Using Hyperprediction to Compensate for Delayed Updates in Value Predictors

Qing Zhao, Sang-Jeong Lee[♣] and David J. Lilja[*]

Dept. of Computer Science and Engineering, Univ. of Minnesota, Minneapolis, MN 55455

[♣]Dept. of Computer Science and Engineering, Soonchunhyang Univ., Chungnam, Korea

[*]Dept. of Electrical and Computer Engineering, Univ. of Minnesota, Minneapolis, MN 55455
zhao@cs.umn.edu, [♣]sjlee@sch.ac.kr, [*]lilja@ece.umn.edu

Abstract

Value prediction has been proposed as a technique to break true data dependences in order to increase the instruction-level parallelism available in programs. Recent work has pointed out, however, that the delay inherent in updating the value prediction table with the actual correct value can introduce a substantial number of wrong value predictions, which then can decrease the overall processor performance. We propose and systematically study a technique that we call *hyperprediction* to compensate for the delay in updating the value prediction table. This approach accurately computes and records the number of *outstanding instances* of an instruction, which is the number of times an instruction has requested a value to be predicted since the last time the corresponding entry in the value prediction table was updated. With this information, the value predictor can provide reliable predictions for the currently requesting instance of an instruction based on both the currently stored value and the number of outstanding instances. We show how the hyperprediction technique can be implemented in a stride value predictor, a context-based predictor, and a hybrid predictor. Our simulations using an extension to the SimpleScalar tool set and integer and floating-point programs from the SPEC95 and SPEC2000 benchmark suites indicate that this technique can consistently improve both the value prediction accuracy and the overall processor performance for each of the different types of value predictors.

1. Introduction

Many recent studies have shown that substantial performance gains may be made possible by predicting the data values that are likely to be produced by instructions [1-11]. This *value prediction* breaks true data dependences to reveal previously unavailable instruction-level parallelism. A few previous studies [3,5,8] have shown, however, that the speedups obtained by using value predictions with *realistic updates* may drop significantly compared to those obtained when using the unrealistic *immediate update* assumption.

While the *immediate update* assumption allows subsequent predictions to use the actual, up-to-date value immediately, the value prediction table in a real pipelined processor can be updated only after the instruction that produces the value has worked its way through the pipeline. Before the value prediction table can be updated with this correct, nonspeculative value, though, subsequent instances of the same instruction (or other instructions) may access the value prediction table to obtain additional predictions.

Since these subsequent predictions will be based on stale values, they are increasingly likely to be incorrect. These incorrect predictions then may introduce additional delays into the pipeline to reissue the incorrectly predicted instructions, thereby decreasing the overall performance. In some cases, the performance may actually decrease to below what it would have been if no prediction were even attempted. The performance impact of the realistic update becomes even more pronounced as the instruction issue width (or the pipeline depth) increases (*cf*. Section 4.3).

Simply recording the number of outstanding predictions that have not yet resolved for each entry of the value prediction table can provide at least a partial relationship between the most recently updated value to the entry and the value that should be predicted for the current instruction instance that is requesting a prediction from this entry [3,12]. For instance, Lee and Yew [3] proposed adding an "age" field in each entry of a stride predictor to record the number of outstanding instructions that requested a prediction with the entry since it was last updated with a known-correct value. The value predictor then makes a prediction for the currently requesting instruction instance as a function of both the current value stored in the entry and the "age" field. We refer to this approach as *hyperprediction* since multiple (possibly different) values are predicted from a single known-correct value. This hyperprediction assumes that the values produced by the dynamic instances of an instruction occur in a predictable sequence.

Since the values in the prediction table are not speculatively updated, this approach eliminates the need for a mechanism to repair stored values that were incorrectly updated with a mispredicted value. It thereby avoids the expensive hardware and misprediction penalty that is required with approaches that speculatively update values, such as those used to speculatively update branch prediction tables [13].

Lee and Yew's previous work [3] has several important shortcomings, however. First, it ignored the influence of instructions squashed in wrongly-predicted branch paths on the number of outstanding instances. As a result, the value of the "age" field was not computed accurately. This deficiency limits both the prediction accuracy of the value predictor and the overall speedup. Second, this previous work studied the hyperprediction technique only for the stride value predictor and did not propose efficient techniques for exploiting hyperprediction in other types of value predictors. Finally, this previous work did not provide a theoretical basis or justification for this new hyperprediction technique, nor did it provide hardware implementation details.

In this paper, we substantially extend this previous work to provide a systematic study of the hyperprediction technique and to propose efficient and complete mechanisms for incorporating hyperprediction into several existing types of value predictors. The mechanisms proposed here compute the number of outstanding instruction instances accurately by compensating for mispredicted branches. These mechanisms also demonstrate how to incorporate hyperprediction into context-based and hybrid predictors. Our simulation results show that with only minimal additional hardware, hyperprediction can improve the prediction accuracy of existing value predictors and, thereby, the processor's overall performance.

The next section describes the basis for the hyperprediction technique in detail, and shows how it can be incorporated into existing value predictors. Section 3 then describes the simulation methodology used to evaluate this new approach with the results analyzed in Section 4. Section 5 discusses some related work with the results and conclusions summarized in Section 6.

2. Hyperprediction of Values

The hyperprediction technique was developed to compensate for the delay that exists when updating the entries in value prediction tables. In the following subsections, we describe the delayed update problem and quantify the magnitude of this delay using the number of *outstanding instruction instances*. We then show how to extend existing value predictors to incorporate hyperprediction.

2.1 The Delayed Update Problem

The delay that occurs when updating a value predictor is illustrated in Figure 1 using a stride value predictor [11] as an example. Assume that the loop enclosing the stride instruction *I* shown in Figure 1(a) has eight machine instructions. An 8-issue processor can fetch one entire loop body in each cycle. Figure 1(b) shows a snapshot of the entry in the stride value prediction table that is indexed by the address of instruction *I* during cycle *i*. During this cycle, the **State** becomes *steady* and the computed **Stride** is one. The **Value** field stores the resolved value of the instance of *I* that occurs immediately before the instance of *I* that is fetched in cycle *i*. Since the branch predictor is still warming up, each of the first several iterations of this loop is followed by the code in the *not-taken* path of the branch instruction *Br*. The next iteration of this loop is fetched only after *Br* is resolved so there is at least a four-cycle branch misprediction penalty. As a result, in the first several iterations of this loop, the instance of the instruction

I in the previous iteration can be resolved before the instance of *I* in the current iteration is fetched. Therefore, the corresponding entry in the value prediction table for *I* is up-to-date during the first several iterations of this loop.

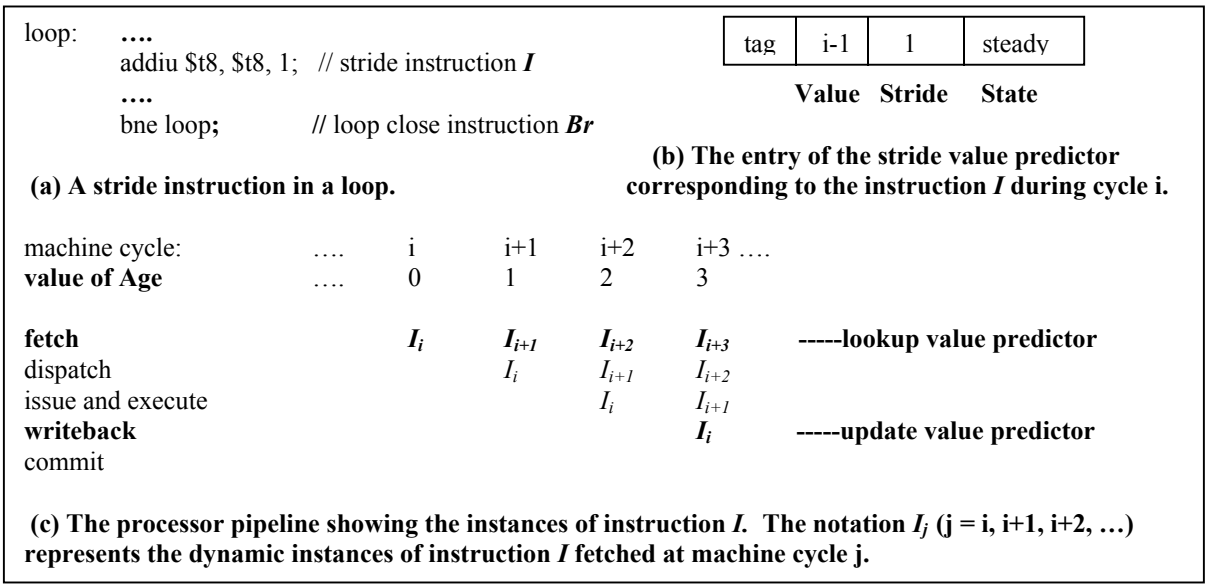


Figure 1. An example to show the delayed update problem for an entry in a stride value predictor.

After the branch predictor stabilizes, however, the processor begins fetching four instances of the instruction *I* (I_i, I_{i+1}, I_{i+2} and I_{i+3}) during cycles *i* to *i+3*, as shown in the pipeline diagram in Figure 1(c). These four instances query the stride value predictor during consecutive machine cycles. The prediction made for instance I_i is based on the up-to-date contents of the entry shown in Figure 1(b). The predictions made for instances I_{i+1}, I_{i+2}, I_{i+3}, though, also are based on the unchanged entry since the actual result produced by instance I_i is not available until machine cycle *i+3*. As a result, all the predictions made for instances I_{i+1}, I_{i+2}, and I_{i+3} are the same value as that made for I_i and are, therefore, wrong.

If there were only four machine instructions in the loop body, or if the processor was capable of issuing sixteen instructions per cycle, a processor with a trace cache [16] would actually fetch two loop bodies in each cycle from cycles *i* to *i+3*. As a result, two instances of *I* would be fetched in each cycle so that eight instances of *I* would query the value predictor and acquire predictions (during cycles *i* to *i+3*) before the next update was made to this entry. Therefore, all but the first of these eight predictions would be wrong. This example suggests that the delayed update problem will tend to get worse for smaller loop bodies or wider processors. In our study, all the benchmark programs tested are compiled with the loop unrolling optimization turned on in order to increase the size of loops and thereby compensate for this

effect as much as possible at compile-time. Even with this compiler optimization, though, our simulation results in the following subsection show the delayed update problem to be significant.

2.2 The Number of Outstanding Instruction Instances

We define an *instance* of a static instruction as a dynamic occurrence of that instruction. The *number of outstanding instances* of an instruction, N, is the number of instances that have been fetched but have not yet produced values.

To record the number of outstanding instances of an instruction that access the value predictor, we augment each entry of the value prediction table, which is indexed by the instruction address, with a new field, called **Age** (as in [3]). Note that there are two levels of prediction tables in some value predictors, such as the two-level or the hybrid value predictor [11]. Only the first level table, the value history table (VHT), is indexed by the instruction address. Thus, only this single table needs to be augmented with the **Age** field. The following algorithm is used to compute the value of **Age** for each instruction.

The value of **Age** is:

1) Incremented by one during the instruction fetch stage *after* an instance of this instruction indexes the entry to make a prediction.
2) Decremented by one during the write-back stage *after* an instance of this instruction updates the entry with its newly-computed value.
3) Decremented by one during the write-back stage when an instance of this instruction is squashed due to a mispredicted branch instruction.

In this algorithm, step (3) is particularly important for computing the value of **Age** accurately since some of the instruction instances that queried the value prediction table may turn out to be on a wrongly-predicted execution path. These instructions will be squashed from the processor pipeline and so must not update the value prediction table. The value prediction table entry for an instruction is up-to-date whenever the value of **Age** is zero. On the other hand, any non-zero value of **Age** represents the number of outstanding instances of this instruction that have queried the table before the current instance.

In Figure 1(c), we listed the values of **Age** for instruction *I* from cycles *i* to *i+3*. Note that all the values listed are computed at the beginning of each cycle. For example, at the beginning of cycle *i+3*, the value of **Age** for instruction *I* is three because we increase it by one after making the prediction for each instance I_i, I_{i+1} and I_{i+2} during cycles *i, i+1* and *i+2*.

To obtain a sense of the magnitude of the delayed update problem, Figure 2 shows the distribution of the value of **Age** for all the dynamic instructions that query a hybrid value predictor (as described in Section 2.3.3) with an infinite prediction table size and an infinite number of read/write ports. The three

bars for each program show the results obtained on a 4-, 8-, and 16-issue superscalar processor, respectively. It is clear that a reasonably large fraction of instructions have non-zero **Age** values for each processor configuration. This fraction becomes larger as the issue-width increases. For programs *ijpeg*, *mcf* and *ammp,* approximately 50% of the dynamic instructions have non-zero **Age** values when the issue width is 8 or 16. As a result, these programs suffer more seriously from delayed updates to the value predictor than the other programs (*cf*. Section 4). It is also clear that, for more than 98% of the instructions, the **Age** value never exceeds eight. Therefore, it is sufficient to use three bits for the **Age** field in a real value predictor for the programs tested. The distribution of **Age** values obtained when using a stride value predictor and a two-level context-based value predictor is almost the same as that shown in Figure 2. Due to space limitations, we do not include these results here.

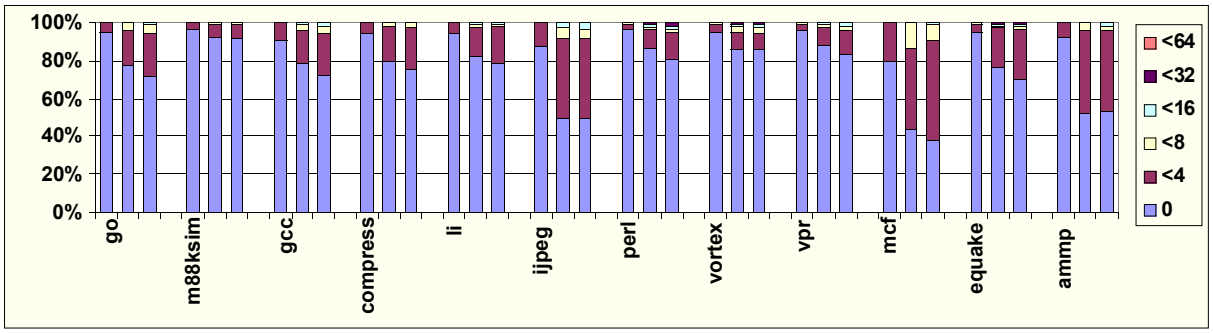


Figure 2. The distribution of the value of **Age** for the dynamic instructions that query a hybrid value predictor. The three bars for each program denote results for a 4-, 8-, and 16-issue processor, respectively, from the left.

The **Age** value of an instruction is a measure of the distance between the current instance of the instruction that is indexing the corresponding entry in the value predictor and the previous instance that last updated the entry. When the values produced by instances of this instruction form a predictable sequence, hyperprediction can reliably predict a value for the current query based on both the current value in the entry and the value of **Age**, even though the current entry is stale.

2.3 Incorporating Hyperprediction into Value Predictors

The types of value predictability characteristics that can be captured by a stride value predictor and by a context-based value predictor are quite different. As a result, these two value predictors require different implementations of the hyperprediction technique. A hybrid value predictor, on the other hand, is a combination of stride and context-based value predictors. Not surprisingly, the implementation of the hyperprediction technique for a hybrid value predictor is a simple combination of that used for a stride

value predictor and that used for a context-based value predictor. A *last value* predictor always tries to predict the same value as the one currently stored. Since this constantly repeating value is independent of the value of **Age** of an instruction, hyperprediction cannot improve the performance of a last value predictor. Consequently, we do not consider the last value predictor further in this study, except where this type of behavior may be captured automatically by a context-based predictor.

In the remainder of this section, we describe how to incorporate the hyperprediction technique into the stride, context-based and hybrid value predictors.

2.3.1 Stride value predictor

When the strided sequence of values produced by adjacent instances of an instruction is stable, there is a functional relationship between the predicted value for the current instruction instance and the value that was used to last update the entry. Specifically,

predicted value = **Value** + (**Age**+1**) * Stride.**

For the stride instruction *I* in Figure 1, for example, instance I_{i+3} is querying the stride value predictor while, as shown in Figure 1(b), the corresponding entry in the predictor is being updated using the computed value of instance I_i. Since the **State** is *steady* already, the stride predictor assumes that the results produced by instances of *I* form a predictable sequence with a **Stride** of 1. Furthermore, since the number of outstanding instances before instance I_{i+3} is three during this cycle, the stride value predictor knows that there are three other instances between instance I_{i+3} and the instance that produced the result currently stored in the **Value** field. Using this information, the predictor can make a reliable prediction for instance I_{i+3}, specifically, [**(i-1) + (3+1) * 1**] = $i+3$, which is the correct value for this instance.

In the original stride value predictor [11], each entry in the prediction table has the following fields -- **Tag, State, Value,** and **Stride.** We need to add only a 3-bit **Age** field in each entry to allow hyperprediction. In order to compute the result of [**Value** + (**Age**+1) * **Stride**], we use a small multiply-accumulate unit (MAU) similar to that described in [14] to replace the adder in the original stride value predictor. Other hardware implementations are possible, of course, but we do not examine these alternatives in this study.

This small MAU performs the following tasks: 1) increments the **Age** by 1; 2) multiplies this result by the **Stride**; and 3) adds the result of step (2) to the **Value**. Since three bits are sufficient for the **Age** field (see Section 2.2), and eight bits are sufficient for the **Stride** field [7,8], the MAU needs to multiply

only a 4-bit operand (the result of **Age**+1) with an 8-bit operand (**Stride**) and add this product to a 32-bit operand (**Value**). Thus, the hardware cost for this small MAU is quite reasonable. Furthermore, as shown in [14], an MAU can be constructed to multiply two 16-bit operands and add the result to a third 40-bit operand in a single cycle. Consequently, the small MAU required to add hyperprediction to the stride value predictor certainly can finish the computation in one cycle.

In addition to the above modification, we need to make the following changes in the operations of the stride predictor to allow hyperprediction.

Operations. There are three basic operations required in the new stride value predictor:

1. *Lookup*. In the lookup operation, a new instruction instance indexes the value predictor using the instruction address during the fetch stage. While the **Tag** is being compared, the predicted value [**Value** + (**Age**+1) * **Stride**] is being computed by the MAU. If the **Tag** does not match this instruction, the lookup is aborted due to a table miss. Otherwise, the **State** of the indexed entry is checked. If it is not *steady* yet, the predictor cannot provide a prediction due to the confidence value being below the threshold necessary for a prediction. If the **State** is *steady*, however, the predictor makes a prediction for the current instance by using the result of the MAU computation. After making the prediction, the **Age** is incremented by one.
2. *Update*. In the update operation, a completed instruction instance indexes the value predictor using the instruction address during the write-back stage. If the **Tag** of the indexed entry does not match this instruction, a new entry is allocated to this instruction. The fields of this new entry are initialized such that: 1) both **Stride** and **Age** are set to zero; 2) **Value** is set to the value this instance is writing back; and 3) **State** is set to *initial*. If the **Tag** does match this instruction, however, the *new stride* is calculated by subtracting the value currently in the table from the value being written back by the current instance. Furthermore, the **State** and **Stride** fields are updated as follows:

```
if State == initial, then
    Stride = new stride; State = transient;
else if State = = transient, then
    if new stride = = old stride currently stored in Stride, then
        State = steady;
else if State = = steady, then
    if new stride != old stride currently stored in Stride, then
        Stride = new stride; State = transient;
    if the resolved value != the predicted value, then
        State = initial;        /* When the resolved value is not the same as the predicted
                                   value, the State is set to initial to stop subsequent
                                   predictions since they are likely to be wrong. */
```

After updating **State** and **Stride**, the new value is stored in the **Value** field, and the **Age** is decremented by one.

3. *Recover*. When an instance of an instruction is on the wrongly-predicted path of a branch instruction, it will be squashed by the branch misprediction recovery mechanism during the write-back stage. This squashed instruction instance must index the value predictor using the instruction address. If the **Tag** of the indexed entry matches this instruction, the **Age** will be decremented by one.

2.3.2 Context-based value predictor

We use the two-level value predictor proposed by Wang and Franklin [11] to study the hyperprediction technique in a context-based value predictor. There are two levels of tables used in this predictor -- the value history table (VHT) and the pattern history table (PHT). The PHT is kept the same as in [11]. Only the VHT needs to be modified to incorporate hyperprediction.

The example shown in Figure 3 is used to illustrate the operation of the new two-level value predictor with hyperprediciton. Figure 3(a) shows an instruction, *I*, that produces a sequence in which the four unique values, 1, 4, 7, and 13, are repeated. Instance I_{i+5} of instruction *I* is the one whose result was used to last update the value predictor, while instance I_{i+8} is querying the value predictor currently. There are two outstanding instances of *I* (I_{i+6} and I_{i+7}) while I_{i+8} is querying the value predictor. Figure 3(b) shows a snapshot of the entry in the VHT indexed by the address of *I* when instance I_{i+8} is querying the VHT.

In the original two-level value predictor [11], there are four fields in each entry of the VHT -- **Tag, Data Values, LRU,** and **Value History Pattern**. The **Data Values** field consists of an array that is indexed as {00, 01, 10, 11} from the left. It holds the four most recent values produced by the instruction. Other fields of the VHT entry use these 2-bit indexes into the array to identify specific values in the **Data Values** array. The **LRU** bits are used to keep track of the order in which the four values were last produced. When a fifth unique value is produced, it replaces the least recently seen value in the **Data Values** array according to the **LRU** bits. The **Value History Pattern** field records a *2p-bit* pattern corresponding to the last *p* outcomes produced by the instruction ($p = 4$ in this example). The leftmost 2-bit field points to the earliest produced value. This *2p-bit* pattern is used to index the second level prediction table (PHT). Each entry of the PHT contains an array of four independent up/down counters that is indexed as {00, 01, 10, 11} from the left. Each element of the array corresponds to the value in the

Data Values array with the same index. Figure 3(c) shows the snapshot of the entry in the PHT indexed by the **Value History Pattern** (with $p = 4$) of the VHT entry in Figure 3(b).

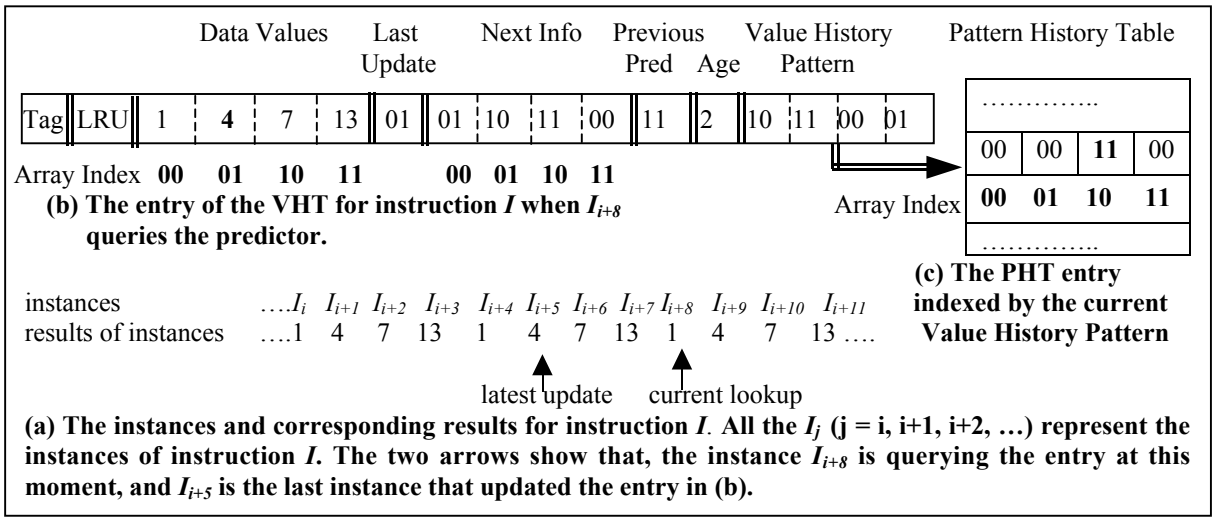


Figure 3. An example to show how the hyperprediction technique is incorporated into a two-level context-based value predictor.

In addition to these fields needed for the basic two-level value predictor [11], we need four more fields in each entry to incorporate hyperprediction -- **Next Info**, **Last Update, Previous Pred** and **Age**. Unlike the stride-based predictor, there is not an explicit functional relationship in the two-level value predictor between the most recently updated value to the entry and the value to be predicted for the current instruction instance. Instead, the two-level value predictor needs to form the relationship at runtime among the values stored in the **Data Values** array. The **Next Info** field is used to record this dynamic relationship by implementing a circular linked-list showing the order in which the corresponding values in the **Data Values** array were produced. It consists of an array of four 2-bit subfields that is indexed as {00, 01, 10, 11} from the left. The subfield of the **Next Info** array with index *N* points to the value produced immediately after the value stored in the **Data Values** array with index *N*. For example, the fourth subfield of the **Next Info** field in Figure 3(b) is **Next Info**[11]=[00], which means that **Data Values**[11], *i.e.*, the value 13, is followed by **Data Values**[00], *i.e.*, the value 1, in the repeated sequence of values produced by this instruction.

The 2-bit **Last Update** field is used to construct the **Next Info** linked-list at runtime. It is set to always point to the value in the **Data Values** array that was last update this entry. The **Next Info** subfield that corresponds to the value currently updating the entry is set to **Last Update** as the entry is updated. When the values produced by subsequent instances of the instruction form a stable repeated sequence,

each value in the **Data Values** array is always followed by the same value. As a result, the **Next Info** list remains constant.

When the original two-level value predictor [11] tries to make a prediction, it uses the **Value History Pattern** to index the PHT. If the maximum counter value among the four counters in the extracted PHT entry is larger than or equal to the predefined threshold, the value corresponding to this maximum counter is chosen as the predicted value. This also is true in the two-level value predictor with hyperprediction if the **Age** is zero. When the **Age** is larger than zero, however, the new value predictor must choose the value that is produced **Age** instances later than the value corresponding to the maximum counter. This can be accomplished by following the **Next Info** list **Age** times starting from the subfield of the **Next Info** list that corresponds to the value with the maximum counter value.

In Figure 3, for example, since the maximum counter value in the extracted PHT entry is larger than the predefined threshold, and since **Age** is two, the index of the value corresponding to the maximum counter, [10], will be the starting point of the search through the **Next Info** list. We need to follow the **Next Info** list **Age** = 2 times, *i.e.*, **Next Info**[10] = [11], **Next Info**[11] = [00]. The index [00] then is the index for the predicted value. However, since the value of **Age** can be as large as eight in this study, up to eight links may need to be followed, which could introduce a long delay to make a prediction. In order to reduce this delay, we add a 2-bit pointer, **Previous Pred,** which always points to the previously predicted value in the **Data values** array. That is, **Previous Pred** points to the tail of the circular linked-list. As a result, whenever a prediction needs to be made, we need to follow the **Next info** list only one more step starting from the **Previous Pred** pointer into the **Next info** list. In Figure 3(b), for instance, the index of the predicted value is **Next Info**[**Previous Pred**] = **Next Info**[11] = [00]. Thus, the value to be predicted is **Data Values**[00] = 1. Note that this predicted value is actually a correct prediction for this instance.

In addition to the above modification of the prediction table, we include the following operational changes to incorporate hyperprediction into the two-level value predictor.

Operations. There are three basic operations in the new two-level value predictor:

1. *Lookup*. In the lookup operation, a new instruction instance indexes the VHT using the instruction address during the instruction fetch stage. As the **Tag** is compared to the address, the PHT is indexed using the **Value History Pattern**. If the **Tag** does not match this instruction, the lookup is aborted due to a table miss. Otherwise, the counter with the largest value in the extracted PHT entry is chosen

(if there is a tie, one of the values can be selected at random, or based on the value that was most recently selected). If the maximum counter value is less than the threshold, no prediction is made. Otherwise, the prediction can be made as follows:

```
if Age = = 0, then
        index of the predicted value = the index of the Data Values array that corresponds
        to the maximum counter value of the extracted PHT entry;
else
        index of the predicted value = Next Info[Previous Pred];
        // The element of the Next Info array with index Previous Pred.
Previous Pred = index of the predicted value;
```

After making the prediction, **Age** is incremented by one.

2. *Update.* In the update operation, an instruction instance in the write-back stage indexes the VHT using the instruction address. The **Value History Pattern** of the selected VHT entry is shifted left by two bits, and the 2-bit index of the new value in the **Data Values** array is entered in the bits left vacant by the shift. The **Next Info** and the **Last Update** fields are updated as follows:

```
Next Info[Last Update] = index of the new value;
Last Update = index of the new value;
```

The **Age** field then is decremented by one.

3. *Recover.* This Recover operation is the same as that for the stride value predictor described in Section 2.3.1.

2.3.3 Hybrid value predictor

Incorporating hyperprediction into the hybrid value predictor requires only a straightforward combination of the modified stride and two-level value predictors described above. Both the stride value predictor and the two-level value predictor make predictions according to the steps described in Sections 2.3.1 and Section 2.3.2, respectively. The prediction made by the two-level value predictor is selected first as the prediction made by the overall hybrid value predictor, although the choice of which component predictor to select first is arbitrary since our simulations have shown that selecting the stride value predictor first produces similar results. If the two-level value predictor does not provide a prediction due to its confidence counter value being below the predefined threshold, the prediction made by the stride value predictor is selected. If neither component predictor can make a prediction, the overall predictor will not make a prediction.

3. Performance Evaluation Methodology

We compare the performance of the stride, two-level, and hybrid value predictors using the hyperprediction technique to the same predictors using a realistic update mechanism. The results from the idealized immediate update mechanism also are included to show how close these techniques come to achieving a coarse upper bound. The value predictors with a realistic update mechanism are the same as those used by Wang and Franklin in [11]. The value predictors with immediate update are the same as the value predictors with realistic update, except that the value tables are updated with the correct value immediately after making a prediction. The new value predictors with hyperprediction are implemented as described in Section 2 with an **Age** field of 3 bits. All the value predictors have 8K entries for each of the tables. Our simulations showed that the relative performance rankings among the various predictors remained the same as the sizes of the tables were varied. Due to space limitations, we do not include all of these results in this paper.

3.1 Baseline Machine Model

The cycle-accurate execution-driven simulator used in this study was derived from the *sim-outorder* simulator in the SimpleScalar tool set [15]. The baseline processor is an out-of-order superscalar processor using the register update unit (RUU) [15] with the parameters shown in Table 1. The processor can issue 8 instructions per cycle out of a 256-entry instruction window, and the load-store queue has 64 entries. To support the high fetch width needed for this configuration, the processor uses a 16K-set, 4-way trace cache [16] with a hybrid branch predictor. The hybrid branch predictor consists of a 16K-entry *gshare* predictor with 14 history bits, and a 16K-entry bimodal predictor.

3.2 Incorporating Value Predictor Hardware into the Baseline Processor

There are many different ways to incorporate value predictors into the pipeline of a superscalar processor [2,3,8,9] and it is beyond the scope of this paper to propose an optimal design. Instead, we focus on a "typical" design. In this study, the value prediction table is indexed by the instruction address during the instruction fetch stage. The predicted result values are inserted into the instruction window after the instructions are decoded in the dispatch stage. When all the operands of an instruction in the instruction window are available, either through prediction or by having the correct value already in a register, and there is an available functional unit, the instruction will be issued and executed. After the actual value is produced during the write-back stage, it will be compared with the predicted value that was used by this

instruction. If the actual result is the same as the predicted result, the execution is continued without any interruption. Otherwise, only those instructions that were issued using this incorrectly predicted value are invalidated and reissued with the correct value [8].

Table 1. Machine configuration for the baseline processor architecture used in the simulations.

	parameter	value
Processor Core	instruction window load-store queue fetch/decode/issue/commit width functional units	256 entries 64 entries 8 instructions/cycle 8 integer units, 1-cycle latency 4 floating point units, 2-cycle latency 2 integer multiply/divide units, 3/12-cycle latency 2 floating point multiply/divide units, 3/12-cycle latency
Branch Predictor	branch target buffer (BTB) hybrid branch predictor return address stack (RAS)	2K, 2-way gshare with16K entries, 14-bit history, bimodal with 16K entries 32 entries
Trace Cache	16K sets, 4-way, 8 instructions/entry	
Caches	level-one data cache level-one instruction cache level-two cache (unified)	512K, 2-way, 32 bytes/block, 3-cycle latency 512K, direct-map, 32 bytes/block, 3-cycle latency 1M, 4-way, 64 bytes/block, 30-cycle latency

3.3 Benchmark Programs

Ten integer programs and two floating-point programs from the SPEC95 and SPEC2000 benchmark suites are used in this study. Table 2 shows the input sets used, the total instruction counts, and the IPC obtained for the 8-issue baseline processor for these programs. The percentage of the total number of dynamic instructions that need to query the value predictor is shown in the fourth column of this table. All the programs were compiled using the GCC 2.6.3 compiler with –O3 optimization and loop unrolling enabled. A few of the input sets for the SPEC95 programs were modified slightly to control the simulation time. Additionally, the input sets for the SPEC2000 programs were carefully modified to reduce simulation time while approximately maintaining the important characteristics that the programs demonstrate when executed using the original input sets [17].

3.4 Evaluation Metrics

The most important metric used in this study is *SpeedUp*, which is computed as (T0/T1 –1)*100%, where T0 is the execution time obtained when executing a benchmark program on the baseline processor and T1 is the execution time obtained on the same processor (or a different processor) with one of the value predictors added. Two additional metrics are defined to evaluate the properties of the value predictors themselves, and to explain the observed differences in speedup. The *Correct prediction rate* is the

percentage of instructions for which the results have been correctly predicted out of the total number of instructions issued. Similarly, the *Misprediction rate* is the percentage of predictions that predict the wrong values out of the total number of instructions issued. Note that the *Correct prediction rate* and the *Misprediction rate* do not sum to one since no predictions are attempted for some of the instructions (i.e., those that are branch instructions, or those for which the confidence counter is below the required threshold, or for which there is a table miss).

Table 2. The run-time characteristics of the SPEC95 and SPEC2000 benchmark programs

Benchmark	Input Set	No. of inst. executed (million)	Percentage of inst. that attempt prediction (%)	IPC for baseline machine	Benchmark set
go	5,9	197	78	1.58	SPEC95 integer
m88ksim	dcrand.lit	242	68	4.24	
gcc	jump.i	75	65	1.90	
compress	10000 e 2231	62	66	2.83	
li	queen6.lsp	69	59	2.65	
ijpeg	spectriv.ppm	110	85	4.70	
perl	scrabbl.in	72	77	2.48	
vortex	persons.250	73	59	3.10	
vpr	small.arch.in	40	70	1.95	SPEC2000 integer
mcf	smred.in	186	61	2.58	
equake	lgred.in	780	55	2.39	SPEC2000 floating-point
ammp	smred.in	45	55	1.04	

The *Correct prediction rate* has the greatest influence on the overall performance improvement since correct predictions break true data dependences to expose more instruction-level parallelism [4]. Not all the correct predictions actually produce a performance improvement, though. For instance, correctly predicting a value for an instruction that is not on the critical path may not produce any performance improvement. As a result, there is not necessarily a proportional relationship between the *SpeedUp* and the *Correct prediction rate.* Mispredictions, on the other hand, may decrease performance since instructions that have been previously issued with an incorrectly predicted value must be squashed and reexecuted.

In the Simplescalar machine model [15] used for this study, the verification of predicted values is done in the write-back stage so that there is no need for an additional stage in the pipeline to verify and update the predicted values. As a result, there is no compulsory penalty for value mispredictions [4,8]. The only penalty caused by wrongly-predicted values is a potential structural hazard when reissued instructions compete for limited hardware resources with other instructions, such as the instruction window and function units. However, it has been shown [4] that these structural hazards tend to have little

influence on performance. Based on these considerations, we use a misprediction penalty of one cycle in most of our study. However, in Section 4.5 we also examine the sensitivity of the performance to the misprediction penalty**.**

3.5 Notation

We use the following notation to refer to the different value predictors in subsequent sections:

T: value predictor of type "T" with a realistic update scheme;

T_hyper: value predictor of type "T" with the hyperprediction scheme;

T_imm: value predictor of type "T" with an immediate update scheme.

In this notation, the value predictor type "T" can be one of "Stride," "Two," and "Hybrid," denoting the stride, two-level, and hybrid value predictors, respectively**.** For example, *Stride_hyper* denotes a baseline processor with a stride value predictor using the new hyperprediction mechanism.

4. Analysis of Results

In this section, we first evaluate the performance potential of the hyperprediction scheme when added to the different value predictors when the predictors have an unlimited number of read/write ports. We then evaluate the impact of limiting the number of read/write ports and the impact of varying the issue width and the processor's branch prediction accuracy. The misprediction penalty of the value predictors is set to one cycle in these simulations. However, we also include results in Section 4.5 examining the sensitivity of the overall performance to other misprediction penalties.

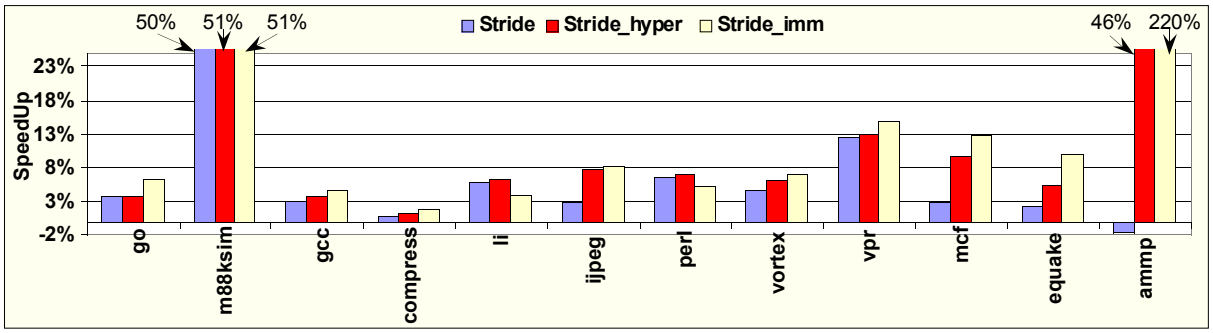
4.1 Performance Improvement Due to Hyperprediction

The speedup results for hyperprediction in the stride and two-level value predictors are presented first. The speedups obtained when augmenting the hybrid value predictor with hyperprediction are presented separately since the hybrid predictor essentially combines the performance effect of both of its two component predictors**.**

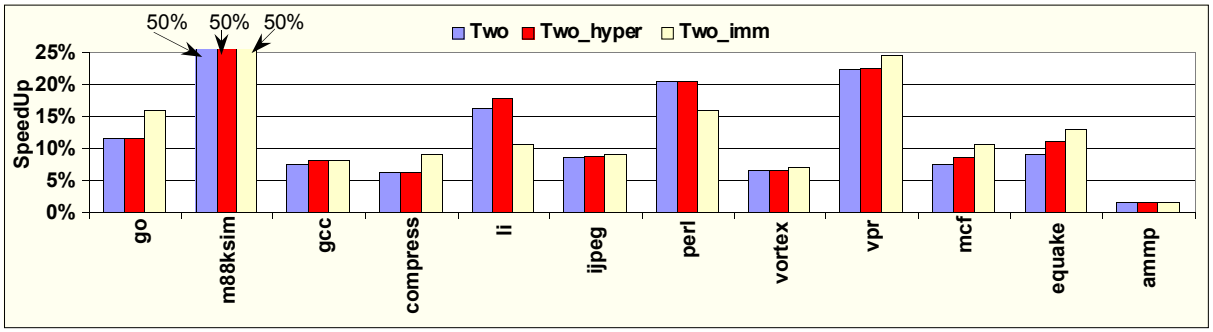
4.1.1 Basic value predictors

Figure 4 shows the speedups obtained over the baseline processor for the stride and two-level value predictors with an (idealized) immediate update mechanism, with the hyperprediction mechanism, and with realistic update. It is clear that all of the prediction mechanisms produce moderate to substantial speedups for all of the applications, except for the stride predictor with realistic update on *ammp*. (Note

that the stride predictor with hyperprediction produces 46% speedup for this program, however.) Compared to the realistic update mechanism, the hyperprediction mechanism in the stride-based predictor produces a particularly dramatic increase on performance for *ijpeg, mcf, equake,* and *ammp*; it produces a slight performance increase for *gcc, compress, li, perl, vortex*, and *vpr*; and it matches the performance for *go* and *m88ksim.* Recall that Figure 2 showed that approximately 50% of the dynamic instructions for *ijpeg*, *mcf* and *ammp*, and 30% of the dynamic instructions for *equake,* have **Age** values larger than zero. As a result, Figure 5(a) shows that these programs have much higher correct prediction rates and lower misprediction rates when using hyperprediction than when using the realistic update scheme, which translates to the performance improvements shown in Figure 4(a).



(a) Stride value predictors



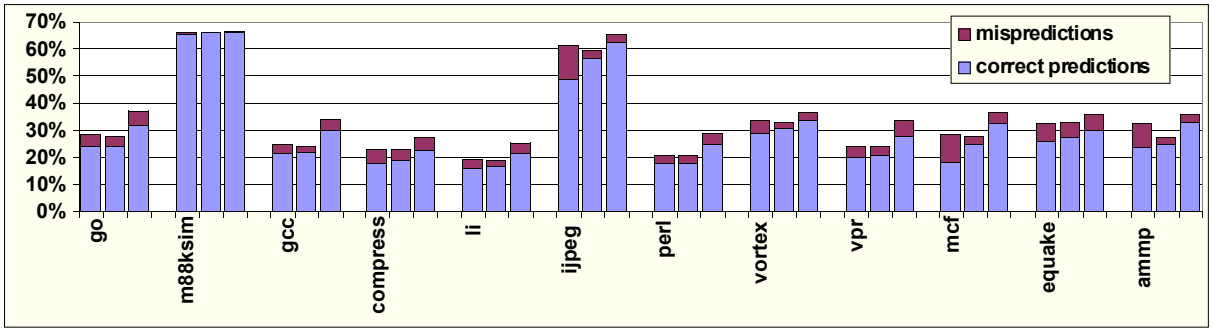
(b) Two-level value predictors

Figure 4. The speedup for the basic value predictors with realistic update, hyperprediction, and immediate update on an 8-issue processor.

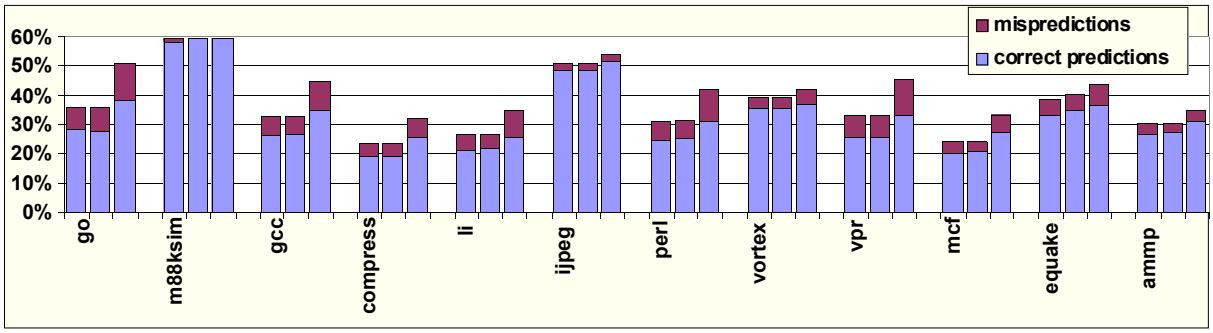
Using hyperprediction in the two-level value predictor improves the performance for *gcc, li, mcf* and *equake* more than when using a realistic update scheme, as shown in Figure 4(b). For the remaining programs, hyperprediction and realistic updating produce essentially the same performance improvement.

Comparing Figures 4(a) and 4(b), we see that hyperprediction has a greater impact on performance with the stride predictor than with the two-level predictor. This difference occurs because the two-level value predictor with hyperprediction exploits repeated value sequences such as {1, 4, 7, 11, 1, 4, 7, 11, 1,

4, 7, 11, …}. However, these types of stable repeated sequences infrequently appear in most of the programs tested. As a result, the two-level value predictor does not benefit as much from hyperprediction as does the stride predictor.



(a) Stride predictors. The three bars for each program denote Stride, Stride_hyper, and Stride_imm from the left.



(b) Two-level predictors. The three bars for each program denote Two, Two_hyper and Two_imm from the left.

Figure 5. The value predictor properties for the basic value predictors when using the realistic update, hyperprediction, and immediate update mechanisms.

It is clear that, for most of the programs, there is a performance gap between the hyperprediction mechanisms and the unrealistic immediate update mechanisms. Most of the performance gap is due to instructions that change their predictability patterns slowly at run-time. For example, in the dynamic sequence {1, 2, 3, 4, 7, 8, 9, 10, …}, the shift from 4 to 7 ends one steady stride sequence. A new stride sequence then starts from 7. Since a value predictor with an immediate update mechanism can be updated with the value 7 immediately after making a prediction for the instruction instance that produced 7, all of its subsequent predictions will be made using the most up-to-date value and are likely to be correct. Although the hyperprediction mechanism has correct information about the number of outstanding instances, it cannot make correct predictions before the value predictor is updated with the value 7 since the starting point of the next subsequence is not yet known.

Figure 5 shows that the unrealistic immediate update mechanism always makes more predictions than the other mechanisms. While it produces a higher correct prediction rate than the other mechanisms for all the programs, the immediate update mechanism also produces a higher misprediction rate than the other mechanisms for *go, gcc, li, perl* and *vpr*. For *li* and *perl*, the immediate update mechanism actually produces lower speedups than the other mechanisms due to these higher misprediction rates.

4.1.2 Hybrid value predictor

Figure 6 shows the speedups obtained for the hybrid value predictor with the different update mechanisms while Figure 7 shows the corresponding predictor properties. It is interesting to note that the hybrid value predictor with hyperprediction can usually pick the correct of the two predictions made by the component value predictors. As a result, the overall speedup introduced by hyperprediction in the hybrid value predictor almost perfectly combines the effect of hyperprediction in both the stride and two-level value predictors. For example, even though hyperprediction in the two-level value predictor cannot improve the performance significantly for *mcf* and *ammp*, the hybrid value predictor with hyperprediction provides significant speedup for both of these programs. On the other hand, for *li* and *equake,* the difference in the speedup obtained with the hyperprediction mechanism and with the realistic update mechanism is clearly larger for the hybrid value predictor than for the stride value predictor. This larger difference occurs for these two programs since the two-level value predictor component of the hybrid value predictor benefits substantially from hyperprediction for these programs, as was shown in Figure 4(b).

In the following sections, we focus on only the hybrid value predictor since it demonstrates the combined effects of hyperprediction in both stride and two-level value predictors.

4.2 Impact of Limited Read/Write Ports

When the number of read/write ports in the value predictor is limited, some instructions that try to access the predictor will conflict with other instructions trying to access the predictor simultaneously. As a result, instructions that are not granted access to a sufficient number of ports in the current cycle will not have predictions made for them. Additionally, read/write port conflicts can prevent retiring instructions from updating the predictor with the actual value in time for it to be used for a subsequent prediction. Both of these effects will decrease the performance produced when using a value predictor. In addition to these conflicts, limited numbers of read/write ports with the hyperprediction mechanism can prevent

instructions that are squashed due to branch mispredictions from updating the **Age** field appropriately, which will cause subsequent mispredictions.

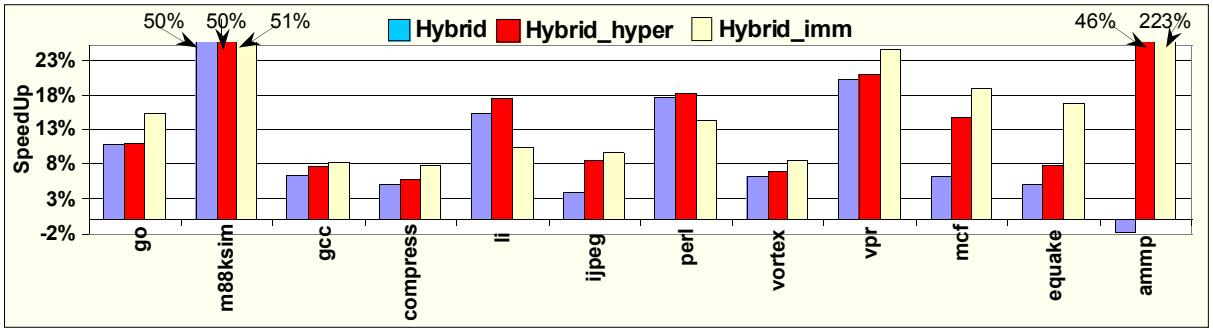


Figure 6. The Speedup for the hybrid value predictor with the different update mechanisms on an 8-issue processor.

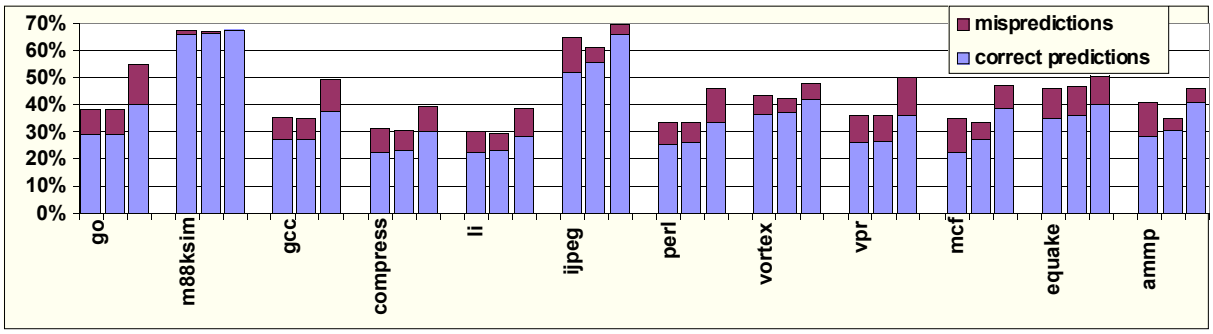


Figure 7. The value predictor properties for the hybrid value predictors with the different update mechanisms. The three bars for each program denote Hybrid, Hybrid_hyper, and Hybrid_imm from the left.

Figure 8 shows the speedups obtained over the baseline processor for the hybrid value predictors with the different update mechanisms when the number of read/write ports in each value predictor is varied. As expected, the speedup decreases for all of the update mechanisms when the number of read/write ports decreases. However, the hyperprediction mechanism continues to consistently outperform or match the performance of the realistic update mechanism in all cases.

4.3 Sensitivity to Instruction Issue Width

Figure 9 shows the speedups obtained over the baseline 8-issue processor for the hybrid value predictor with the different update mechanisms as the number of instructions that can be issued per cycle is varied. Note that the baseline processor configuration for each issue-width in this figure is the same as described in Table 1. Consequently, the speedups for the 4-issue processor are negative in most of the cases. It is clear that the hyperprediction mechanism improves upon or matches the performance of the realistic update mechanism in all cases. Figure 2 showed that the fraction of dynamic instructions whose **Age**

value is larger than zero increases with increases in the instruction issue-width. As a result, there is an increasingly greater benefit from using the hyperprediction mechanism as the issue width increases compared to the realistic update mechanism.

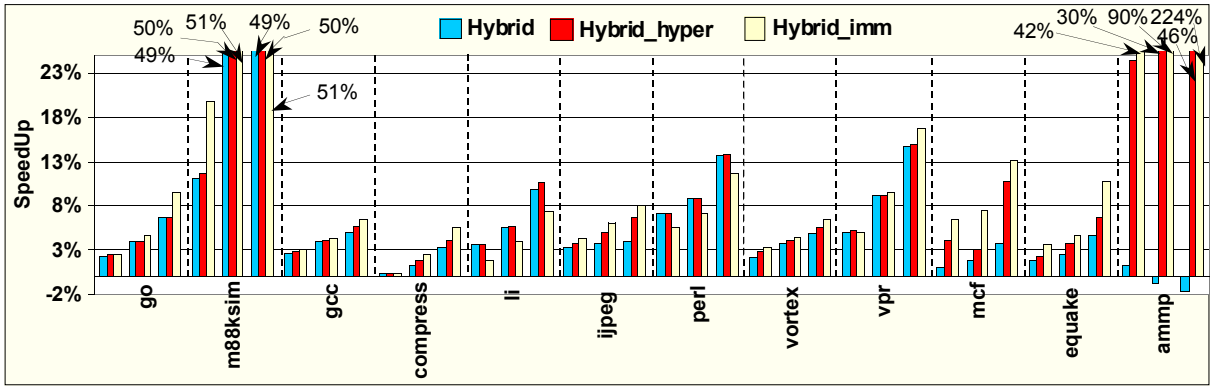


Figure 8. The performance effect of varying the number of read/write ports in a hybrid value predictor with the three different update mechanisms on an 8-issue processor. The nine bars for each program are divided into three groups showing speedups for 1, 2, and 4 read/write ports, from the left.

4.4 Impact of the Branch Prediction Accuracy

The hyperprediction mechanism requires instructions that are squashed due to branch mispredictions to update the value prediction table. Consequently, it is possible that this mechanism could be sensitive to the accuracy of the branch predictor used in the processor. To test this sensitivity, we experimented with three different branch predictors: 1) a bimodal branch predictor with 64 entries; 2) the hybrid branch predictor used in the previous simulations and described in Table 1; and 3) an unrealistic perfect branch predictor. The prediction accuracies provided by the bimodal and the hybrid branch predictors are shown in Table 3. The perfect branch predictor has 100% prediction accuracy.

Table 3. The prediction accuracy (%) of the bimodal and the hybrid branch predictors.

	go	m88ksim	gcc	compress	li	ijpeg	perl	vortex	vpr	mcf	equake	ammp
bimod	77	97	77	88	86	95	81	85	77	92	80	91
hybrid	85	99	93	94	97	97	97	99	89	95	94	99

Figure 10 shows the speedups obtained with the various update mechanisms over the baseline machine when using the different branch predictors. We can see that the hyperprediction mechanism consistently improves the overall performance in all cases, and that it is not particularly sensitive to the quality of the branch predictor. Note that the baseline processor configuration for each branch predictor in this figure is the same as described in Table 1. Consequently, the speedups for the bimodal branch predictor are negative in most of the cases.

4.5 Sensitivity to the Value Misprediction Penalty

The previous simulations used a penalty of one cycle whenever an instruction needed to be reissued due to using a mispredicted value. To determine the sensitivity of these results to this specific misprediction penalty value, we tested penalties of 0, 1, 2, and 4 cycles. All of the value predictors produced negative speedups for almost all of the test programs with a misprediction penalty of 4 cycles. Thus, we do not show these results here. From Figure 11, which shows the speedups obtained for misprediction penalties of 0, 1 and 2 cycles, we see that the relative performance benefits of hyperprediction tend to increase compared to the realistic update mechanism as the misprediction penalty increases. These results suggest that the hyperprediction technique is not as sensitive to the misprediction penalty as the realistic update mechanism. Furthermore, even with a misprediction penalty as high as two cycles, the hyperprediction mechanism produces positive speedups for all of the test programs except *compress*. The realistic update mechanism, in contrast, slows down the processor for five of the twelve test programs with a two-cycle misprediction penalty.

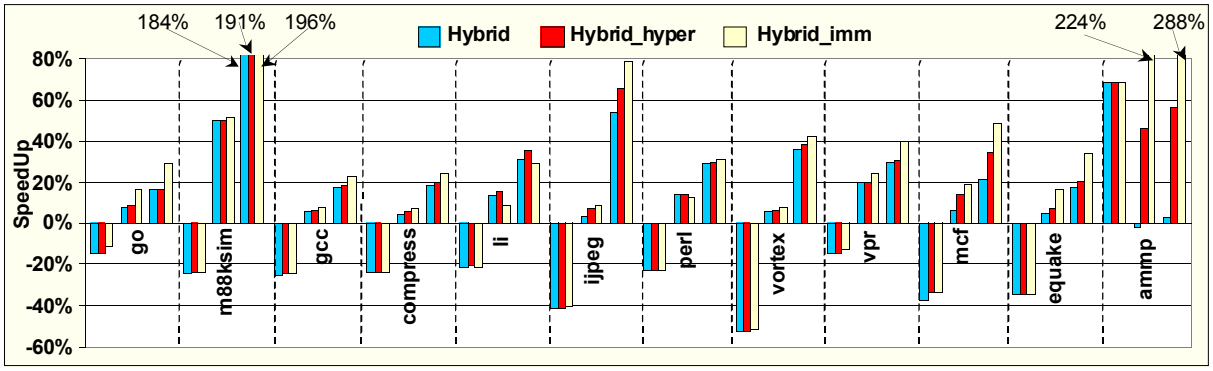


Figure 9. The performance effect of varying the instruction issue width on the hybrid value predictors with the three different update mechanisms. The nine bars for each program are divided into three groups showing speedups for issue widths of 4, 8, and 16 instructions per cycle, from the left.

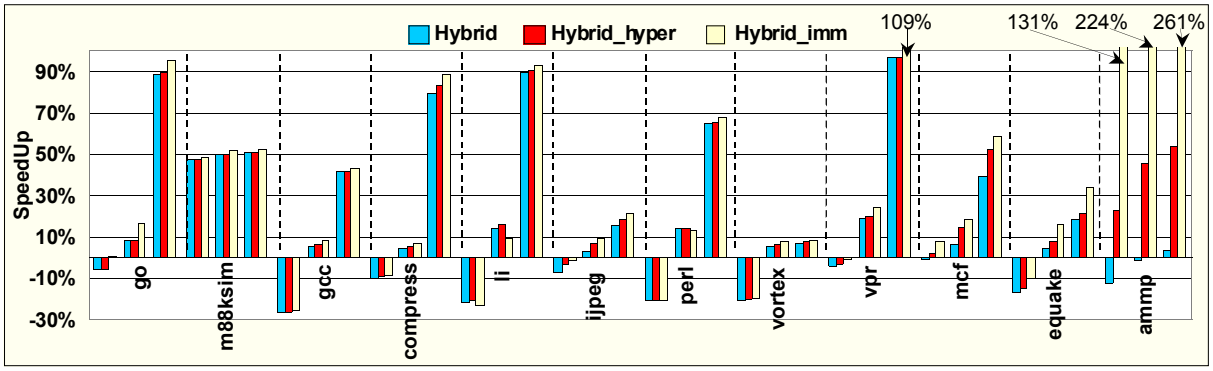


Figure 10. The performance impact of varying the branch prediction accuracy on the hybrid value predictors with the three different update mechanisms. The nine bars of each program are divided into three groups. The first group is for a bimodal branch predictor with 64 entries, the second group is for the hybrid branch predictor described in Table 1, and the last group is for a perfect branch predictor.

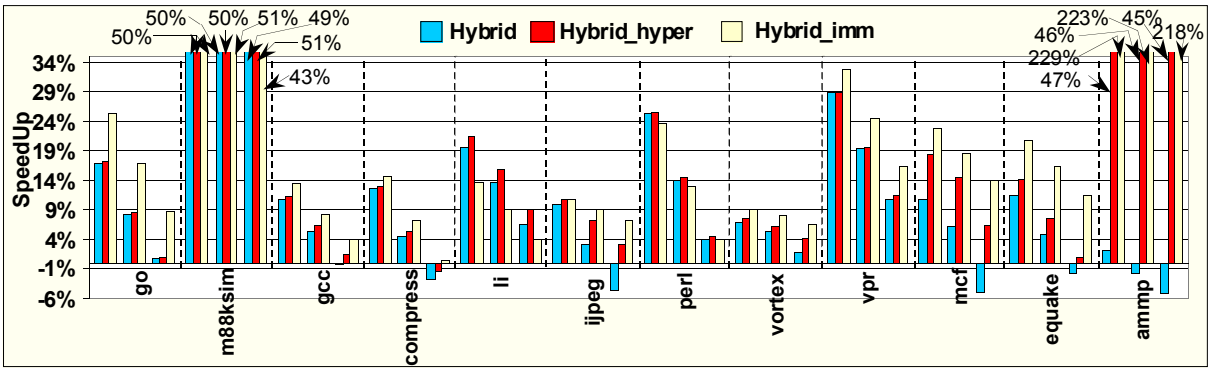


Figure 11. The performance impact of varying the misprediction penalty of the hybrid value predictors with the three different update mechanisms. The nine bars for each program are divided into three groups showing speedups for misprediction penalties of 0, 1, and 2 cycles, from the left.

5. Related Work

Rychlik *et al*. [8] reported that, in some cases, a realistic update mechanism can cut the performance improvement due to value prediction in half compared to the (unrealistic) immediate update mechanism in an 8-issue superscalar processor. Calder *et al.* [9] mentioned the speculative update concept in their selective value prediction study, but they did not propose a detailed solution. Heil *et al.* [12] improved the accuracy of conditional branch prediction by correlating branch directions with data values. In order to efficiently manage the delayed updates of the data values on which subsequent branch predictions depend, they kept a dynamic count of the number of times each static branch instruction was fetched but not yet resolved. Both the last committed data values and the number of outstanding branches were used together to predict the data values to be used by the branch predictor. This scheme worked well for predicting data values for this particular branch predictor. However, they did not propose any corresponding schemes for compensating for delayed updates in value predictors. Recently, Lee and Yew [3] proposed several schemes to handle the delayed update problem in a value predictor by decoupling value prediction from the instruction fetch stage. However, those schemes did not provide a particularly significant improvement to the value prediction techniques.

A similar delayed update problem exists for branch predictors, especially for the global branch history register or the entries of the branch history table, of the two-level adaptive scheme [13]. The speculative update technique was shown to improve some of the branch predictors significantly and has already appeared in some recently announced microprocessors, such as the Alpha 21264 [18]. In this speculative update scheme, the branch history register (or entries) is (are) updated using the predicted

branch outcome immediately after making the prediction in the instruction fetch stage instead of waiting to use the resolved outcome in the write-back or commit stage of the pipeline [13]. Repair mechanisms are needed to fix up the register or tables with the correct branch outcomes when a misprediction occurs. These mechanisms require additional hardware to record the contents of prior branch history registers (or entries) for all unresolved branches. The additional hardware cost is quite low for the global history register since there is only a single register that may need to be repaired. However, the cost is much higher for the local history-based branch predictors since there are many local history entries in the branch history table, and potentially each of them could need to be repaired. Furthermore, recovering from a detected misprediction for a specific outstanding branch requires the proper prior branch history register or entry contents to be extracted from all of the recorded prior branch history register or entry contents. This complex recovery procedure will increase the branch misprediction penalty.

Directly applying the speculative update techniques used for branch predictors to value predictors is not feasible due to the complexity of the required repair mechanism. Similar to what would be needed to repair the state in a local history-based branch predictor, the value predictor would need to store the prior contents of all its entries for all outstanding instances of the corresponding instructions. Since the size of each entry, and the number of the entries in the value prediction table, will be substantially larger than those of the branch history table (BHT), the size of this prior contents table would be prohibitive. Furthermore, the time required to extract the appropriate entry from this large table to fix-up an incorrectly updated entry in the value prediction table would substantially increase the misprediction penalty. Combined with the relatively higher misprediction rates of value predictors compared to branch predictors, this type of speculative update with repair mechanism would likely actually reduce the overall performance. Instead of speculatively updating the value prediction table, our results in this paper demonstrate the effectiveness of tolerating the delay in updating the table using hyperprediction.

6. Conclusion

This paper has proposed and systematically analyzed the *hyperprediction* technique to compensate for delayed updates in value predictors. Using a few extra bits in each entry of the value prediction table and a simple algorithm, the value predictor computes and records the number of *outstanding instances* of each instruction during a program's execution. Accurately recording the number of outstanding instances

builds a partial relationship between the most recently updated value to the entry and the value that should be predicted for the current instruction instance. The value predictor assumes that the values produced by the successive instances of the instruction occur in a predictable sequence. It then makes a prediction for the currently requesting instruction instance based on both the current value and the number of outstanding instances stored in the entry. We showed how this hyperprediction technique could be implemented in several different types of value predictors, including a stride value predictor, a two-level context-based value predictor, and a hybrid value predictor.

We developed a simulator based on the SimpleScalar tool set to evaluate the performance potential of this hyperprediction technique using several integer and floating-point programs from the SPEC95 and SPEC2000 benchmark suites. The speedups obtained when the different value predictors are augmented with the hyperprediction technique are compared in an 8-issue superscalar processor to that obtained when the value predictors use both a realistic update mechanism and an idealized immediate update mechanism. Our simulation results show that, with minimal additional hardware, the hyperprediction technique can consistently improve both the prediction accuracy of existing value predictors and the processor's overall performance. We also found that the relative performance benefit of hyperprediction tends to increase as the processor's instruction-issuing capability increases, and as the misprediction penalty increases. Finally, this technique comfortably tolerates changes in both the branch prediction accuracy and the number of available read/write ports in the value predictor.

Acknowledgements

This work was supported in part by National Science Foundation grants no. EIA-9971666 and CCR-9900605, by KAI Software, a division of Intel America, Inc., and by the Minnesota Supercomputing Institute. D. Lilja was supported by a Fulbright award from the Australian-American Education Foundation during portions of this work.

References

[1] M. Lipasti, C. Wilkerson, and J. Shen. "Value Locality and Data speculation." 7th International Conference on Architectural Support for Programming Languages and Operating System, pages 138-147, October 1996.

[2] M. Lipasti and J. Shen. "Exceeding the DataFlow Limit via Value Prediction." 29th International Symposium on Microarchitecture (MICRO), pages 226-237, December 1996.

[3] S. Lee and P. Yew. "On Some Implementation Issues for Value Prediction on Wide-Issue ILP Processors." International Conference on Parallel Architecture and Compiler Techniques (PACT2000), October 2000.

[4] T. Sato. "Analyzing Overhead of Reissued Instructions on Data Speculative Processors." Workshop on Performance Analysis and its Impaction on Design held in conjunction with 25th ISCA, 1998.

[5] F. Gabbay and A. Mendelson. "The Effect of Instruction Fetch Bandwidth on Value Prediction." 25th International Symposium on Computer Architecture (ISCA), pages 272-281, 1998.

[6] F. Gabbay and A. Mendelson. "Speculative Execution Based on Value Prediction." EE Department Technical Report 1080, Technion-Israel Institute of Technology, November 1996.

[7] B. Rychlik, J. Faistl, B. Krug, A. Kurland, J. Jung, M. Velev, and J. Shen. "Efficient and Accurate Value Prediction Using Dynamic Classification." Technical Report of Microarchitecture Research Team, Department of Electrical and Computer Engineering, Carnegie Mellon Univ., 1998.

[8] B. Rychlik, J. Faistl, B. Krug, and J. Shen. "Efficacy and Performance Impact of Value Prediction." Parallel Architectures and Compilation Techniques, Paris, October 1998.

[9] B. Calder, G. Reinman, and D. Tullsen. "Selective Value Prediction." 26th International Symposium on Computer Architecture, pages 64-74, May 1999.

[10] F. Gabbay and A. Mendelson. "Can Program Profiling Support Value Prediction?" 30th International Symposium on Microarchitecture (MICRO), pages 270-280, December 1997.

[11] K. Wang, M. Franklin. "Highly Accurate Data Value Predictions using Hybrid Predictors." 30th International Symposium on Microarchitecture (MICRO), pages 281-290, December 1997.

[12] T. Heil, Z. Smith, and J. Smith. "Improving Branch Predictors by Correlating on Data Values." 32nd Annual International Symposium on Microarchitecture (MICRO), November 1999.

[13] K. Skadron, M. Martonosi, and D. W. Clark. "Speculative Update of Local and Global Branch History: A Quantitative Analysis." Journal of Instruction-Level Parallelism, volume 2, January 2000.

[14] http://www.motorola.com/SPS/DSP/documentation/MSC8100.html. "StarCore SC140 DSP Core Reference Manual"

[15] D. Burger, T. Austin, and S. Bennett. The Simplescalar Tool Set, Version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin, Madison.

[16] E. Rotenberg, S. Bennett and J. Smith. "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching." 29th Annual ACM/IEEE International Symposium on Microarchitecture, pages 24-35, December 1996.

[17] AJ KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja, "Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research," Workshop on Workload Characterization, International Conference on Computer Design, September, 2000.

[18] R. Kessler, E. Mclellan, and D. Webb. "The Alpha 21264 microprocessor architecture." International Conference on Computer Design, 1998, October 1998.