is geared towards demonstrating the feasibility and potential of the concept. An exhaustive evaluation of all interesting cases is not possible due to space restrictions. Furthermore, we also do not pursue a detailed microarchitectural evaluation in this paper, as this paper focuses on concepts and ideas, and not on bottom-line, configuration-dependent performance numbers.

5.1 Experimental Setup and Hardware Configurations

The experimental setup used is the same as that described in Section 2.2. That is, we use a trace-driven simulator that is based on the MIPS ISA. The conventional hybrid predictor that we use is the same as the one described in Section 2.2. In addition to simulating the *simple confidence estimator* scheme for the conventional hybrid predictor (as described in Section 2.2), we also simulate a *perfect confidence estimator* for the conventional hybrid predictor.

DDISC Configuration Simulated: We implemented a DDISC predictor in our simulation framework. It uses a 32 entry Signature Register File (SRF) as described in Section 4.2. Both fields in an SRF entry are kept 32 bits wide, in line with the 32-bit word size of the MIPS ISA. For the Signature Memory Buffer (SMB), we assume perfect store-load linking. This is because an efficient SMB implementation is heavily dependent on the specific microarchitecture chosen, and will have to be accordingly tuned based on the hardware configurations used. Aliasing effects in the DDISC table depend on the specific implementation of the DDISC algorithm. To provide an unbiased flavor of the fundamental improvement in predictability due to DDISC, and to account for the possible improvements in alias reduction techniques, the measurements are with zero aliasing in the DDISC predictor table.

5.2 Unpredictability Converted

Figure 6 shows the percentage of instructions that were unpredictable in a conventional hybrid predictor, and yet are correctly predicted by the DDISC predictor. The X-axis denotes the benchmarks, and the Y-axis represents the percentage of unpredictable instructions.

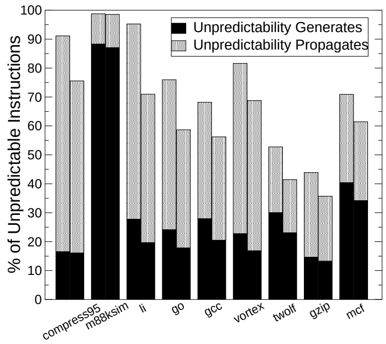

Figure 6: Percentage of Unpredictable Instructions that were Correctly Predicted by the DDISC Predictor

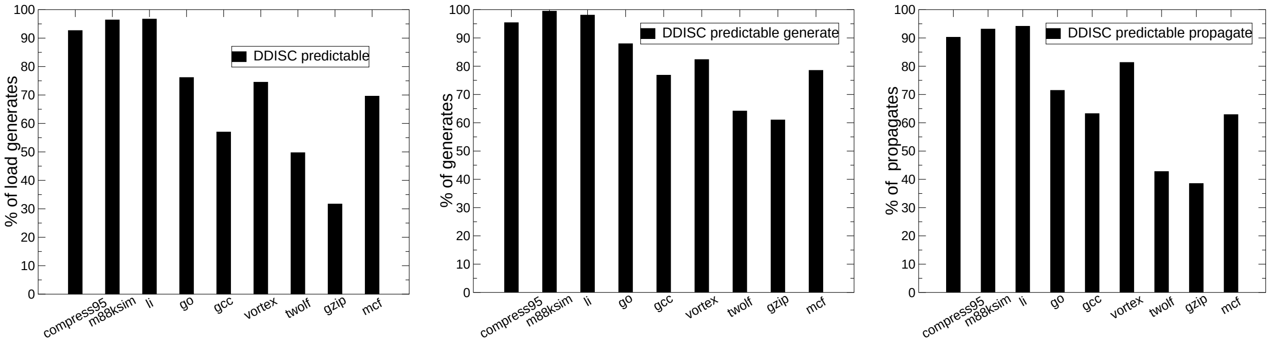

Figure 7: Percentage of {(i) Unpredictable Generate Loads, (ii) Unpredictability Generates, (iii) Unpredictability Propagates} that were Correctly Predicted by the DDISC Predictor with Perfect Confidence Estimator for the Conventional Predictor

Two histogram bars are shown for each benchmark; the first one corresponds to the results with a perfect confidence estimator for the conventional predictor, and the second one corresponds to the results with a simple confidence estimator for the conventional predictor. The height of the bar indicates the percentage of unpredictable instructions that were correctly predicted by the DDISC predictor. Each bar is split into an *unpredictability generates* portion and an *unpredictability propagates* portion. From the figure, we can see that the percentage of unpredictable instructions converted to predictable ones ranges from 42% (for `gzip`) to 99% (for `m88ksim`). The average percentage of conversion is about 70%.

On comparing the two sets of bars in Figure 6, we can see that the decrease in conversion due to the introduction of the simple confidence estimator is not very appreciable. In other words, even with the simple confidence estimator for the conventional predictor, the DDISC predictor is able to convert a major portion of the unpredictability into predictability.

Figure 7(i) shows the percentage of load unpredictability generate instructions that is correctly predicted by the DDISC predictor (with perfect confidence estimator for the conventional predictor), whereas the conventional predictor could not predict them. Figures 7(ii) and (iii) show the percentage of unpredictability generates and propagates, respectively, that were converted into correctly predictable instructions.

5.3 Overall Predictability

Figure 8 shows the overall predictability statistics of the combined {hybrid, DDISC} predictor, with the simple confidence estimator for the conventional predictor. Each benchmark has a single histogram bar, consisting of 5 differently shaded parts. The bottom-most part shows the percentage of instructions predicted correctly by the conventional hybrid predictor. The next part shows the percentage of instructions predicted correctly by the DDISC predictor. The next two bars show the percentage of instructions incorrectly predicted by the hybrid predictor and the DDISC predictor, respectively. The top-most bar shows the percentage of instructions not predicted by the combined {hybrid, DDISC} predictor. Thus, the sum of the bottom-most 2 parts gives the overall percentage of prediction-eligible instructions that were correctly predicted by the combined {hybrid, DDISC} predictor.

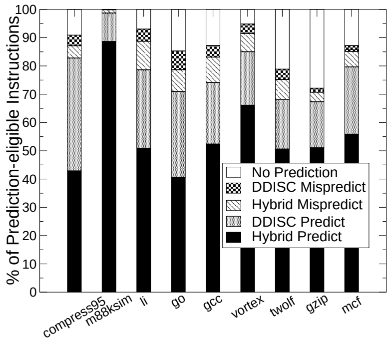

Figure 8: Overall Prediction Statistics of Combined {Hybrid, DDISC} Predictor, when using a simple confidence estimator for the conventional predictor

From the results presented in Figure 8, we can see that the overall predictability of the combined {hybrid, DDISC} predictor ranges from 68% (for `gzip`) to 99.9% (for `m88ksim`). The percentage of mispredictions ranges from 0.1% (for `m88ksim`) to 12% (for `go`). These results are very encouraging. It is important to note that in a real microarchitecture, several instructions may already have their operands available at fetch time, and therefore their results are likely to be available shortly. For those instructions, the signature can be their result value itself, and not the predicted value. This arrangement is likely to improve the performance of the combined {hybrid, DDISC} predictor further.

6 Conclusions

We explored the reasons behind the rather low value predictability in conventional value predictors. Our studies found that the context used by the conventional context predictor does not always encapsulate all of the information required for correct prediction, thereby limiting its predictability. Complex interactions between data flow and control flow change the context in ways that result in predictability loss for a significant number of dynamic instructions. For crossing this predictability barrier, we proposed the concept of using contexts derived from the predictable portions of the data flow graph. That is, the predictability of hard-to-predict instructions can be improved by taking advantage of the predictability of the easy-to-predict instructions that precede it in the data flow graph.

We proposed and investigated a hardware scheme for producing an improved context from the predicted values of previous instructions. Based on this idea, we proposed a novel predictor called *dynamic dataflow-inherited speculative context (DDISC) based predictor* for specifically predicting the hard-to-predict instructions in a conventional hybrid predictor. In order to verify the potential of the DDISC concept, we conducted a set of simulations with an alias-free DDISC table. These experimental results verify that the use of contexts based on dataflow yields significant improvements in prediction accuracies, ranging from 35% to 99% with an alias-free DDISC table. This translates to an overall prediction accuracy of 68% to 99.9%. The DDISC predictor thus enhances the predictability of instructions beyond the predictability barriers of conventional predictors.

Acknowledgements

We thank the anonymous reviewers for their many helpful comments. These comments have contributed significantly to the readability of the paper.

References

[1] B. Calder and G. Reinman, "A Comparative Survey of Load Speculation Architectures," *Journal of Instruction-Level Parallelism 1*, 2000.

[2] F. Gabbay and A. Mendelson, "Using Value Prediction to Increase the Power of Speculative Execution Hardware," *ACM Transactions on Computer Systems*, Vol. 16, No. 3, pp. 234-270, August 1998.

[3] J. González and A. González, "The Potential of Data Value Speculation to Boost ILP," *Proc. ACM International Conference on Supercomputing*, 1998.

[4] S. J. Lee, Y. Wang, and P. C. Yew, "Decoupled Value Prediction in Trace Processors," *Proc. 6th International Symposium on High Performance Computer Architecture (HPCA-6)*, 2000.

[5] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proc. 29th International Symposium on Microarchitecture (MICRO-29)*, pp. 226-237, 1996.

[6] A. Moshovos and G.S. Sohi, "Streamlining Interoperation Memory Communication via data Dependence Prediction," *Proc. 30th International Symposium on Microarchitecture (MICRO-30)*, 1997.

[7] T. Nakra, R. Gupta, and M. L. Soffa, "Global Context-based Value Prediction," *Proc. 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, pp. 4-12, 1999.

[8] Y. Sazeides and J. E. Smith, "Modeling Program Predictability," *Proc. 25th Annual International Symposium on Computer Architecture*, 1998.

[9] G. S. Tyson and T. M. Austin, "Improving the Accuracy and Performance of Memory Communication Through Renaming," *Proc. 30th International Symposium on Microarchitecture (MICRO-30)*, 1997.

[10] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, "Dynamic Predictions of Critical Path Instructions," *Proc. 7th International Symposium on High Performance Computer Architecture*, 2001.

[11] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," *Proc. 30th International Symposium on Microarchitecture (MICRO-30)*, pp. 281-290, 1997.