Improving Value Prediction by Exploiting Both Operand and Output Value Locality

Youngsoo Choi[1], Joshua J. Yi[2], Jian Huang[3], David J. Lilja[2]

[1] - Department of Computer Science and Engineering
[2] - Department of Electrical and Computer Engineering
[3] - Sun Microsystems
Minnesota Supercomputing Institute
University of Minnesota, Minneapolis, MN 55455
youngsc@cs.umn.edu
jjyi, lilja@ece.umn.edu
jian.huang@eng.sun.com

Abstract

Existing value reuse and prediction schemes use a hardware prediction table or reuse buffer to store an instruction's value history based on its program. The result cache [17], on the other hand, has been proposed to exploit operand value locality by reusing the output values produced by any instruction of the same type that has been executed previously using the same input operands. However, due to the non-speculative nature of the result cache, it is effective only for long-latency instructions. In this paper, we extend the reuse-based result cache to support speculative execution. We call this new scheme the Speculative Result Cache (SRC). Our simulations show that value prediction with the SRC alone can produce speedups of 1%-4% for the SPEC95 integer benchmark programs in an 8-issue superscalar processor simulator. We extend the SRC to construct the Combined Dynamic Predictor (CDP) by coupling the SRC with a two-level value predictor [21] and dynamically selecting between the two component predictors. We evaluate how table storage should be partitioned among the two components predictors. We also assign different types of instructions to each component predictor so that the instruction's characteristics match the target locality of the predictor. In our experiments, load instructions are predicted by the SRC while other ALU instructions are predicted by the two-level predictor. This approach outperforms both the SRC and the two-level predictor by themselves, achieving higher effective prediction accuracy and speedups of 4.17% on the average.

Keywords: speculated, instruction reuse, value locality, value prediction, combined dynamic predictor

1. Introduction

To maximize the performance obtained by increasing the issue width of superscalar processors, additional instruction-level parallelism (ILP) must be exposed. Value reuse and value prediction are two methods of exposing additional ILP. In *value reuse* [15], the input operands of the instruction and the corresponding output are saved in the *reuse buffer*, which is indexed by the instruction's address. The next time the instruction is executed, this buffer is checked to determine whether the current operand values match previously seen values. If they do, the stored output value can be used immediately without needing to re-execute the instruction. This approach yields a performance improvement by reducing the effective latency for long-latency instructions to one cycle, which is the time needed to access the reuse buffer. This reduction in latency allows dependent instructions to execute they normally could, which increases the ILP. This approach has been shown to improve performance by 1–5 percent on the SPEC benchmarks [16].

In *value prediction* [2, 8, 18], a prediction for an instruction output is based on the values previously produced for that instruction. The predicted value then is used as an input for any dependent instructions. This prediction allows the dependent instructions to begin speculative execution before the input operands are actually available. When the actual output of the predicted instruction becomes available, it is compared with the predicted value. If the two values match, the prediction is correct and program execution can continue. If the two values differ, then the speculative (dependent) instructions must be squashed and re-executed with the correct input. This approach yields a performance improvement by breaking flow dependences between instructions, which exposes additional ILP.

To be effective, the value reuse buffer must store the input operands and results for as many instructions as possible. Similarly, for each instruction, value prediction needs as large a cache of previously seen values as possible since previously seen values are used for the next prediction. Consequently, both of these approaches require large tables, which will be implemented on the processor die itself. For instance, each entry in the hybrid predictor [13, 21] needs at least 16 bytes to store the last four unique values produced previously by a single instruction. Additionally, these tables must have a low access time in order to produce any performance benefit.

While the total possible number of results for all instructions is enormous, our experiments with the SPECint95 benchmark suite show that the actual result space for all of the instructions for a single benchmark is relatively small. In other words, while the potential range of values that could be produced is enormous, the actual range of values is a much smaller. For some instructions, the output values exhibit a predictable pattern, which is often repeated. For other instructions, different instructions of the same type share the same input operands and therefore produce the same output value. For this paper, the first type of value locality is called *output value locality* while the second is called *operand value locality*.

The *Combined Dynamic Predictor* (CDP) proposed in this paper uses a two-level (TL) predictor [21] – which exploits output value locality – and the *Speculative Result Cache* (SRC) – which exploits operand value locality. The SRC is a combination of value reuse and value prediction. It predicts the instruction's input operands (value prediction) and uses those inputs as an index to retrieve the output for any instruction of that type and input combination (value reuse).

Simulations with the SPECint95 benchmarks show that the CDP outperforms either component predictor, with prediction accuracies and speedups up to 98% and 20%, respectively. Furthermore, the CDP also uses die area more efficiently because each SRC entry is smaller than each TL predictor entry. In addition to dividing the area between the SRC and TL, we evaluate how the instruction prediction should be divided between the two component predictors based on the instruction type. In particular, we use the SRC for load instructions and the TL predicts the outputs for the remaining (ALU) instructions.

The remainder of the paper is organized as follows: Section 2 describes the CDP; Sections 3 and 4 describe the simulation environment, results, and analysis; Section 5 summarizes some related work; and the conclusion is given in section 6.

2. The Combined Dynamic Predictor

The goal of the Combined Dynamic Predictor (CDP) is to exploit the value locality for a single instruction and operand value locality from the execution of any instruction of that type. To accomplish that goal, the CDP uses the two-level (TL) and Speculative Result Cache (SRC) to exploit the output and operand value locality, respectively.

2.1 Speculative Result Cache

Several approaches can be used to predict the result of an instruction before it is executed:

1) If the operands are available and the values of these operands have been previously used by this instruction, the previously stored result can be simply reused.
2) If the operands are available, but the values of these operands have not been previously used by this instruction, the previously stored result cannot be reused. However, if the values of these operands were used by another instruction of the same type and that instruction's result was stored, the result of the previously executed instruction can be used as a prediction for the current instruction.
3) If the operands are not available, but the results for that instruction follow a predictable pattern, a prediction can be made by exploiting that pattern.

Instruction reuse [15] and the value cache [5] handle the first case while existing value predictors [9, 10, 13, 14, 21] handle at least part of the third case. The result cache [17] and the proposed SRC target the second case.

For example, as shown in Figure 1, arrays A, B, and C each have 100 elements. The index variables for these arrays (i1 and i2) are incremented from 0 to 99 in two different loops. The instructions used to increment i1 and i2 are both *addu* instructions. Thus, the results for the i1 addu instruction can be used to predict the results for the i2 addu instruction. Additionally, the *slt* instruction that is used to determine the end of first loop can also be used to predict the outcome of the *slt* instruction used in the second loop. This similar behavior of two different instruction instances allows the branch instruction (*bne*) to resolve earlier than normal.

```
/* C source code */
int i1, i2;
int A[100], B[100], C[100];

for (i1=0; i1 < 100; i1++)
   A[i1] = A[i1]+C[i1]+B[i1];
for (i2=0; i2 < 100; i2++)
    B[i2] = B[i2]+A[i2];
```

```
LOOP1:
…..
addu    $6,$6,1          # update the index
addu    $2,$2,$3
addu    $2,$2,$4
……
slt    $2,$6,99 # compare index to loop-bound
……
LOOP2:
……
addu    $5,$5,1          # update the index
addu    $2,$2,$3
……
slt    $2,$5,99 # compare index to loop-bound
```

Figure 1: Example code that could utilize the Speculative Result Cache

2.1.2 Basic Operation of the Speculative Result Cache

The basic operation of the SRC is similar to other value predictors, except that it is indexed by the values of the instruction's operands instead of its program address. The SRC buffers output values of each instruction that may need prediction in the future, which are indexed by hashing the values of the instruction's input operands. When a decoded instruction cannot be issued because the source instructions that produce its operands have not yet completed, the processor can predict the values that will eventually be produced by the source instructions using the SRC. To perform this prediction, the processor indexes the SRC using the currently available values of the operands of the source instructions. If predictable values are found in the SRC, the instruction can be speculatively issued for execution using these predicted values for its input operands.

While the existing predictors exploit the value locality of an instruction, the SRC is capable of detecting the value locality of an operation with the same operands. Although the addresses of two instructions are different, they access the same entry of SRC and predict with high accuracy as long as their access pattern and operands are the same.

2.1.3 The SRC Implementation

Instructions in the execution window are decoded before being issued, which allows dependence chains to be constructed before the instructions reach the issue stage. Once an instruction is decoded, the source operands are available to access the SRC. Figure 2 shows the SRC structure and how it is accessed in a superscalar processor pipeline.

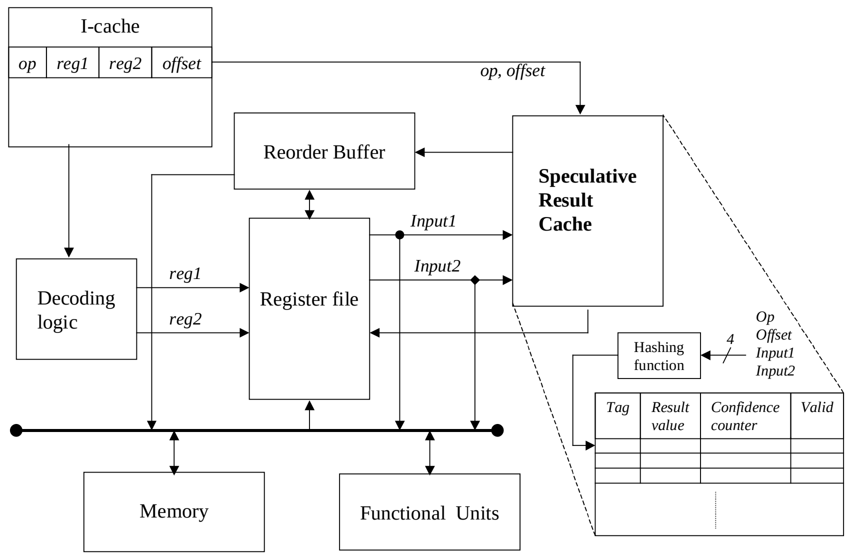


Figure 2: An implementation of the SRC in a superscalar processor

Each SRC entry contains a tag, result, confidence counter, and valid bits. The size of each entry is about 9 bytes, which is half the size of the two-level value predictor [21]. Currently, the SRC stores only a single result. Multiple output values could be stored for better performance with a corresponding increase in cost and indexing complexity. The 4-bit confidence counter is incremented or decremented by a fixed amount depending on the result of the prediction when each instruction is retired. When an entry is accessed, the confidence value is compared with the confidence threshold required to make a prediction decision. If the valid bit is not set, the SRC will signal the issue stage to not predict the output value since the value stored in the SRC is invalid. This situation occurs when the hashed operand values have not been previously seen. By the time an instruction reaches the issue stage, the results of the SRC will be available. If this instruction cannot be issued due to data dependences, the issue logic can either speculatively issue the instruction with the predicted operand values from the SRC or wait until the actual operand values become available. While the second choice forces the existing dependences to be completely resolved (and thus eliminates the chance to increase the instruction issue rate), it also eliminates the chance of a potentially expensive misprediction.

The operand values are XORed to create an index into a section of the table determined by the instruction type. Figure 3 illustrates how the index is formed and how it is used to access the SRC.

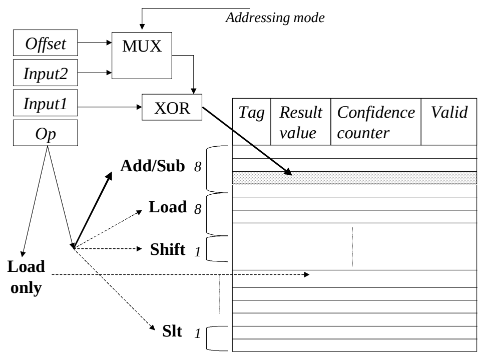


Figure 3: SRC Index Formation

Since the MIPS-like architecture simulated in this paper supports two addressing modes for loads, *register+register* and *register+offset*, the second operand is either the second register value or the offset value.

Since different instruction types produce different output results even when the input operands are identical, the SRC is partitioned into different sections based on the instruction type specified in the op-code field. As a result, to minimize its cost while maximizing its performance, the SRC stores results only for instructions that have the highest frequencies of occurrence.

Furthermore, from these high-frequency instructions, only the instructions with the highest frequency times latency products (FLP) are stored. This product attempts to capture the instructions that have the largest impact on performance. Using the parameters in Table 2, the following table shows the instruction frequency, effective latency, and the FLP for each candidate instruction type for the SPECint95 benchmarks (benchmarks statistics given in Table 3).

Instruction Type	**Frequency**	**Effective Latency**	**Frequency * Latency**
INT Add/Sub, etc.	0.635816	1	0.6358
Load	0.229451	1.123	0.2577
Store	0.133009	1	0.1330
INT Multiplication	0.001602	3	0.0048
FP Add/Sub, etc.	0.000115	2	0.0002
FP Division	4.24E-06	12	0.0001
INT Division	2.28E-06	20	0.0000
FP Multiplication	7.5E-09	4	0.0000

Table 1: SPECint95 Instruction Frequency Time Latency Products. (The effective latency for the load instructions is simply the weighted sum of the products of the hit rate and latency for each level in the memory hierarchy.)

As seen in Table 1, the INT ALU, load, and store instructions have the highest FLPs. Since the optimization of the store instructions will not yield any performance benefit, stores are excluded from the SRC. Furthermore, since there are many different types of INT ALU instructions, only the highest frequency INT ALU instructions should be stored in the SRC. Long-latency instructions such as FP division, INT division, and FP multiplication have low FLPs and therefore are not included in the SRC. The inclusion of those instruction types will not yield a performance improvement since their low instruction frequencies do not significantly contribute to the overall execution time. Since value prediction only applies to the register-writing instructions, and since the add/sub, logical, shift-left, shift-right, and set-less-than (slt) instructions, in addition to loads, are the vast majority of register-writing instructions, these instructions are included in the SRC.

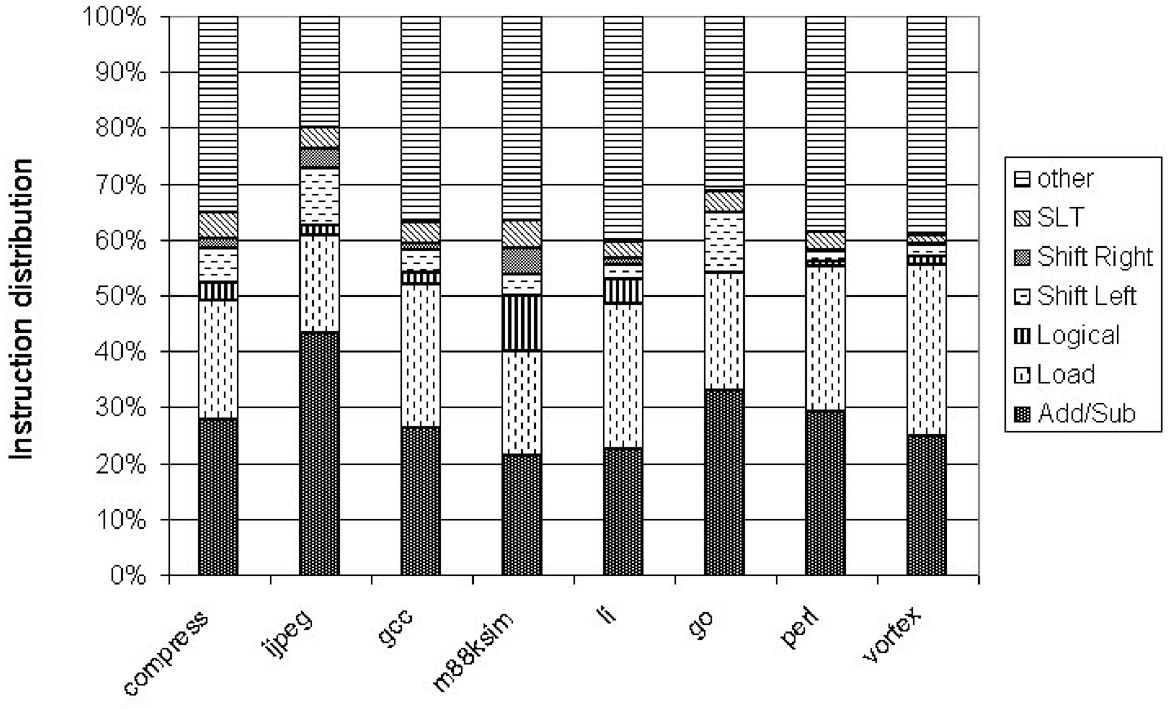


Figure 4: Frequency of register-writing instructions

The SRC entries are allocated according to their relative instruction frequencies. Using the instruction distribution for the SPECint95 benchmarks, shown in Figure 4, we find an 8:8:1:1:1:1 ratio for load:add/sub:logical:shift-left:shift-right:slt. For example, in order to build a 24KB SRC, 2560 possible entries are needed. Of these available entries, 1024 entries are assigned to both load and add/sub instructions while 128 entries are assigned to each of the other four types.

2.2 Implementation of the Combined Dynamic Predictor

While the SRC exploits operand value locality, most of the existing value predictors [10, 13, 21] exploit output value locality based on the instruction address. That is, they simply assume that each instruction will produce regular outputs for each instance. Consequently, saving these results will help predict what value that instruction will likely to produce next. Since the SRC and these predictors exploit different types of value locality, combining the SRC with such a value predictor may produce higher prediction accuracy, and therefore, better performance. This new predictor is called the Combined Dynamic Predictor (CDP).

The two-level predictor presented in [21] detects a periodic recurrence of values. The last four unique values produced by an instruction are stored in each entry of the value history table (VHT). An additional eight bits are used for the Value History Pattern(VHP) which records the order in which the four values were produced. The VHP then indexes the second level of the predictor, the Pattern History Table (PHT), which is a table of confidence counters for each stored value. Of the four values, the one with the highest confidence level is picked as the predicted value. If that confidence level is higher than the confidence threshold, the predictor returns the value corresponding to that confidence counter as the predicted result. If the confidence level is lower than the confidence threshold, no prediction is returned.

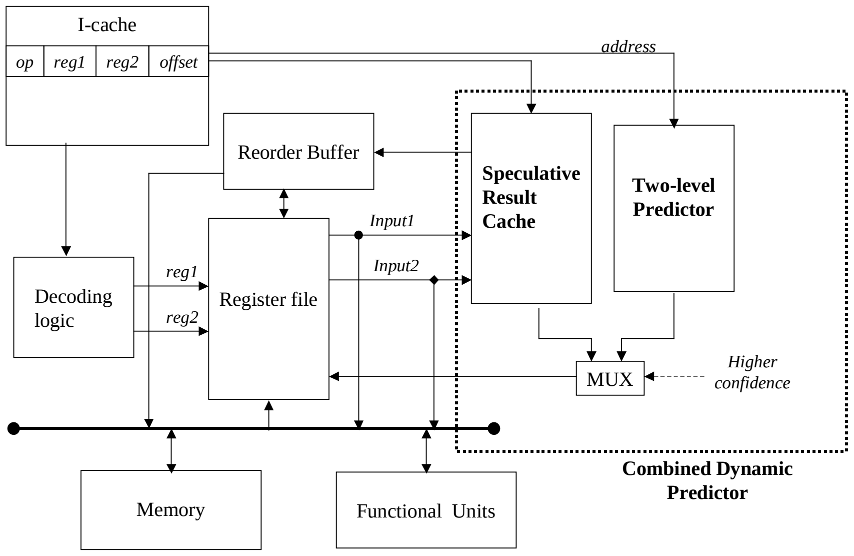


Figure 5: The Combined Dynamic Predictor (CDP).

When the CDP is accessed, both of the component predictors (SRC or two-level) go to work to (potentially) produce predicted values. The CDP then selects the predicted value from the component predictor with the higher confidence level. Figure 5 shows the CDP. Note that the SRC is indexed with the operand values while the two-level predictor is indexed with the instruction's address.

3. Simulation Environment

The simulations are based on the sim-outorder simulator from the SimpleScalar Tool Set Version 3.0 [1]. We modified it to include the SRC and the TL predictor. The base processor model is an 8-issue superscalar processor with out-of-order execution. The predicted output values are available one cycle after the predictor is accessed and are forwarded to dependent instructions so that they can be issued simultaneously. Table 2 shows the key base processor model parameters.

Fetch/Decode/Issue Width	8
Reorder Buffer Size	128
L1 Data Cache	4-way set-associative, 64KB
L1 Data Cache Hit Latency	1 cycle
L2 Data Cache	4-way set-associative, 1024KB
L2 Data Cache Hit Latency	9 cycles
Memory Width	32 bytes
Memory Access Latency	60 cycles
Integer ALUs	16
Integer Multipliers	4

Table 2: Base processor model parameters

The value prediction table is updated when a predicted instruction reaches the writeback stage, where the predicted value is compared with the real value. If the prediction is correct, the dependent instructions (if done executing) can be retired with the predicted in the same cycle. If a value is mispredicted, however, the dependent instructions are selectively re-issued to the functional units, based on availability. The actual misprediction penalty depends on the number of instructions to be re-issued. A bus between the reorder buffer and the functional units is used to directly dispatch the misspeculated instructions. The bus width was configured to match the issue width. Thus, as long as the number of misspeculated instructions is less than the issue width, they can be re-issued within one cycle, as presented in [13]. If it is greater than the issue width, then the misprediction penalty is the number of cycles it takes to re-issue all the dependent instruction. Both the load/store queue and the reorder buffer were configured to have 128 entries.

Both of the CDP component predictors use a four-bit confidence counter. The SRC counter increments by 3 for a correct prediction and decrements by 1 for an incorrect prediction. The SRC confidence threshold is 8. The TL predictor increments by 2 for a correct prediction and decrements by 1 for an incorrect prediction. The two-level confidence threshold is 13. The TL increment/decrement and the TL confidence thresholds were based on the values given in [13, 21] and refined by trial and error while trying to maximize the performance. The SRC increment/decrement and the SRC confidence thresholds were chosen by trial and error while trying to maximize the performance.

Benchmark	**Millions of Instructions**	**Percentage of Loads**	**Percentage of ALU**
compress	153.3	20.64	41.98
ijpeg	553.4	17.63	62.39
gcc	973.2	25.69	37.38
m88ksim	120.1	18.98	44.40
li	173.3	25.92	33.65
go	548.1	21.12	47.62
perl	1891.5	26.14	35.26
vortex	2120.1	30.75	30.00

Table 3: Characteristics of the benchmark programs showing the instruction count and the percentage of Load and ALU Instructions. These two categories were the only types of instructions that were predicted.

This paper used the following eight programs from SPECint95 benchmark suite: compress, gcc, go, ijpeg, li, m88ksim, perl, and vortex. All the benchmarks were compiled using gcc 2.7 with '-O2' flag. The simulations used the train input set and some instructions in the initialization phase were skipped in order to concentrate on the main body of the programs. Table 3 shows the relevant benchmark statistics.

4. Performance Evaluation and Analysis

4.1 Resource Allocation Between Component Predictors

One evaluation goal was to determine how to partition the available chip area between the two component predictors. These experiments assumed that die space for an additional 64KB of storage was available, plus some additional area for control logic. Several configurations that varied the size ratio between the component predictors were evaluated. For comparison purposes, another configuration utilized the available space to make a larger L1 data cache (128 KB).

SRC size	TL Predictor size	Total size	SRC %	Configuration Name
0 KB	47 KB	47 KB	0 %	47k_TL
11 KB	47 KB	58KB	19%	S11k(type)+TL47k(type)
22 KB	22 KB	44KB	50%	S22k(type)+TL22k(type)
45 KB	11 KB	56KB	80%	S45k(type)+TL11k(type)
45 KB	0 KB	45 KB	100 %	S45k(type)

Table 4: Simulated CDP configurations. The SRC % column is the percentage of the total area devoted to the SRC. The configuration names, or similar variants, are used in Figures 9 – 12. The (type) field corresponds to what type of instruction the predictor targets; the possibilities are: all (default), load, and others (excluding loads).

Table 4 shows the five CDP configurations. These configurations represent an SRC allocation of 20%, 50%, and 80% of the designated 64KB area, with the remaining area allocated to the TL predictor. Evaluating these combinations indicates how the available resources should be partitioned among the component predictors while maintaining a constant overall predictor size. These configurations were chosen to be the power-of-two sizes that most closely approximate the desired distribution of resources.

4.2 Impact of Different CDP Configurations

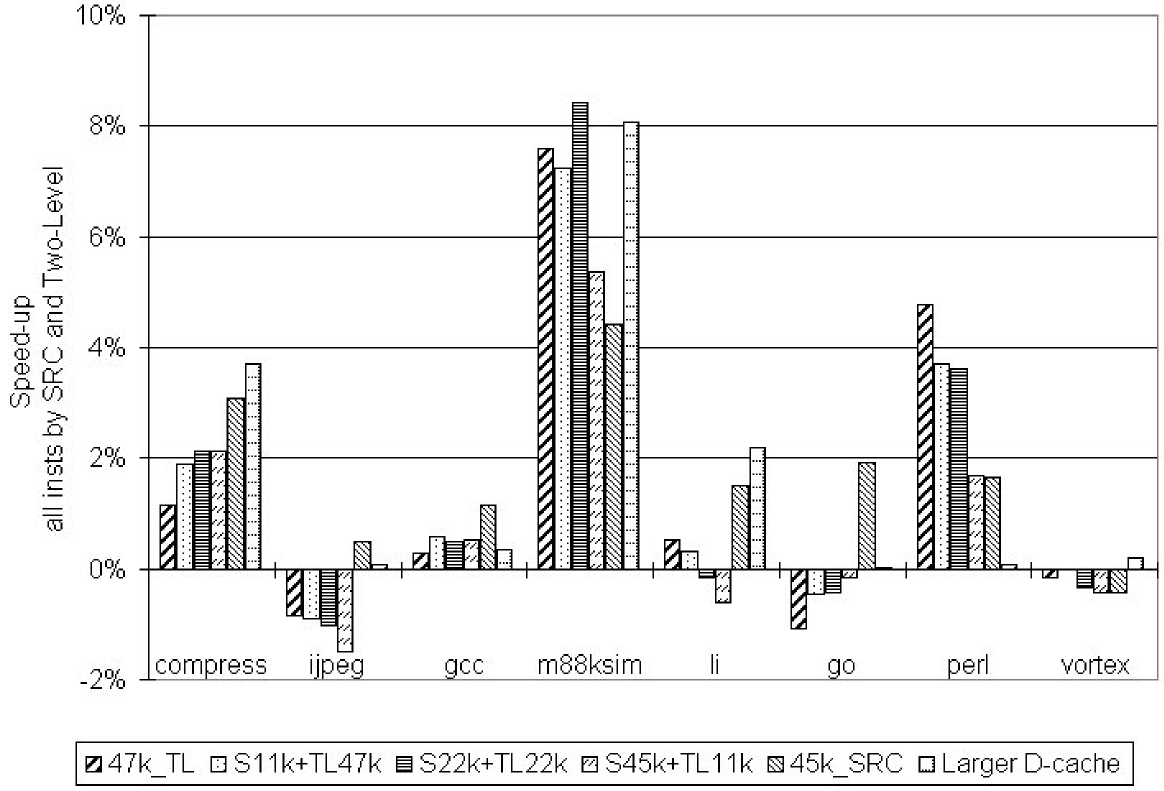


Figure 6: IPC Speedups for selected configurations. The 47k_TL configuration corresponds to a standalone 47k two-level predictor. The S11k+TL47k, S22k+TL22k, and S45k+11k configurations correspond to the CDP predictors with 11k, 22k, and 45k allocated to the SRC component predictor and 47k, 22k, and 11k, respectively, allocated to the two-level predictor. The 45K_SRC configuration corresponds to a standalone 45k SRC. In this figure, the SRC and TL predictor both predict all types of instructions.

The Combined Dynamic Predictor (CDP) produces speedups up to 8.4% (Figure 6). For most cases except vortex, the SRC outperformed the baseline, while the CDP showed –1.5% to 8%

speedup, depending on the configuration. The biggest gain occurred in m88ksim (8.4% speedup) for the S22K + TL22k configuration, while the larger D-cache produced slightly lower performance.

For some benchmarks (go, ijpeg, li, and vortex), due to the relatively low performance of the two-level predictor, the CDP does perform worse than the other configurations. In compress, m88ksim, gcc, and perl, however, the speedup of the S11k + TL47k (11k allocated to the SRC and 47k to the TL predictor) configuration is higher than those of the baseline and larger D-cache. As expected, the CDP does a good job capturing both types of value locality. This is shown by, for most benchmarks, using both component predictors together produces better performance than when a single predictor is used.

Figure 7 illustrates the performance prediction accuracy for the different configurations. The performance prediction accuracy is defined as the number of correct predictions divided by the total number of instructions executed. This metric attempts to quantify the impact that the CDP has on the entire program. If the performance prediction accuracy is high, then the predictor is correctly predicting a high fraction of the total number of instructions, which should produce noticeable speedups.

The performance prediction accuracy of the S11k+TL47k is quite respectable, ranging from 15% - 48%**,** and is higher than other configurations in most of the benchmarks. These performance prediction accuracies show that the CDP predictor accurately predicts the values for a significant number of the benchmarks. An increase in the performance prediction accuracy does not directly translate into an increase in speedup, however, because the speedup depends on many other factors, such as latency, numbers of dependent instructions, etc.

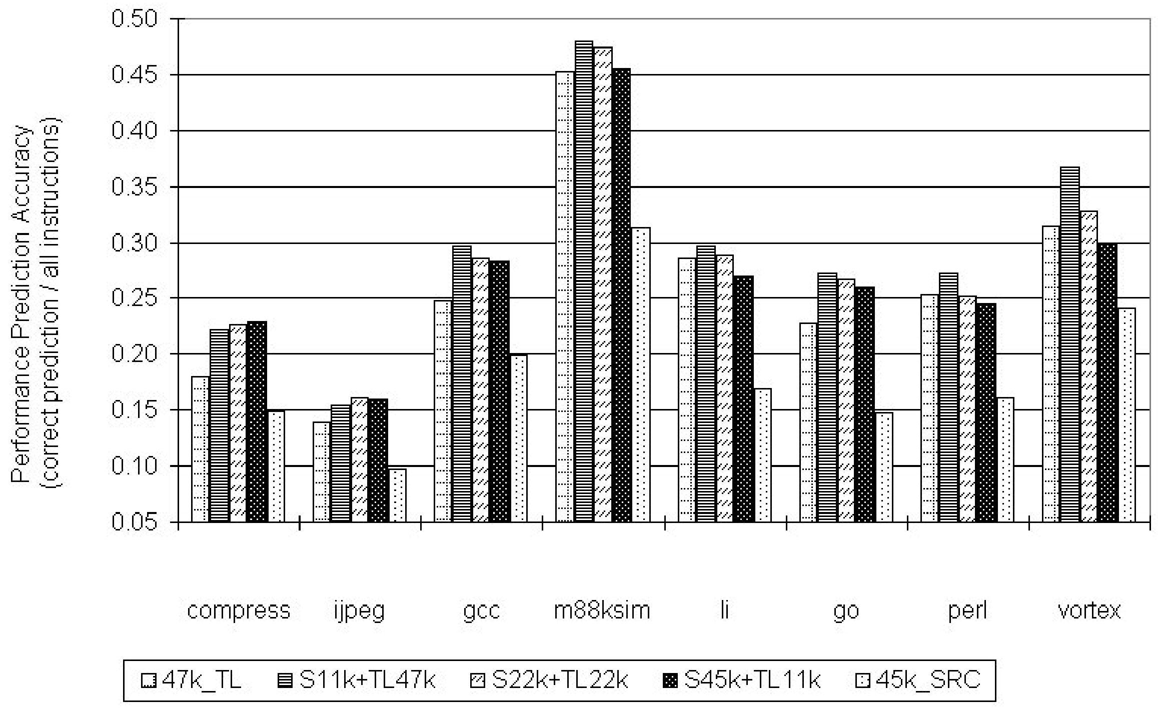


Figure 7: Performance Prediction Accuracy of the SRC. The S11k+TL47k, S22k+TL22k, and S45k+11k configurations correspond to the CDP predictors with 11k, 22k, and 45k allocated to the SRC component predictor and 47k, 22k, and 11k, respectively, allocated to the two-level predictor. The 45K_SRC configuration corresponds to a standalone 45k SRC. In this figure, both the SRC and TL predictor predict all instructions.

4.3 Impact of Predicting Different Types of Instructions

Since the SRC predicts the output values right after an instruction is decoded, and since all INT ALU instructions have a single cycle latency, there does not seem to be much benefit in predicting the outputs for these instructions (their actual results are available in a single cycle later anyway), unless there are many dependent instructions and these dependent instructions cause resource conflicts. Therefore, what is the performance impact of the using the SRC to predict only longer latency instructions, such as loads, while the TL predictors predicts the remaining instructions (selected INT ALU instructions) or all types of instructions.

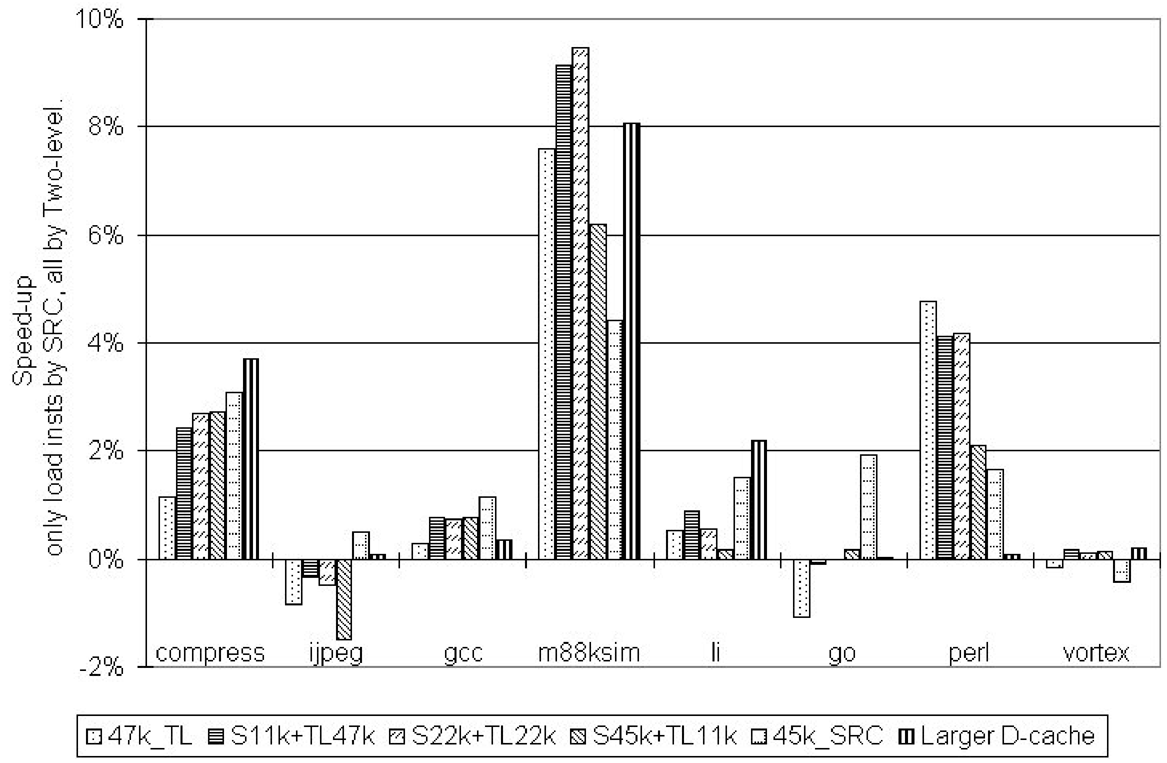


Figure 8: IPC Speedups for selected configurations. The S11k+TL47k, S22k+TL22k, and S45k+11k configurations correspond to the CDP predictors with 11k, 22k, and 45k allocated to the SRC component predictor and 47k, 22k, and 11k, respectively, allocated to the two-level predictor. The 45K_SRC configuration corresponds to a standalone 45k SRC. In this figure, the SRC predicts only loads while the TL predictor predicts all instructions.

For each configuration, the speedups for most benchmarks in Figure 8 are slightly higher than those in Figure 6 (SRC predicts all instructions). Especially, the performance of *li* and *vortex* using CDP configurations turns out to be higher than that of the baseline architecture since those two programs include more load instructions than others and they are more accurately predicted by the SRC. Thus, limiting the SRC to loads while using the TL predictors to predict all instructions improves the performance. Furthermore, since limiting the SRC to predict only loads improved the performance, we extended this approach to limit the TL predictor to predict only INT ALU instruction further improve the performance. Figure 9 shows the results for this configuration.

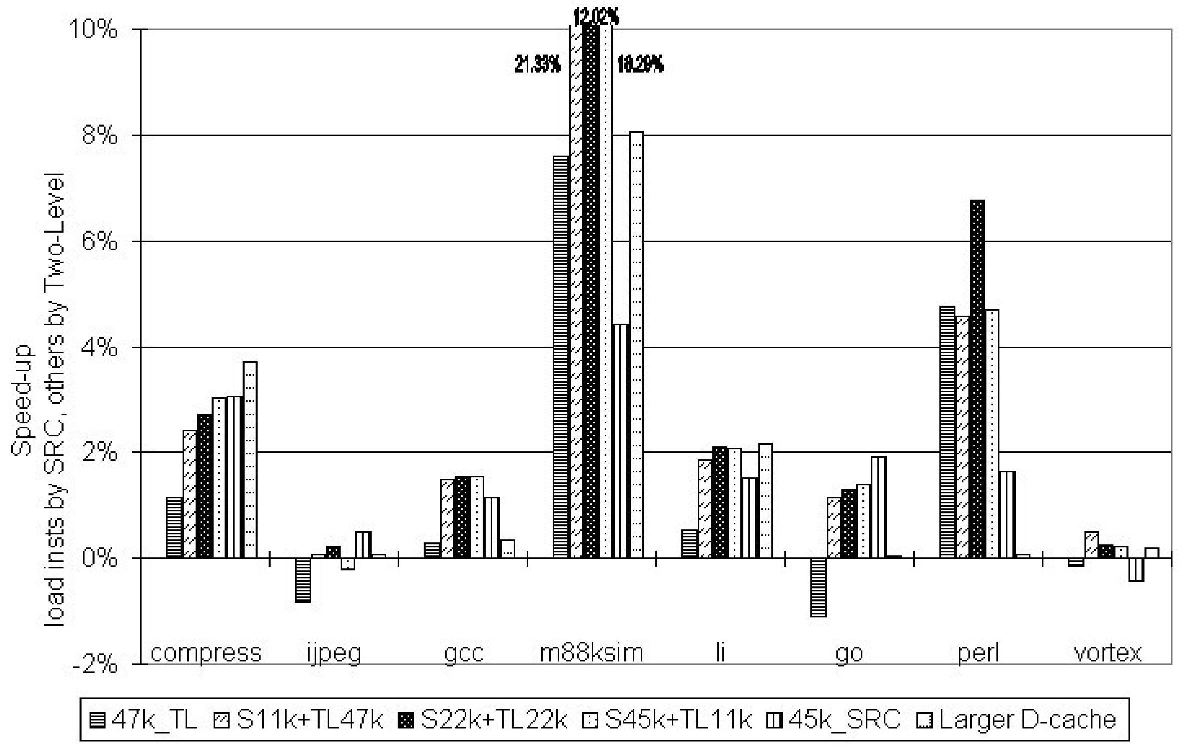
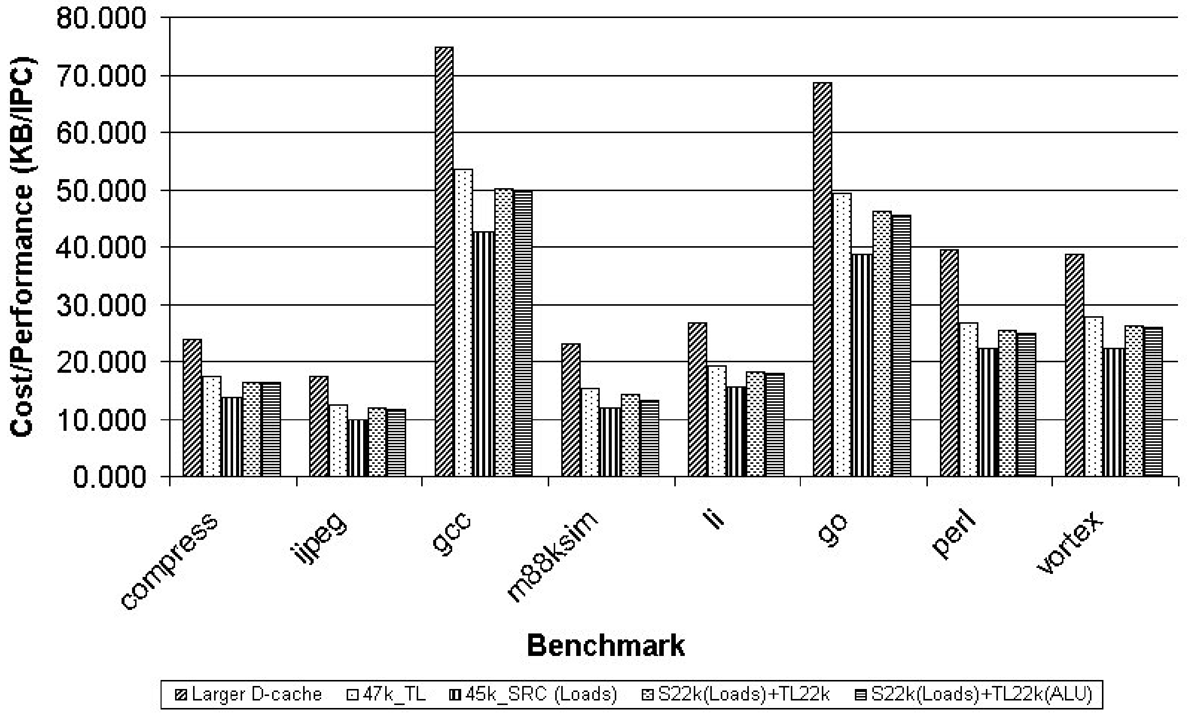


Figure 9: IPC Speedups for selected configurations. The S11k+TL47k, S22k+TL22k, and S45k+11k configurations correspond to the CDP predictors with 11k, 22k, and 45k allocated to the SRC component predictor and 47k, 22k, and 11k, respectively, allocated to the two-level predictor. The 45K_SRC configuration corresponds to a standalone 45k SRC. In this figure, the SRC predicts only loads while the TL predictor predicts remaining instructions (INT ALU).

The speedups for all benchmarks except ijpeg in Figure 9 are slightly higher than those in Figure 8 (SRC predicts loads) and Figure 6 (SRC predicts all instructions). S11k+TL47k configuration produced the best performance, speedup of 4.17% on the average, ranging from 0.1% up to 21% depending on the benchmarks. On the other hand, S22k+TL22k configuration outperforms others in overall, higher than larger D-cache configuration except *li* and *compress*. By limiting the predicted instructions based on their types, the results showed that CDP configurations worked better than the baseline regardless of value locality pattern and instruction types.

The performance is due to one of two reasons: 1) more entries in TL available to the INT ALU instructions, which decreases the number of conflicts while enhancing the accuracy of both predictors, 2) the SRC does a better job of capturing the value locality for loads while TL does a better job of capturing the value locality of INT ALU instructions. The second reason would imply that the loads exhibit more operand value locality than output value locality while the reverse would be true for INT ALU instructions.

4.4 The Cost and Performance Tradeoff



Figures 10: The cost performance tradeoff for selected predictors (Lower cost/performance = better cost performance tradeoff). The 47k_TL configuration corresponds to a standalone 47k two-level predictor. The 45K_SRC configuration corresponds to a standalone 45k SRC that only predicts loads. In the S22k(Loads)+TL22k configuration, the SRC corresponds predicts only loads and the TL predictor predicts all instructions. In the

S22k(Loads)+TL22k(ALU) configuration, the SRC corresponds predicts loads and the TL predictor predicts only INT ALU instructions.

One consideration concerning value prediction is the table size. While an extremely large table may show good speedup, its large size makes it impractical to implement. On the other hand, if might be worthwhile to implement a very small predictor, like the CDP or SRC, that provides an incremental improvement in IPC. The cost of this predictor (measured in area) may justify its implemented in spite of the small IPC improvement.

Figure 10 shows the cost-performance tradeoff where the predictor area in KB represents the cost while IPC represents the performance. The units for the cost performance tradeoff are KB/IPC. Therefore, the smaller the bar, the better the cost performance tradeoff. Configurations have a lower cost performance tradeoff if the area to implement the predictor is small, the IPC is high, or a combination of both.

The last three configurations (45K_SRC, S22k(Loads)+TL22k, and S22k(Loads)+TL22k(ALU)) are shown because they have about the same area as the TL predictor. For all benchmarks, the larger d-cache configuration has the worst (highest) cost performance tradeoff while the SRC by itself (45k_SRC) has the best (lowest) cost performance tradeoff. While the TL predictor cost performance tradeoff is better than that of the d-cache for all benchmarks, its cost performance tradeoff is worse than the cost performance tradeoff for both CDP configurations for all benchmarks. These results indicate that the CDP, due to the SRC component predictor, is a better use of area than the TL predictor alone while the its performance is similar or better than that of the TL predictor.

Furthermore, for the different CDP instruction allocation prediction configurations, on average, the cost performance tradeoff from best to worst is: SRC(Load)+TL(ALU), SRC(Load)+TL(All), and SRC(All)+TL(All). In the first configuration, the SRC predicts only loads while the TL predicts only INT ALU instructions; in the second configuration, the SRC predicts only loads while the TL predictors all instructions; finally, in the last configuration, both predictors predict all instructions. These results are independent of the area allocated to either component predictor. Therefore, these results indicate that the SRC(Load) + TL(ALU) is the most effective (i.e. best cost performance tradeoff) implementation of the CDP, regardless of how much area is allocated to each predictor.

Finally, it is worth mentioning that the 11k SRC had the best cost performance tradeoffs. The cost performance tradeoffs for these two configurations ranged from 2.489 (ijpeg) to 10.684 (gcc) when the SRC predicts loads only and 3.129 (ijpeg) to 13.414 (gcc) when the SRC predicts for all instructions. These cost performance tradeoffs are approximately 4 and 5 *times* better for the two configurations when compared against the TL predictor. This result shows that even a small SRC can provide similar performance benefits as the two-level predictor, at a small fraction of the area.

5. Related Work

Sodani and Sohi [15] investigated the potential of value reuse with a reuse buffer that stored the inputs and outputs of each executed instruction. In addition to reusing the results from previous instructions, they tried to reuse the results for wrong-path execution. This approach yielded speedups of 1–5 percent for the SPECint benchmarks [16]. Huang and Lilja [6] extended instruction-level value reuse to basic block-level reuse. This approach dynamically detects a dependence chain and compares all of the inputs of the basic block to make a prediction. This block-level reuse produces slightly greater performance improvements than instruction-level reuse since a larger number of instructions reused. Harbison [5] used a value cache, indexed by the instruction address, to store the instruction's result, dependence information, and the execution phase. The value cache produced speedups of about 2.0 on a stack machine by eliminating hardware-level common subexpressions.

Richardson [17] proposed a scheme that targeted floating-point instructions. This scheme used a tabulation method to store the return values of function calls with no side-effects using software and a result cache to store long-latency instruction results. The result cache was indexed by XORing the most significant bits of the instruction's operands. This mechanism reduced the execution time of some of the SPECfp82 benchmarks by 15%–44%. However, since this result cache requires that the actual operands be used to produce the index and these operands are not available until right before the instructions are issued, its best performance is for long-latency operations, such as floating-point division.

Tullsen and Seng [19] proposed a method that saves valuable chip area by using the values that are already available in the register file. They found that at least 75% of the time, the value loaded from memory is either already in the register file, or was recently there. For this approach, the instruction uses the value currently in the destination register as a prediction for the new value. Compiler support is used to increase the prediction opportunities for this method. This method produced speedups of up to 11% for the SPECint95 benchmarks and up to 13% for the SPECfp95 benchmarks.

Lipasti *et al* [8, 9] investigated the potential of improving ILP by using value prediction for load instructions. With limited additional hardware, this approach produced speedups of 4%–12% on the DEC 21164 and PowerPC 620 processors. While several predictors (finite-context, two-level, and TL [10, 13, 14, 21]) have been proposed to improve the prediction accuracy, Gonzalez and Gonzalez [4] suggest that the increase in ILP due to value prediction is limited due to the misprediction penalty.

The SRC approach differs from value reuse since the operands are speculatively used to index the SRC instead of waiting for the actual input operands. As a result, because the speculative inputs are available before the actual inputs, the SRC effectively extends value reuse to shorter latency instructions than [5, 6, 15, 17]. The SRC shares three similarities with value prediction. First of all, both approaches make a prediction for one or more input operands. Secondly, after making a prediction for an input, the following dependent instructions can be speculatively executed. Finally, both approaches verify the prediction and restore the correct architectural state if a misprediction occurs. However, the SRC and value predictor have one key difference. Each entry in the SRC is not implicitly tied to a specific instruction (via the instruction address) as in value prediction, but can make a prediction for any instruction that matches of the input operands. Since the CDP is the combination of the SRC and a two-level value predictor, the comparison to value reuse and value prediction is the same as the SRC.

6. Conclusion

Most existing value predictors [8, 10, 13, 21] exploit only instruction output value locality. On the other hand, the result cache [17], however, exploits operand value locality by reusing the values produced by any of the previously executed instructions of the same type with the same operands. Due to its non-speculative nature, the result cache can be accessed only when the instructions are ready to be issued. As a result, since most integer instructions in a superscalar processor complete within a single cycle, the result cache can provide only limited performance improvement for integer programs.

This paper presented the Combined Dynamic Predictor (CDP), which is composed a conventional two-level (TL) value predictor [21] and the *Speculative Result Cache* (SRC). The SRC extends the result cache to speculate on the input operands. As a result, the SRC exploits operand value locality while the two-level value predictor exploits output value locality. Simulations show that the SRC can produce speedups up to 7% for an 8-issue out-of-order superscalar processor for the SPECint95 benchmarks. The reason the TL value predictor was used in the CDP was that it has good prediction accuracy and substantial performance benefit.

The CDP exploits both operand value locality and output value locality by dynamically selecting between the predicted values produced by its two component predictors, always choosing the predicted value the value with the higher confidence level. The CDP can produce speedups of up to 20% on a 8-issue out-of-order superscalar processor for the SPECint 95 benchmarks. The CDP outperforms either component predictor when each is used separately.

This paper further evaluated directing instructions based on their type to one or the other component predictor. To compensate for the late prediction of the SRC in the pipeline, the SRC was configured to predict load instructions while the two-level predictor predicted other ALU instructions. This configuration produced the best performance of all the configurations tested, resulting in up to 20% speedup.

From a cost performance tradeoff standpoint, the cost-performance tradeoff analysis revealed that the CDP is more effective than the two-level predictor and that allocating the load instructions to the SRC while allocating the remaining instructions to the two-level predictor is the most effective CDP implementation. Finally, the results show that even a small, “standalone” SRC can provide a modest performance increase while using a small area.

In conclusion, the best value prediction approach for integer programs is obtained by dynamically selecting between a predictor that exploits operand value locality and another that exploits output value locality. Furthermore, assigning different types of instructions to each predictor outperforms previous allocation strategies since different types of instructions show different types of value locality behaviors.

7. Acknowledgements

The authors would like to thank Robert Glamm for his many helpful discussions and comments. This work was supported in part by National Science Foundation grants no. EIA-9971666 and CCR-9900605, and by the Minnesota Supercomputing Institute.

8. References

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

[2] F. Gabbay and A. Mendelson, “Using Value Prediction to Increase the Power of Speculative Execution Hardware”, In ACM Transactions on Computer Systems, Vol. 16, No. 3, August 1998.

[3] B. Calder, G. Reinman, and D. Tullsen, “Selective Value Prediction”, In Proceedings of the 26th International Symposium on Computer Architecture (ISCA-26), Atlanta, May 1999.

[4] J. Gonzalez and A. Gonzalez, “The Potential of Data Value Speculation to Boost ILP”, Proceedings of the 1998 international conference on Supercomputing (ICS'98), July 1998.

[5] S. Harbison, “An architectural alternative to optimizing compilers”, In Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 1982.

[6] J. Huang and D. Lilja, “Exploiting Basic Block Value Locality with Block Reuse”, In the Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA-5), January 1999.

[7] S.J. Lee, “Several data value predictors”, http://www.cs.wisc.edu/mscalar/ss_misc.html

[8] M. Lipasti, C. Wilkerson, and J. Shen, “Value Locality and Load Value Prediction”, In the Proceedings of 7th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS VII), October 1996.

[9] M. Lipasti and J. Shen, “Exceeding the Dataflow Limit via Value Prediction”, In the Proceedings of 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-29), December 1996.

[10] T. Nakra, R. Gupta, and M.L. Soffa, “Global Context-Based Value Prediction”, In the Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA-5) January 1999.

[11] S. Oberman and M. Flynn, “On Division and Reciprocal Caches”, Stanford University Technical Report, CSL-TR-95-666, April 1995.

[12] Y. Patt, S. Patel, M. Evers, D. Friendly, and J. Stark, “One Billion Transistors, One Uniprocessor, One Chip”, In IEEE Computer, vol. 30, No. 9, September 1997.

[13] B. Rychlik, J. Faistl, B. Krug, and J. Shen, “Efficacy and Performance Impact of Value Prediction”, Parallel Architectures and Compilation Techniques, October 1998.

[14] Y. Sazeides and J. Smith, “The Predictability of Data Values”, In the 30th Annual International Symposium on Microarchitecture (MICRO-30), December 1997.

[15] A. Sodani and G. Sohi, “Dynamic Instruction Reuse”, In the 24th International Symposium on Computer Architecture (ISCA-24), June, 1997.

[16] A. Sodani and G. Sohi, “Understanding the Differences Between Value Prediction and Instruction Reuse”, In the 31st International Symposium on Microarchitecture (MICRO-31), November-December 1998.

[17] S. Richardson, “Caching function results: Faster arithmetic by avoiding unnecessary computation”, Technical report SMLI TR-92-1, Sun Microsystems Laboratories, September 1992.

[18] J. Smith and S. Vajapeyam. "Trace Processors: Moving to Fourth Generation

Microarchitectures". In IEEE Computer, Col. 30, No. 9, September 1997.

[19] D. Tullsen, and J. Seng, “Storageless Value Prediction using Prior Register Values”, In Proceedings of the 26th International Symposium on Computer Architecture (ISCA-26), May 1999.

[20] G. Tyson and T. Austin, “Improving the Accuracy and Performance of Memory Communication Through Renaming”, In the Proceedings of the 30th Annual International Symposium on Microarchitecture(MICRO-30), December 1997.

[21] K. Wang and M. Franklin, “Highly Accurate Data Value Prediction using Hybrid Predictors”, In the 30th Annual International Symposium on Microarchitecture (MICRO-30), December 1997.