Dependence Based Value Prediction

Yiannakis Sazeides

Department of Computer Science

University of Cyprus

{yanos}@cs.ucy.ac.cy

Abstract

This paper introduces **dependence-based value prediction***: prediction based on information that can be propagated through dependences. We propose an organization for a dependence-based value predictor and investigate how to use different types of dependence information to predict values produced by instructions. We consider first register dependences and then memory dependences. Memory dependence prediction is employed for propagating the history used for predicting output values.*

When considering only register dependences average prediction of 75% is achieved with a 64K entry table. The use of memory dependence prediction increases average accuracy for a 64K entry table to 82%. Several benchmarks achieve value prediction accuracy over 90%. We provide evidence that the proposed predictor requires small amount of speculative state (for speculative updates) which may suggest its design feasibility.

1 Introduction

Performance has been a driving force for microarchitecture research since the advent of the computer. One of the popular approaches employed to improve performance is the application of software and hardware transformations to a program to increase its Instruction Level Parallelism (ILP). Fundamentally, true dependences limit the amount of ILP that can be extracted from a program. Two instructions exhibit a true dependence, or are simply dependent, when the output of one of the instructions is an input operand of the other.

Dependences are typically divided into control and data dependences. Prediction and speculation [1] have been proposed as means for alleviating the impediments of control dependences by predicting the next PC of control transfer instructions and speculatively executing instructions that follow them. In addition to control dependences through the PC, there is also serialization of execution due to dependences through registers and memory locations. These dependences were shown recently to be amenable to prediction and speculation[2, 3].

The levels of predictability for branches and memory dependences are quite high usually over 95%. However, the predictability of register values produced by instructions is significantly lower even under ideal assumptions[4]. Higher predictability is desired because usually it translates to bigger performance returns. Consequently methods that can increase value predictability may hold significant performance potential.

In this paper we introduce a dynamic prediction method, dependence-based value prediction, and investigate how to use dependence information to predict output values. The hypothesis behind dependence-based value prediction is based on the following:

- the predictability of an instruction is determined by information on the dependence path that leads to it, and
- dependence path information and recent values on the dependence path of an instruction are good predictors of the future value behavior of the instruction.

The above are based on the postulate made in [5] that the predictability of an instruction is determined by the "generates" that influence it. [5] introduced the notion that predictability is generated, propagated and terminated by instructions and dependences during the execution of a program. In that work it was suggested that the use of dependence information holds potential for improving the prediction accuracy of value predictors.

The proposed predictor has several novel features: predicts values based on information propagated through register and memory dependences, and uses memory dependence prediction for communicating the history used for value prediction. These features enable high value prediction accuracy over 90% for several benchmarks with a 64K entry prediction table. Another important property of this predictor is that on a branch misprediction its speculative state (due to speculative updates[6, 7]) can be undone fast because it can be managed as a renamed register file - this was not the case with previously proposed history predictors that require the undoing of speculative state in large tables[8, 9].

1.1 Outline

The main concepts of dependence-based value prediction are introduced in Section 2. First, we describe prediction using register dependence information and then extend it to memory dependence information. A qualitative comparison between context-based and dependence-based value prediction concludes Section 2. The experimental framework is described in Section 3. The results are presented and discussed in Section 4.

Related work is discussed in Section 5. We conclude in Section 6.

2 Dependence-Based Value Prediction (DBVP)

The process of dependence-based value prediction can be divided into two phases: the construction of a dependence record, and the use of the dependence record to access a predictor to obtain a prediction. We define what is a dependence record and then we introduce a dependence-based value predictor. We describe first a predictor that considers information propagated only through register dependences and then extend it to dependence information through memory. We conclude this section with a comparison of Dependence-Based and Context-Based value prediction.

2.1 Dependence Record

To predict an instruction with dependence-based value prediction its *Dependence Record (DR)* is needed. A DR is derived from a dynamic sequence of instructions called *instruction region (IR)* (we define them later). The DR consists of:

- *Generate Registers (GR)*: registers that (a) are liveins to the IR, and (b) belong to the data dependence path that reach the instruction.
- *Dependence Path Id (DPI)*: information about the instructions on the dependence path that emanate from generate registers and reach the predicted instruction (the information can be pcs, optypes immediates, branch history etc). We refer to the dependence path of a predicted instruction that emanates from the generates as its *Prediction Dependence Path (PDP)*.

Effectively the DR of an instruction is a summary about its PDP. GR represent information about the earliest predecessor registers with known values whereas the DPI information about the intervening predecessors.

The process of constructing the DR of an instruction is as follows follows: given an IR, each instruction, in dynamic order, reads the DR of its inputs registers and combines them together with information about itself to form a new DR. It then propagates the new DR through its output register to its successors. The DR of a register that is a generate, is the register itself. It is noteworthy that the DR is derived solely on information that is encoded in the instruction (no need for actual register values). It is also important to note that prediction

	OpCode	Operands	GR	DPI	Output Values
1	srl	$2,$6,5	6	pc1	$(0)^{32}(1)^{32}$
2	sll	$2,$2,2	6	pc2	$(0)^{32}(4)^{32}$
3	addu	$2,$2,$19	6, 19	pc3	$(0x1002f8b0)^{32}(0x1002f8b4)^{32}$
4	lw	$2,($2)	6, 19	pc4	$(0x8000bfff)^{32}(0xffffffff)^{32}$
5	andi	$3,$6,31	6	pc5	$(0,1,..,31)^{2}$
6	srlv	$2,$2,$3	6, 19	pc6	$v_0,v_1,..,v_{62},v_{31},$
7	andi	$2,$2,1	6, 19	pc7	$(1)^{14}$ 0 1 $(0)^{15}(1)^{33}$
8	beq	$2,0,...	6, 19	pc8	
9	addiu	$6,$6,1	6	pc9	1,2,..,64
10	slti	$2,$6,64	6	pc10	$(1)^{63}0$
11	bne	$2,0,...	6	pc11	

Figure 1: Example Dynamic Sequence from *126.gcc*

need not be preceded always by DR construction. The DR construction can be performed at a different time and be stored in another structure where it can be retrieved prior to prediction. The implementation specifics of DR construction are not investigated in this paper but we assume its functionality.

To illustrate the process of DR construction consider the frequently executed dynamic instruction sequence from the SPECINT95 benchmark *126.gcc* shown in Fig. 1. The column labeled output values shows the sequence of values produced by each instruction each time the function they belong is called. Lets assume the above is an IR. The liveins to the IR are registers $6 and $19. Column labeled GR shows the generate registers that reach each instruction. For example, instruction 1 has GR register $6. This information is propagated to instruction 2 through the output register $2. Meaning in this case that instruction 2 has also GR register $6. Instruction 3 has two GR, register $6 through the dependence of register 2 and register $19 because its a livein. The rest of the GRs can be obtained similarly. The process for determining the DPI for each instruction is identical to the GR formation, with only difference the type of information that is propagated. We show the DPI for each instruction when it contains its PC (this is a case where no DPI information is propagated to the successors).

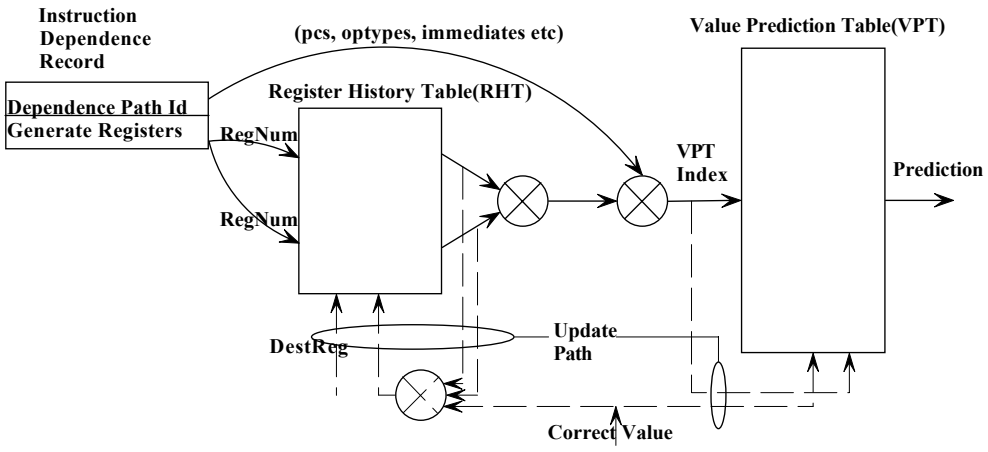

Figure 2: Dependence-Based Value Predictor with Register History

Once the DR of an instruction is available it can be used for predicting the output value of the instruction (described in the next section).

It should be noted that load instructions propagate the dependence information coming through their address operands. In the above example the GR for the load, instruction 4, are registers $6 and $19 which are on the address dependence path of the load. Though this information may be useful for predicting the output value of a load, it ignores the possible (memory) dependence of the load to a store. This is a potential source of non-determinism that can degrade prediction accuracy - we will revisit the issue of memory dependences in Section 2.3.

In the next section we introduce an organization for a DBVP that predicts based on DR information. It considers only propagation of DR through register dependences. In Section 2.3 we extend the predictor to propagate dependence information through memory dependences.

2.2 DBVP with Propagation through Register Dependences

In Fig. 2 we present a generic organization for a DBVP predictor. It consists of two tables the register history table (RHT) and the value prediction table (VPT). To obtain a prediction for an instruction: (1) the RHT is indexed with the instruction's GR, (2) the obtained RHT history (we describe below what it is) is combined with the DPI to form an index (vpt index), and (3) the prediction table is accessed to obtain a prediction.

A predicted instruction updates the RHT entry corresponding to its **destination** register with a combination of the RHT history read during the prediction and the new value produced by the instruction. Note that the RHT

entries read and written by an instruction can be different - effectively the register history is also propagated through the dependences. The VPT gets updated with the new value produced by the instruction at the location that was accessed to obtain the prediction.

The purpose of the two **complementary** types of DR information is evident now: GR are used to obtain values that are indicative of what will be the next value of one or more instructions whereas the DPI is used to differentiate the predictions between instructions with the same GRs.

In Fig. 1 several instructions have the same GR (for example instructions 1,2,5,9,10 and 11 have register $6). If we were predicting in parallel all these instructions without DPI information they would have accessed the same VPT location and consequently mispredicted. The use of DPI enables instructions with same GR to get predictions from different locations. The use of DPI information without the GR is also problematic but for a different reason: instructions with the same GR can go to different locations, however, they will go to the same location each time, predicting always the next value to be the same as the last.

It is important to note that the above predictor resembles a Context-Based (CB) predictor introduced in [4]. The two predictors, however, have significant differences in terms of the information that is used to access the tables. The implications of the similarities and differences of the two predictors are discussed in Section 2.4.

Orthogonal issues regarding memory structures and predictors are relevant here also - (associativity, replacement policy, hash functions for combining information etc). Although these issues represent important directions of research we do not address them in this paper but leave them for future work.

2.3 DBVP with Propagation through Memory Dependences

DBVP so far ignores the effects of memory dependences because it only propagates dependence information through register dependences. In particular, when a load and its dependent instructions are predicted, the predictions are based on the dependence information of the registers used to compute the address for the load. This may be sometimes an accurate indication of the value behavior of a load, however, there are cases where such correlation does not exist - after all, addresses are mainly side effects of communication through memory. And this leads naturally to misprediction of the load and its dependent instructions.

Consider the following IR that belongs to a loop where the induction variable was spilled to memory. The column labeled output values shows the values produced by each instruction when the loop executes.

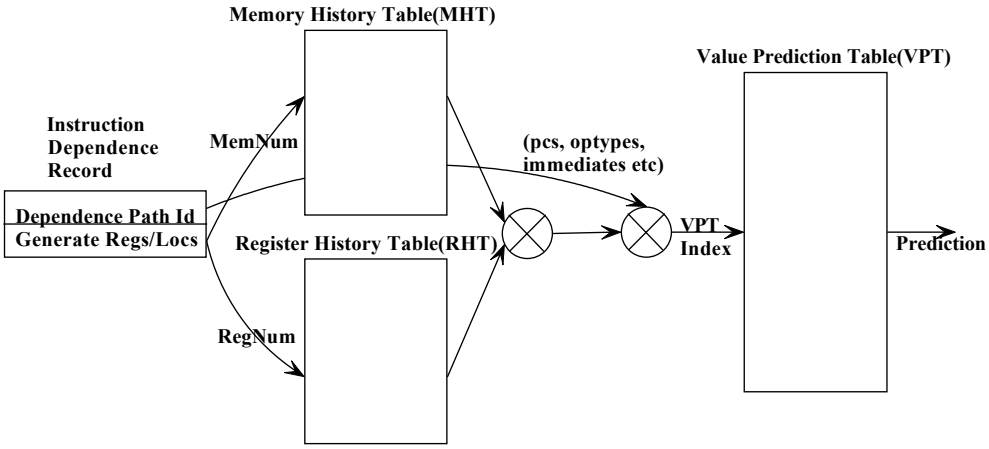

Figure 3: Dependence-Based Value Predictor with Register and Memory History

	OpType	Operands	GR	Output Values
1	ld	$2,($28)	28	0,1,2,3,..,63
2	add	$2, $2, 1	28	1,2,3,4,..,64
3	st	$2, ($28)	28	
4	slti	$2,$2,64	28	$(0)^{63}$,1
5	beq	$2,$0, LL1	28	

Assume the above sequence is going to be predicted and the value of register history corresponding to register $28 is known (and is constant for all executions of the sequence).

Although the values produced by instructions 1, 2 and 4 are repeating, the DBVP predictor will be unable to predict them correctly. This is the case because the value of their GR ($28) is constant and therefore the index formed to access the VPT will be constant. This is an undesirable situation since *different output values of an instruction have the same generate value*.

This problem can be resolved by:

- propagating DR through memory dependences, and
- converting load dependence values of an IR to liveins.

DR propagation through memory should be employed for store-load dependences in the same IR. These would convert def-store-load-use dependence chains in an IR to a def-use chain. For loads with dependences to stores outside the IR, the stored value should become a livein of the IR. These requirements are analogous to the ones for register dependences.

Assuming the above are feasible (we will argue for this at the end of this section) then instructions that depend on loads can be predicted as described next.

The DR construction should be expanded to provide Generate Locations (GL) in addition to Generate Registers. A GL is an index into a table (Memory History Table (MHT)) that contains recent values written to memory (see modified predictor in Fig. 3). The method by which stored values are written/read to/from the Memory History Table is described at the end of this section.

Load instructions should be predicted as to whether they have a memory dependence. If they do they are not value predicted and during DR construction (a) if it is predicted that their dependence value will be found in the MHT, then the corresponding MHT location is propagated to the load dependent instructions, (b) else the DR of the instruction producing the value should be propagated to the dependent instructions.

When a load is predicted to not have a dependence on a dynamic store (for reasons such as static data, i/o, finite MHT size etc) then its output value is predicted based on the dependence information of the address operands (as described in Section 2.2).

The above discussion for DR construction using dependence information through memory assumed the following mechanisms : a) memory dependence prediction - that is predicting whether a load depends on a previous store, b) direct (not through regular memory hierarchy) communication of values from producing store to dependent load c) communication of DR through def-store-load-use dependence chains.

We believe that previously proposed mechanisms can provide the above functionality [3, 10, 11, 12]. For dependence prediction mechanisms we can employ the mechanisms proposed by [3, 12]. For direct communication, the MHT used in the predictor can be managed as a transient value cache (TVC)[3] or value file[11]. The TVC maintains recently store values and loads predicted to have a dependence in the TVC can receive their value directly form there. The communication through def-store-load-use dependence chains can be achieved using speculative memory bypassing[3]. Speculative memory bypassing converts def-store-load-use to def-use.

It is important to point again that DR construction need not be performed just before predicting an instruction, in one pass. For example, it can be constructed incrementally similar to store sets[12]. We do not investigate the details for DR construction but we believe that slightly modified/enhanced schemes for memory dependence prediction can be used to provide the desired functionality.

2.4 Dependence-Based vs Context-Based Prediction Value Prediction

The basic organization and principle of operation of DBVP is identical to a CB predictor [4]. Both predictors (a) consist of two-level tables, and (b) learn past behavior by associating output values with a given context, when the same context appears the value associated with the context is predicted to be the next output. However, important differences exist between the two predictors:

- first level table is accessed with different information, and
- the context information is different

The first level table of DBVP is indexed with register numbers whereas in a context-based predictor the PC is used. This represents a significant reduction in the size of the first level table (without compromises in performance as we will see next). In a previous work was shown that CB predictors required accurate local history per PC (i.e. large first level table with $\geq$4K entries [13, 8]) to achieve high prediction accuracy. With DBVP a 64 entry RHT will be sufficient for most current processors.

Another important implication of the smaller size first level table [4, 8, 9] is on the difficulty level of implementing recovery from speculative updates. To clarify, in a realistic environment that employs value prediction it should be expected that some delay will be incurred between prediction and updating. Consequently, is possible for an instance of an instruction to be predicted before a previous instance of it has updated the tables (delayed-update). If delayed-updates are frequent, then lower prediction may be expected for history based predictors since multiple predictions may be performed using the same history (unlike the case where predictors are updated immediately).

Assuming that delayed updates cause a performance degradation, one way to deal wth them is to speculatively update the predictor entries. This is aimed to advance the contents of a history entry as if the updated value was available immediately after the prediction was made. For the CB and DBVP the type of information that needs to be speculatively updated, is the first level table entry. For a context-based predictor the location corresponding to the PC of the predicted instruction is updated with the predicted value. For DBVP the location in the RHT corresponding to the destination register of the predicted instruction is updated with the predicted value.

Speculative updating is complicated by branch misprediction due to updates performed by instructions on the wrong path of execution. It was shown in [8] that if the history state of a CB is not recovered prior to the

offending branch on a branch mispredict then prediction accuracy decreases dramatically. All studies using context-based prediction until know assumed the ability to recover fast large first level table state. But none was able to demonstrate any realistic scheme that can achieve it[8, 9].

With DBVP recovery of first level table state appears to be feasible. This can be accomplished by managing the RHT as a renamed register file (recall that the RHT is indexed with register names).

A potential drawback of the above scheme may be that the GR will need to be translated and hence the prediction may come later in the pipeline. We do not explore how critical this is on performance or possible ways around it, however we recognize its importance as an important item for future work.

The second important difference between DBVP and CB, is that the context for a DBVP is the combination of DR information and register dependences value history. In contrast, the CB predictor relies on the local history of a PC. The DBVP history information provides the benefits of longer local history with a different (hopefully simpler) approach.

Consider the example in Fig. 1. All instructions produce exactly the same sequence every time the function they belong is called. Context-based prediction is suited for these cases since it is theoretically capable of learning and predicting any repeating sequence. However, in the above example each of the instructions 1,2,3,4,5,7 and 10 will incur one or more mispredictions every time the loop is executed if history with less than 15 previous value is used. *Order* is the parameter that describes the number of previous values used to determine what is going to be the next value. No mispredictions (following learning) will occur if the predictor had order 63. Long order can be problematic for two reasons — learning may take longer and in a practical scheme with finite sized tables may lead to destructive aliasing.

This demonstrates that context-based prediction is sensitive to the order and may cause repeating sequences not to be learned and consequently mispredicted. With dependence-based value prediction the above can be resolved, because we do not rely on local history but on dependence information and predecessor values. In our example, the use of $6 as GR enables 100% accuracy, following learning, since each iteration of the loop will be associated with a distinct GR value.

2.5 Instruction Regions

In the discussion so far we did not describe why IR regions are needed, how are formed, how big they can be etc. IR regions are needed to express the granularity with which instructions are processed by a given execution paradigm. The size of an IR may have important ramifications on prediction accuracy since the bigger the granularity the older are the generate values that will be used to predict the instructions in an IR.

The formation of an IR is the process by which the dynamic instruction stream is divided into IR. This is a critical issue because if an instruction belongs to many IR regions then (a) the predictor training may take significantly longer, and (b) with a finite prediction table this can result in destructive aliasing. Also if an IR is too big then the likelyhood of predicting multiple instances of the same instruction increases. We are currently exploring several heuristics for partitioning the dynamic execution of a program to IR that facilitate higher prediction accuracy and performance.

In the experimental evaluation we will examine only one partitioning scheme that is described in the next Section 3.

3 Experimental Framework

To evaluate the effectiveness of the proposed predictor and explore the importance of its different parameters we performed a simulation study. The simulator is based on the simplescalar toolset[14]. The simulation did not consider microarchitectural parameters because we wanted to avoid having implementation idiosyncracies influencing our observations. Thus the metric used to compare performance is prediction accuracy.

Simulations were performed for the integer SPEC95 benchmarks shown in Table 1. The benchmarks were compiled using the gcc compiler provided with the toolset using -O3 optimization.

We performed experiments for several configurations. Configurations that use only register dependence information are denoted as DBVP-R whereas those that use both register and memory information are denoted by DBVP-M. For all configurations direct mapped tables were used. A DR could contain as many generates (GR and GL) as needed. The values read from the history tables were folded to as many bits as the $\log_2$(size the VPT table) by xoring them together. To form an index into the VPT the hashed generate values and the DPI information are xored together.

We explored several types of DPI for predicting an instruction's: (a) optype, (b) PC, and (c) immediate

Benchmark	Input Flags	Dynamic Instr. (mil)	Instructions Predicted (mil)
compress	30000 e	8.2	5.8 (71%)
gcc	-O3 gcc.i	178	120 (68%)
go	9 9	132	105 (80%)
ijpeg	specmun.ppm	129	108 (84%)
vortex	mod train	100	63 (63%)
perl	scrabbl.in	40	26 (65%)
xlisp	7 queens	202	125 (62%)

Table 1: Benchmarks Characteristics

values. The eight combinations shown in Table 2 were explored (the first column shows the code denoting each combination).

We assumed a perfect dependence record construction (we expect this to be close to a realistic performance since memory dependence prediction schemes achieve near optimal performance). The only constrain we introduced was limited MHT size: we consider the following MHT sizes 0, 1, 4, 16, 64, 256, 1K and ALL. We assumed an MHT managed as fifo. An MHT table size of 0 means that memory dependences were not considered.

To gain an insight about the effects of VPT table size, we measured the performance for the following table sizes: 4k, 16k, 64k, 256k and 1M entries. The default size is 64k entries. The replacement in the vpt table was guided by a 1 bit counter.

Instruction regions used in the experimentation were *fixed length instruction sequences*. A sequence length was reinitialized whenever a value was mispredicted. This may not represent a good heuristic because it can lead to many distinct IR regions and as a result to slower learning. However, proved adequate to get sufficient evidence for the prediction accuracy potential of DBVP.

We consider sequence lengths of 1, 2, 4, 8, 16, 32 and 64 instructions. Instructions in a region are predicted in parallel, and then update their prediction tables. The default IR length is 1 - the case which the predictor is updated following each prediction. For delay 1 the generate values of an instruction are its inputs values.

To compare the performance of DBVP and context-based prediction we measured prediction accuracies

Benchmark	Input OpType	Dynamic PC	Instructions Immediate
H0	0	0	0
H1	0	0	1
H2	0	1	0
H3	0	1	1
H4	1	0	0
H5	1	0	1
H6	1	1	0
H7	1	1	1

Table 2: DPI Configuration

with a 64k entry CB predictor configuration as described in [8]. This predictor uses a 64k entry local history (first level table) and a 64k entry second level table. The second level table can be shared among all instructions and is accessed using the last four values produced by an instruction in hashed form (this configuration is denoted as CB.64K). We also considered the performance of a third-order CB predictor with unbounded first level table and unbounded second level table per instruction (denoted CB.inf.local)(see [4] for more details about this configuration).

In some of the experiments 3-tuples and 2-tuples are used to identify a configuration as follows **<IR Length><MHT Size><VPT size>** and **<MHT size><VPT size>**. For averaging prediction accuracies we used the arithmetic mean.

The paper introduces the concept of dependence-based value prediction, it makes, however, a small effort towards optimizing its performance. Therefore, the reported results should be viewed as bounds of the assumptions and methodology employed and not representative of the overall potential of the proposed scheme.

4 Results

4.1 Register Dependences

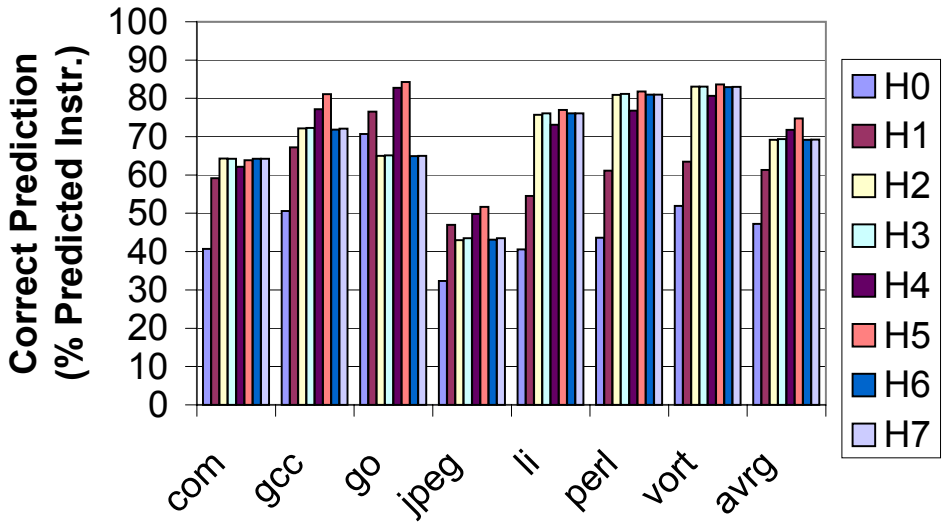

Figure 4: DBVP-R for Different DPI Configurations

Fig. 4 shows the prediction accuracy of a DPVB that considers only dependences through registers. For each benchmark the prediction is shown for eight different DPI configurations.

The objective here is to find a good combination of dependence path information for predicting values. The data show that the use of DPI is necessary for high prediction accuracy. The DPI H0, that is predict using generate values but no DPI information is the worst performing across all benchmarks. This indicates that often different types of instructions have the same input value.

The best performing DPI configuration is the one that includes the optype and the immediate value of the predicted instruction (H5). Note that the accuracy of H5 is always higher than its individual components (H1 or H4). The data show that using only the immediate value (H1) in the DPI is better than none (H0). Its performance, however, is the second lowest across all benchmarks because it does not provide any information to distinguish the value behavior of different types instructions - this low accuracy should be expected because there are many instructions with no immediate inputs or even with the same immediate inputs. Prediction accuracy goes higher (the second best overall) when only the OpType is used (H4). The optype is more effective in distinguishing the predictions between instructions with same inputs but with different type. It can also create constructive aliasing since different instructions of the same type and value behavior can learn from each other.

Finally the data show that when comparing configurations that use the PC with configurations that use the optype but not the PC (H4 and H5), the former provide little or no advantage and in some cases decrease

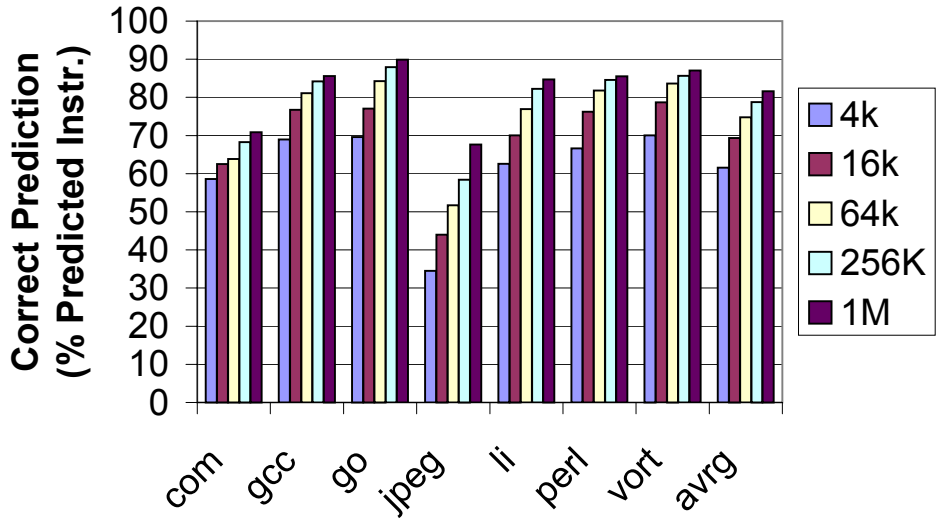

Figure 5: DBVP-R as a function of VPT size

prediction accuracy (see for example go in Fig. 4). This may be caused by a combination of the following: (a) finite table size, because the use of the PC causes a lot of destructive aliasing, and (b) slower learning, since each instruction needs to learn its own behavior and can not exploit constructive aliasing due to optypes. It is noteworthy that all DPI configurations that use the PC have very similar accuracy, this the case because the PC is a uniquely identifying each instruction.

Based on the above results the configuration H5 was used for the remaining experiments.

Fig. 5 shows the performance of the DBVP for different size value prediction tables. The data suggest that for all benchmarks bigger tables mean higher accuracy. However, with increasing table size the benefits diminish. The exception to this is jpeg that appears to have room for higher accuracy with bigger tables. These results show that one important problem with DBVP prediction is aliasing – it should be interesting to establish whether capacity or/and conflict misses are the causes of this behavior and come up with ways to reduce their effects.

Benchmark behavior can be divided in to two groups *gcc, go, li, perl* and *vortex* appear to have higher predictability than *compress and jpeg*. The distinct features of *compress* and *jpeg* is simple control flow structure and execution driven by their input data. The benchmarks results seem to suggest that high levels of prediction accuracy may not be feasible for all programs.

Fig. 6 shows the prediction accuracy as a function of the IR size. The data show that bigger regions mean lower accuracy. However, the degradation becomes insignificant as distance increases. This is the case because

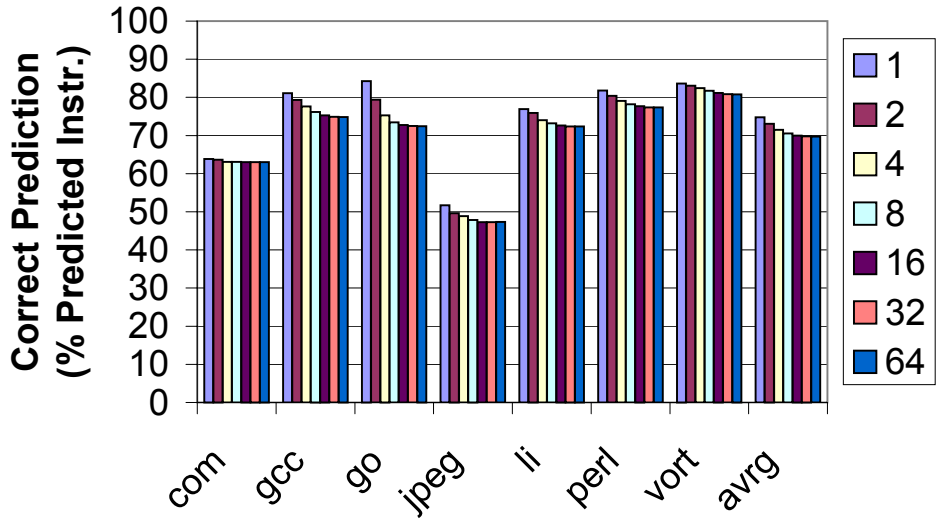

Figure 6: DBVP-R Accuracy with various IR sizes

with increasing distance is more likely for a misprediction to occur.

The overall degradation with increasing IR is not significant, however its presence suggests that either (a) the hash function used with immediate updates is not as effective with longer IR, or (2) the IR partition heuristic used, introduced destructive aliasing in the table. We investigated this further by inspection and found that fixed length IR often forces the same instruction to belong to different IR. As a result there is more destructive aliasing in the prediction table. This underlines the importance of considering the value prediction behavior when partitioning a program in IR regions in a way that this redundancy is reduced.

4.2 Memory Dependences

In this section we investigate how propagation of dependence information through memory affects prediction accuracy. Fig. 7 shows for each benchmark the prediction rate as a function of increasing MHT table size.

The results indicate that propagation through memory dependence can increase prediction rate significantly (compare MHT 0 with MHT 256). The data show that with increasing RHT size prediction accuracy increases, however, with diminishing returns. The data show that the performance of a small RHT has comparable performance to the optimal scheme where a load can detect its dependence to any previous store. This suggests that a small RHT (256 entries) can be sufficient.

It can be observed that the prediction increases are not uniform across benchmarks. Increases are significant

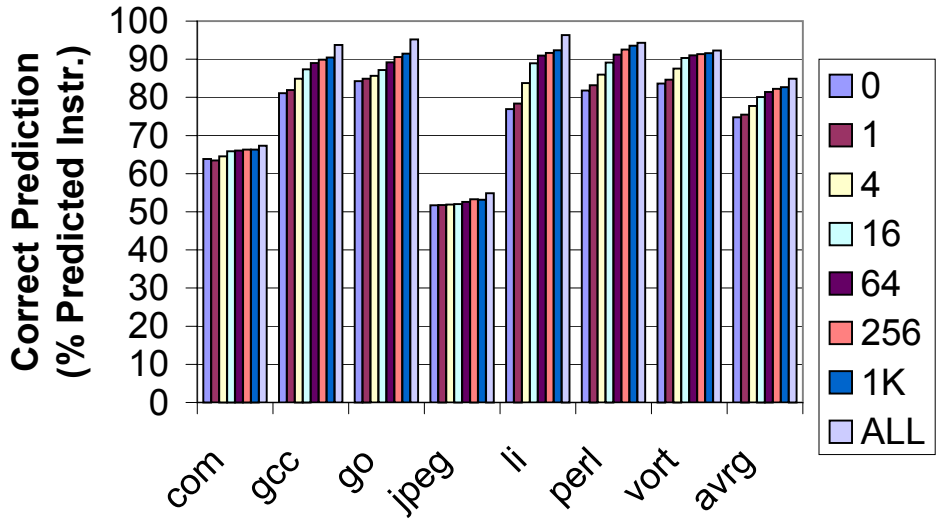

Figure 7: DBVP-M as a function of RHT size

for *gcc*, *go*, *li*, *perl* and *vort* but for *compress* and *jpeg* are small. These appears to support the earlier observation that *compress* and *jpeg* value behavior is determined directly from their input data which does not appear to be very predictable.

Fig. 8 shows the effects of IR size on the accuracy of DBVP with propagation through memory. We can observe a small degradation (on the average 4%) with update distance of 16. The reasons for the degradation are similar to the ones for DBVP-R.

4.3 Context Based vs Dependence-Based Value Prediction

In this section we compare the performance of the proposed predictor against a context-based predictor (see Fig. 9). CB predictor results are for two configurations with immediate updates: the one for a finite and the other for an unbounded predictor. For DBVP the results are shown with IR length 1 (immediate updates) and with IR length 16, and without/with dependence propagation through memory. It is important to note that the size of the CB predictor's first level table is 64K entries whereas for the DBVP the RHT is only 67 (32 integer, 33 floating point, and 2 multiplication registers).

The results show that all DBVP configurations have superior performance over the finite CB predictor. For several benchmarks the configuration DBVP.1.256.64K can provide accuracy over 90%. The most pronounced improvements are in the case of benchmark *go* where depending on the configuration the improvements range

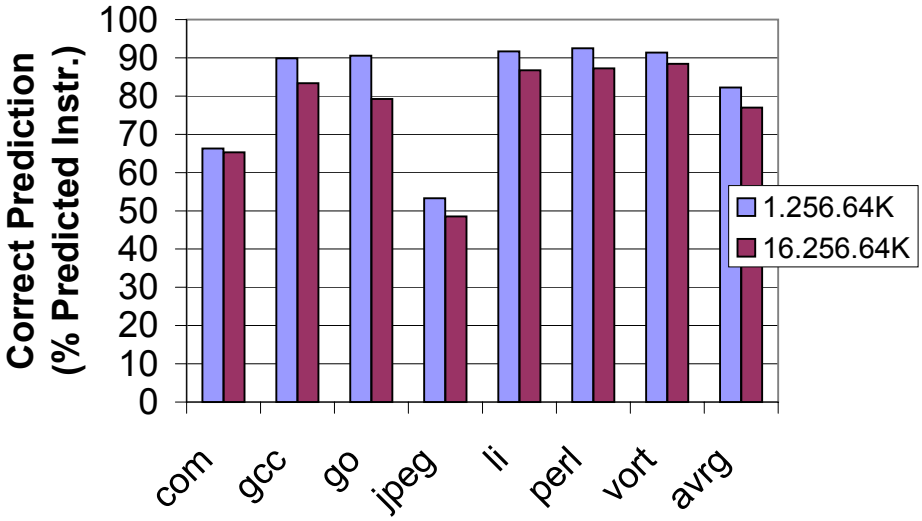

Figure 8: DBVP-M Accuracy for IR sizes 1 and 16

from 20% to 50%. This is an indication of the potential of the new predictor.

When comparing DBVP to the unbounded table we can observe that DBVP with propagation through memory dependences has, in most cases, comparable or better prediction accuracy. The exception is jpeg. Further investigation, revealed that *jpeg* is repetitive but the distance between repetition is very long that capacity misses lead to misprediction. Note that in Fig. 5 with increasing table size the performance of *jpeg* went up.

5 Related Work

Several value predictors have been proposed recently[2, 15, 16, 17, 4, 18, 8, 19]. All of them rely on the local history of an instruction to predict the next value. An exception to the above is the storageless value predictor [20] where the allocation of registers is done in such a way so that instructions are allocated to registers that contain their predicted output value.

Dependence information is used in [21] for constructing, dynamically, representations of linked data structures. The representations are used to drive a seperate prefetch engine for getting data into the cache.

Dependence information is also used in [22] for constructing, manually, expressions that lead to difficult to predict branches. When a difficult to predict branch is fetched into a processor, the values of the input registers of its expression are predicted using a value predictor (any value predictor can do) and the expressions are

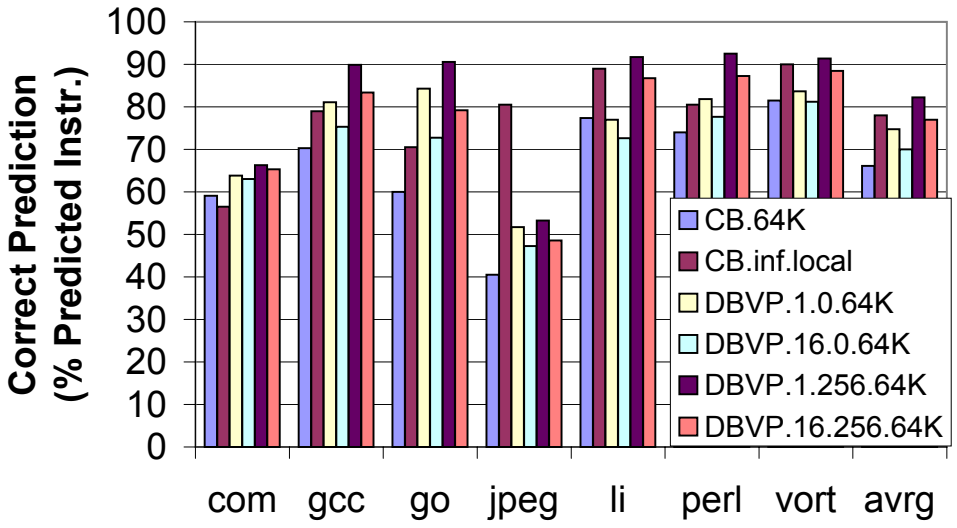

Figure 9: DBVP vs CB prediction

executed. This allows the "branch" flow to move ahead of the normal execution enabling the detection of a misprecicted branch faster.

The proposed predictor is fundamentally different from previous work because it relies on information propagated through dependences for predicting values. Another important characteristic of the proposed predictor is that it uses memory dependence prediction for communicating from a store to the consumers of a load. This is borrowed from work done for memory dependence prediction [17, 3, 10, 12]. However, the way the propagated information is used is new. More specifically, the communicated information is used to predict the output of the consumers of a load and their successors.

In [5] a predictability model was proposed to investigate the sources of predictable behavior. Some of the conclusions in [5] served as starting points for this paper. In particular the observation that the predictability of an instruction is determined by its predecessors, was one of the hypothesis in this paper.

6 Conclusions and Future Work

In this work we introduced dependence-based value prediction. This is prediction based on dependence information propagated through registers and memory dependences. The proposed predictor uses memory dependence prediction for propagating history information from a store to a load's consumer instructions. The history information is used for predicting the output of the consumer instructions.

We provided evidence suggesting that the proposed predictor may have better implementation properties than previously proposed context-based predictors because it requires small first level table that can be managed as a renamed register file. This can be used for fast recovery from speculative updates down the wrong execution path.

Experimental results showed that DBVP can predict with high accuracy. The best performance is obtained when both register and memory dependences are considered. For several benchmarks accuracy over 90% is possible with a 64K entry prediction table.

A comparison of the DBVP with CB predictor showed that the proposed predictor, for similar value prediction table cost, can achieve always better performance. Finally, we found in some cases that a DBVP with a 64K entry vpt could outperform unbounded CB predictors.

Some of the issues that need further investigation are: (a) dependence record construction (b) the formation of instruction regions, (c) how to best use dependence information for predicting, and (d) performance evaluation of the proposed scheme for a given microarchitecture.

References

[1] J. E. Smith, "A Study of Branch Prediction Strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[2] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Data Speculation," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138–147, October 1996.

[3] A. Moshovos, S. E. Breach, T. J. Vijaykumar, and G. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 181–193, June 1997.

[4] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 248–258, December 1997.

[5] Y. Sazeides and J. E. Smith, "Modeling Program Predictability," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 73–84, June 1998.

[6] T.-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," in *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 124–134, May 1992.

[7] S. Jourdan, T.-H. Hsing, J. Stark, and Y. N. Patt, "The Effects of Mispredicted-Path Execution on Branch Prediction Structures," in *PACT'96*, October 1996.

[8] Y. Sazeides, "An Analysis of Value Predictability and its Application to a Superscalar Processor," *PhD Thesis, University of Wisconsin-Madison*, 1999.

[9] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated load-address predictors," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[10] A. Moshovos and G. Sohi, "Streamlining Inter-operation Memory Communication via Data Dependence Prediction," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[11] G. Tyson and T. Austin, "Improving the Accuracy and Performance of Memory Communication through Renaming," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[12] G. Z. Chrysos and J. S. Emer, "Memory Dependence Prediction using Store Sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 142–153, June 1998.

[13] Y. Sazeides and J. E. Smith, "Implementations of Context–Based Value Predictors," Tech. Rep. ECE-TR-97-8, University of Wisconsin-Madison, Dec. 1997.

[14] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi, "Streamlining Data Access with Fast Address Calculation," in *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 369–380, june 1995.

[15] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit via Value Prediction," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 226–237, December 1996.

[16] F. Gabbay and A. Mendelson, "Speculative Execution Based on Value Prediction," Tech. Rep. (Available from http://www-ee.technion.ac.il/fredg), Technion, November 1996.

[17] M. H. Lipasti, "Value Locality and Speculative Execution," Tech. Rep. CMU-CSC-97-4, Carnegie Mellon University, May 1997.

[18] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 281–290, December 1997.

[19] B. Calder, G. Reinman, and D. Tullsen, "Selective value prediction," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[20] D. M. Tullsen and J. S. Seng, "Storageless value prediction using prior register values," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[21] A. R. A. Moshovos and G. Sohi, "Dependence Based Prefetching for Linked Data Strctures," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 115–126, December 1999.

[22] A. Farcy, O. Temam, R. Espasa, and T. Juan, "Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 59–68, December 1998.