Received: 24 February 2017 | Revised: 22 June 2017 | Accepted: 23 June 2017

DOI: 10.1002/cpe.4250

WILEY

RESEARCH ARTICLE

A survey of value prediction techniques for leveraging value locality

Sparsh Mittal

Indian Institute of Technology, Hyderabad, India

Correspondence
Sparsh Mittal, IIT Hyderabad, Kandi, Sangareddy 502285, Telangana, India.
Email: sparsh0mittal@gmail.com

Funding information
Science and Engineering Research Board (SERB), India, Grant/Award Number: ECR/2017/000622

Summary

Value locality (VL) refers to recurrence of values in a memory structure, and value prediction (VP) refers to predicting VL and leveraging it for diverse optimizations. VP holds the promise of exceeding true-data dependencies and provide performance and bandwidth advantages in both single- and multi-threaded applications. Fully exploiting the potential of VL, however, requires addressing several challenges, such as achieving high accuracy and coverage, reducing hardware and latency overheads, etc. In this paper, we present a survey of techniques for leveraging value locality. We categorize the research works based on key parameters to provide insights and highlight similarities and differences. This paper is expected to be useful for researchers, processor architects, and chip-designers.

KEYWORDS

classification, parallelization, review, speculation, value locality, value prediction

1 | INTRODUCTION

It has been traditionally accepted that true-data dependent instructions cannot be executed in parallel, and the dataflow-graph of a sequential application decides the limits of the maximum instruction-level parallelism (ILP)* that can be exploited. However, the phenomenon of "value locality" and the ability to exploit it using "value prediction"[1,2] allows exceeding the limits of true-data dependencies, while still maintaining application correctness. VP for an instruction may allow executing its dependent instruction much before the completion of its execution. For example, as shown in Figure 1, without using VP, the second instruction can be dispatched only after the first instruction has been executed since its result becomes available only at this point. However, on using VP, the second instruction can be dispatched based on the predicted outcome of the first instruction. Later on, the predicted value can be compared with the actual value and in case of correct prediction, the execution can continue.

Further, by facilitating parallelization of serial portions, VP can allow exceeding the speedup-limits shown by Amdahl's law.[3] VP allows addressing both latency and bandwidth issues, for instance, by bypassing the memory hierarchy for highly predictable loads, bandwidth is saved and by predicting the loads based on the instruction addresses, load latency can be reduced. Similarly, VP allows reducing data bus traffic[4] and communication delay,[3] etc. Furthermore, VP is a must for speculative multithreaded processors, since without VP, their performance will be no better than that of traditional superscalar processors.[5]

Leveraging the full potential of VP, however, requires addressing several challenges. The predictor design involves an inherent trade-off between the complexity and accuracy. Complex/large predictors may have high area, latency, and energy penalty which reduces their performance benefits, for example, Sam et al[6] note that due to increase in latency with size/complexity, an 80 KB finite context method (FCM)[7] predictor provides lower performance than the same predictor with 10 KB size. Further, even with accurate prediction, the improvement in performance from VP may be small. Clearly, a careful choice of value predictor parameters and management strategies is definitely required. Recently, several techniques have been proposed to address this need.

In this paper, we present a survey of techniques for exploiting value locality and performing value prediction. Figure 2 shows the outline of this paper. In Section 2, we provide a background on VP and discuss some common value predictors. In Section 3, we mention the challenges associated with VP and provide a classification of research works on VP. We discuss several VP techniques in Section 4 and the techniques for performing

*The following acronyms are used frequently in this paper: compensation code (CCode), confidence estimation (CE), finite context method (FCM), instruction level parallelism (ILP), instruction set architecture (ISA), memory level parallelism (MLP), out-of-order (OOO), program counter (PC), register file (RF), reorder buffer (ROB), thread-level parallelism (TLP), value prediction (VP).

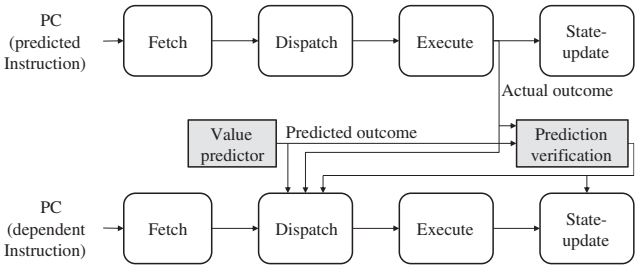

FIGURE 1 An example of how prediction of an instruction's outcome (i.e., value prediction) allows executing its dependent instructions in the same cycle[8]

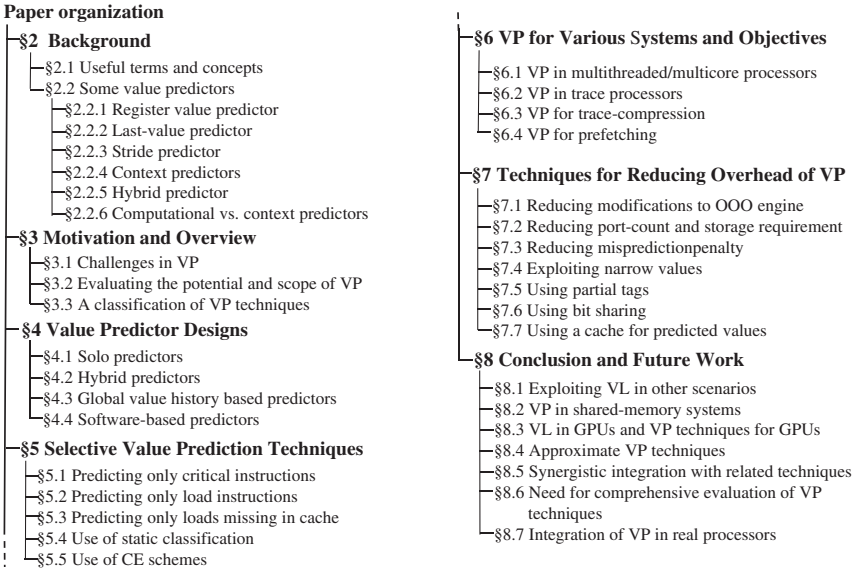

FIGURE 2 Organization of the paper

selective VP in Section 5. In Section 6, we discuss VP as used in various systems and for achieving different optimization objectives. We then discuss the techniques for reducing the overheads of VP in Section 7. Although many of the referenced papers span multiple categories, we organize them based on their primary contribution. We focus on key insights of proposed works and not quantitative results since different works have been evaluated using different evaluation platforms. We discuss the conclusion and future challenges in Section 8.

2 | BACKGROUND

2.1 | Useful terms and concepts

We now discuss some terms/concepts that will be useful throughout the paper.

Static and dynamic instruction: Static instruction count shows the number of instructions in a program, whereas the dynamic instruction count shows the actual number of instructions executed in a given program execution. Static and dynamic instruction counts can differ, for example, due to branch operations, some instructions may not be executed whereas due to loops, some instructions may be executed multiple times.

Value locality (VL) and value prediction (VP): Value locality refers to the likelihood of the recurrence of previously-encountered values within a storage location.[1,9] To understand the origin of VL and the scope for VP, consider Figure 3. Here, I1, I2 and I3 are static instructions which are executed inside a loop. The output produced by different *dynamic* instances of these *static* instructions shows some correlation/pattern which is referred to as value locality. For example, in each loop-iteration, the outcome of I1 is a constant (4). Value prediction relies on inferring the value pattern by observing previous outcomes of dynamic instruction instances and based on these observations, the outcomes of future dynamic instances can be predicted. For example, the outcome of I1 in next iteration is same as that in previous iteration and for I2, it differs with that in previous iteration by one. Section 8 discusses other sources of VL.

Recovery schemes on mispredictions: Since VP is a speculative technique, on a misprediction, a recovery scheme is required. One option is to flush and refetch *all* the instructions after a mispredicted instruction. As another option, only instructions directly/indirectly dependent on the mispredicted instruction may be re-executed.[12]

```
for (val = 0; val < MAX; val++) {
  for (j = 0; j < M ; j++) {
     if ( val == 0)
/*I1*/         R1 = 4;     /* Last-value */
     else
/*I2*/         R1 = j+3; /* Single-stride */

/*I3*/   R2 = R1 + 5;    /* Mixed pattern */
  }
}
```

FIGURE 3 An example code which leads to last-value, single-stride, and mixed-pattern behaviors.[10] I1, I2, and I3 are three static instructions

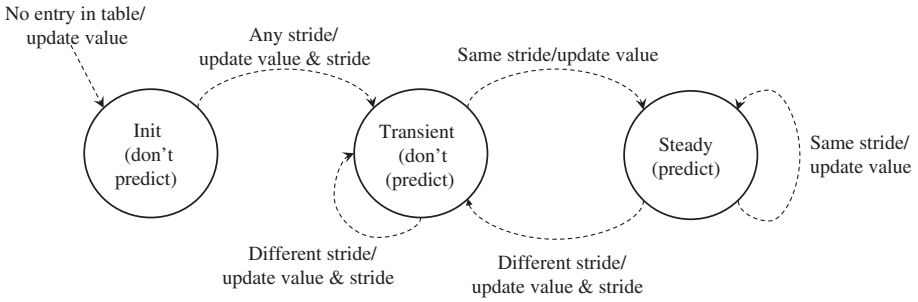

FIGURE 4 An example of confidence estimation scheme for stride predictor[11]

Confidence estimation scheme: Since a misprediction has high overhead, value prediction should be done only in case of high accuracy. CE schemes help in throttling VP in case of poor accuracy. Figure 4 shows the example of a simple 3-state CE scheme for the stride predictor.[11] On the first instance of an instruction, the state is set to "init." In "init" state, when the result of another instance of the same instruction is available, the state is set to "transient" and the stride is recorded. In "transient" state, if another instance of the same instruction produces the result and the new stride is same as the previous stride, the state is set to "steady," otherwise, the new stride replaces the previous stride and the state remains "transient." In the "steady" state, prediction is performed by adding the last seen value with the stride. If a different stride is seen, the state changes to "transient." Clearly, the confidence in a stride is said to be high when the same stride is observed twice. By increasing the number of states traversed before reaching the "steady" state, the bar of confidence required for making a prediction can be increased further.

Predictor learning time and degree: For a predictor, the "learning time" shows the number of values to be observed before the first correct prediction and "learning degree" shows the percentage of correct predictions after the first correct prediction.[7]

Computation reuse: Computation reuse, also called instruction reuse,[13] works by storing the inputs and results of past executions of an instruction. Later on, on encountering the same instruction with same inputs, the stored results are directly utilized without executing the instruction. While VP is a speculative approach, computation reuse is a non-speculative approach.

2.2 | Some value predictors

We briefly review some value predictors to provide insights into VP. Several other predictors are discussed in the remainder of this paper.

2.2.1 | Register value predictor

Register value predictor[14] estimates that the value produced by a load instruction is already present in the destination register, and hence, it avoids writing the register. Such an instruction is termed as a redundant instruction. Except for its confidence estimation (CE) scheme, it does not require any storage, since it uses the register file itself as the source of predicted values.

2.2.2 | Last-value predictor

The last-value (also called last-outcome) predictor[1] predicts the value of an instruction to be the same as the last seen value. As shown in Figure 5A, this predictor stores the last outcome of an instruction in a table and uses this to predict the next outcome. This predictor is useful only for the constant sequences. This predictor can be seen as a special case of last-*K* value predictor.[15]

2.2.3 | Stride predictor

A stride predictor records both the last value fetched by an instruction and the delta between the value and the previous value, called stride. As shown in Figure 5B, this predictor records both the stride and the last seen outcome for an instruction. The next predicted value for the instruction

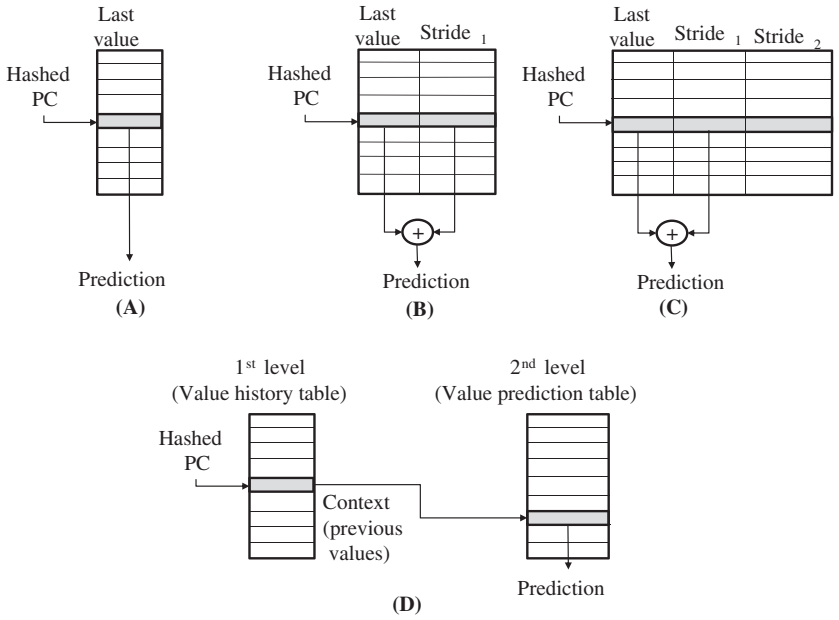

FIGURE 5 A, Last-value; B, stride; C, two-stride; and D, two-level predictors (assuming a hit in predictor table. CE scheme is not shown for simplicity.)

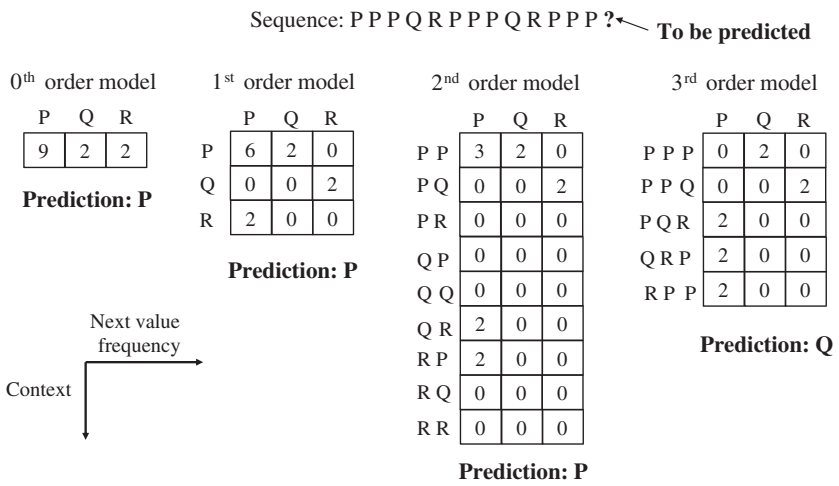

FIGURE 6 Illustration of finite-context models of different orders[7]

is the sum of the last value and the stride. Stride patterns appear due to immediate add/subtract operations and computing addresses of memory references that step with a fixed stride on memory arrays. A variant of stride predictor is the two-delta stride predictor[16] which stores last two stride values, as shown in Figure 5C. It confirms a stride only when it has been observed twice, and thus, it trades-off coverage for achieving higher accuracy.

2.2.4 | Context predictors

Context-based predictors estimate the next value based on the recurring pattern of last several values seen. Context-predictors can also track finite number of reference patterns which cannot be predicted by a stride predictor. These predictors may use two levels of tables.[17] As shown in Figure 5D, the level-1 table records the context or the recent history of an instruction. For every context, the level-2 table records the value which has highest likelihood of following it. For making a prediction, the PC (program counter) is used to find the context (recent values) for an instruction in the level-1 table (Figure 2A). Using this as the index in level-2 table, the predicted value is obtained.

To provide more insights into the meaning of "context," Figure 6 shows finite-context models of different orders. Examples of context-based predictors are FCM[7,18] and two-level[11] predictors. Section 4.1 provides more details on these predictors.

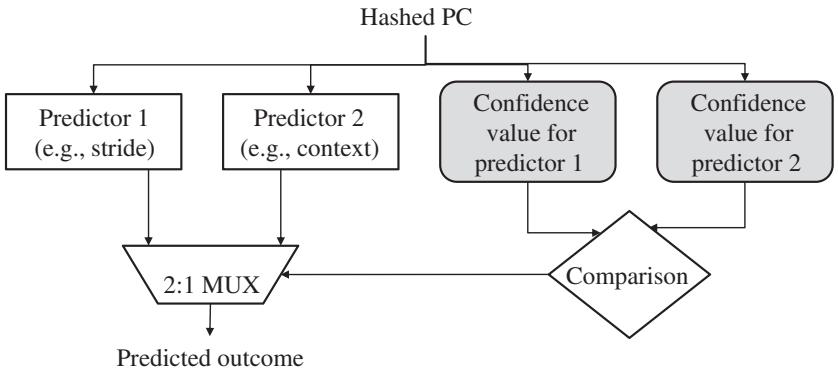

FIGURE 7 A hybrid predictor with CE scheme

2.2.5 | Hybrid predictor

Since different solo predictors are good at predicting different instructions, a hybrid predictor uses multiple component predictors along with a CE or classification scheme to choose the best among them. As an example, Rychlik et al[19,20] combine last-value, stride and FCM predictors. A generic hybrid predictor is illustrated in Figure 7. Here, two solo predictors are used along with CE schemes for them. For any instruction, both the predictors predict the values, and the final prediction is taken from the predictor having higher confidence value. A tie can be broken randomly or using a fixed priority order.

2.2.6 | Computational vs. context predictors

Sazeides et al[7] classify the predictors as computational and context-based. Former predictors speculate the next value based on some operation on the previous values, for example, stride and last value predictors. Context predictors match recent history of values with previous values and speculate based on the previously observed patterns. Context predictors have higher learning time, and hence, they may be unnecessary for constant sequences. Computational predictors have smaller learning time, but they show lower accuracy with repeating patterns. Only context predictors are effective for repeating non-stride sequences. Examples of context predictors are FCM, DFCM (differential FCM), etc. By virtue of exploiting different value patterns, computational and context predictors are complementary, and hence, several hybrid predictors have been designed by combining them, as shown in the remainder of this paper.

3 | MOTIVATION AND OVERVIEW

To motivate the paper, we now discuss the factors, trade-offs, and challenges in VP techniques (Section 3.1). We also discuss the works that evaluate the potential of VP (Section 3.2). We then provide an overview and classification of VP techniques (Section 3.3).

3.1 | Challenges in VP

Efficient implementation of VP presents several challenges, as we show below.

Low accuracy: Compared to a branch predictor, which predicts only one bit of information, VP predicts 32 (or 64) bits of information. Clearly, predicting larger number of data bits is one reason for lower accuracy of VP. Further, several instructions are inherently difficult to predict.[7,21,22] Also, while simple predictors have low accuracy and coverage, increasing the complexity of predictors provides only limited returns.[23]

High recovery penalty: The value prediction needs to be performed on the critical path and while the average performance improvement brought by a correct prediction is small, the misprediction penalty is high.[24] Hence, to improve performance using VP requires high accuracy and/or aggressive recovery scheme.

Hardware overhead and complexity: Several prediction schemes (e.g., Sazeides and Smith[7]) require multi-bank/ported prediction tables which have large overhead. In fact, Bhargava et al[25] note that VP tables need nearly 8 KB to 200 KB storage. Techniques such as use of partial tag[26,27] and bit sharing[28] reduce storage overhead at the cost of reduced accuracy.

A different approach to reduce storage overhead is to implement the predictors in software; however, this brings its own challenges.[29] For example, software-based techniques need to add new instructions in the instruction set architecture (ISA) to enable predictions, update VP tables, and handle mispredictions. These additional instructions exacerbate register pressure and waste memory bandwidth. Further, these overheads increase with increasing complexity of the predictor due to which only simple predictors can be used in the software.

Latency penalty: Bhargava et al[25] note that since most load and integer operation results are consumed within a cycle, VP needs to be very fast. However, even moderately-sized predictors have an access latency of multiple cycles which increases with rising port-count,[30] for instance, predictor latency can be as high as 24 cycles.[25,31] Due to this, VP latency may exceed the execution latency, nullifying the advantage of VP.[32]

Furthermore, VP table needs to be updated with actual result of a predicted instruction. However, if this instruction has large execution latency, VP table is not updated in timely manner. Due to this, subsequent dynamic instructions may use the old value, which affects predictor accuracy.[31,33] To avoid this, some works propose speculatively updating the VP tables[12,34,35] which, however, may reduce accuracy.

Similarly, two-level predictors (e.g., Sazeides and Smith[7] and Goeman et al[36]) require two-step lookups, where the second lookup can start only on completion of the first lookup. This makes prediction latency too high for quickly speculating multiple instances of a static instruction such as the instructions appearing inside the tight loops. In fact, a performance-oriented VP design is expected to add significant complexity and energy overhead to every pipeline stage,[37] especially in the out-of-order (OOO) engine.

Energy penalty: VP affects energy consumption in several ways,[6] for example, speculative execution and recovery mechanisms lead to wastage of energy. Further, predictor access energy increases with rising associativity and port-count.[25] Due to these limitations, VP may not be useful for power-constrained systems.

Limitations due to finite processor resources: Benefits of VP are limited due to finite resources, such as low instruction fetch width and finite instruction window.[31] Low fetch-width limits the ability of fetching data dependent instructions in same cycle. This serializes dependent instructions which provides extra time for computing inputs of dependent instructions. Increasing instruction window size incurs exponential hardware complexity, and after a threshold value, performance improvement of VP does not increase with increasing window size. One approach to address this is using VP in multi-threaded systems.[38]

Similarly, since the branch predictors used in processors are not perfect, they can lead to further branch and value misprediction, increasing the penalty of value misprediction[31] even further. To remove this effect, some works make unrealistic assumptions, such as availability of a perfect branch predictor.[20,32]

The techniques presented in Sections 4 through 7 seek to address these challenges.

3.2 | Evaluating potential and scope of VP

Gabbay et al[32] show that the potential of VP is limited by instruction-fetch bandwidth and instruction issue rate. The fetch bandwidth is constrained by multiple factors such as I-cache hit rate, branch prediction accuracy and misprediction overhead, number of branches predicted per cycle and number of taken branches. Gabbay et al observe that VP does not provide improvement for small instruction fetch rate (e.g., 4) and provides increasing improvement with rising fetch rate, for example, 80% for the fetch rate of 40 instructions. When the producer and consumer of a value are very far distant, in terms of number of instructions, compared to fetch-rate, then both producer and consumer are unlikely to be fetched or executed in the same cycle. In this case, when the consumer is issued, its input values are likely to be available, obviating the need of VP. Thus, when data dependencies span across instructions which are fetched consecutively, the dependent instructions are inherently executed sequentially due to sequential fetching and not due to dependencies. In the second case where the distance between producer and consumer is short compared to the fetch rate, they are likely to be fetched in the same cycle. Also, when the consumer is issued, its input values are unlikely to be ready, and hence, VP is useful for this case. They also propose techniques to improve efficacy of VP in processors with trace-cache.

Gonzalez et al[23] evaluate the potential of VP assuming infinite resources such as memory and register file (RF) ports, fetch bandwidth, RF and cache storage, etc. They evaluate the impact of some architectural parameters on efficacy of VP using a stride predictor. They note that with increasing instruction window size, speedup due to VP increases. VP reduces the penalty of mispredicted branches and hence shows higher improvement with realistic branch predictor than with the ideal branch predictor. Also, speedup with stride predictor is limited since instructions with stride characteristics may not be on the critical path. Further, while even small increase in accuracy of branch predictors is reflected in processor performance, a small increase in accuracy of VP does not lead to similar performance improvement.

Sam et al[6] evaluate the trade-off between predictor size, latency, energy consumption, and performance. They show that on not accounting for predictor latency, complex predictors outperform simple predictors. However, with realistic latency values, simple predictors provide a better trade-off between accuracy and performance than complex ones. Specifically, on increasing the predictor size beyond 20 KB, the energy consumption of predictor increases much faster than the processor performance. For example, above 20 KB, even simplest last-value predictor outperforms the hybrid predictor. Overall, they conclude that too complex or too large/small predictors are inefficient.

Sato et al[39] evaluate the impact of compiler optimization levels on VP efficiency. They predict only register-writing instructions and define "predictability" as the fraction of the register-writing instructions that are predicted by a predictor. They use two predictors: the last-value predictor[40] and the stride+two-level hybrid predictor.[11] They evaluate the applications with `gcc` compiler at `O0`, `O1`, and `O2` optimization levels using `gcc 2.7.2`, MIPS ISA, and SPEC95 benchmarks. They note that for both predictors, for nearly all the tested applications, the predictability does not change significantly with the optimization level. Thus, compiler optimizations do not obviate the need of VP, that is, VP still remains effective for highly optimized binaries.

Endo et al[41] evaluate the impact of compiler optimization levels (`O0` to `O3`) on VP efficacy. They experiment with `gcc 4.9.3` compiler, AARCH64 ISA, and SPEC06 benchmarks. They use their VP technique[37] which is summarized in Section 7.1. They note that VP provides higher performance at lower optimization levels. For the unoptimized code (`O0`), the speedup from their VP technique is especially high due to unscheduled and redundant loads. At `O0` level, predicted load values are more likely to be used before the actual results have been computed. This explains larger performance with `O0` than with other optimization levels.

TABLE 1 A classification of research works

Category	References
	Predictor types
Register value predictor	14,42
Last value predictor	1,3,7,10,11,22,24,26-28,32,34,39,40,43-49
Last *K*-value predictor	11,15,47,50
Single/two-delta stride predictor	2,7,8,10,11,16,20,22,23,25,26,28,29,32,34,39,45-53
FCM	3,7,18,44,47,49,50,53-55
DFCM	36,47,49,55
Two-level predictor	8,10,11,22,28,34,38,39
Hybrid predictors	5-8,10-12,20-22,25,26,28,32,34,36,39,42,47,53,56-60
Comparison of predictors	5-7,11,25,35,36,42
	Objective
Performance	Nearly all
Energy saving	6,25,44
Compressing program traces	50,55
Prefetching/pre-execution	51,52
	Other features
Experiments on real-system	50,55,61
Use of compiler	8,10,14,29,39,41,47-49,57,59,61-65
Comparing VP with	Prefetching,[2] increasing cache size,[27,47] producer-identification,[54] runahead execution[43]
Integrating VP with	Computation reuse,[59] last-width prediction[44]
VP in shared-memory systems	66-68
VP in GPUs	53

We believe that the difference in the conclusions of Sato et al[39] and Endo et al[41] is due to the differences in their value predictor designs, microarchitecture/ISA, evaluation platform, and workloads.

3.3 | A classification of VP techniques

Table 1 classifies the research works on several key parameters, such as the predictor type, objective, strategies to reduce hardware overhead, etc.

4 | VALUE PREDICTOR DESIGNS

In this section, we first discuss several individual predictors (Section 4.1) and hybrid predictors (Section 4.2). We then discuss value predictors that work based on global value history (Section 4.3) and that use software-support (Section 4.4).

4.1 | Solo predictors

Tullsen et al[14] present the register-value (REG) predictor (refer Section 2.2.1) that uses both static and dynamic approaches to decide the instructions to be predicted. Static approach predicts only for loads and dynamic approach can predict for either only the loads or all the instructions. Using compiler management of RF values, effectiveness of the REG-predictor can be further enhanced, e.g., compiler can transform various ways of register reuse into same-register reuse. When there is no intervening write to a destination register, an instruction showing last-value predictability also shows register-value predictability. The advantage of their approach is that it requires no hardware for storing values, no changes to ISA and only minimal changes to datapaths. Also, software has full control over which variables/values are in which register. Further, since the register map is kept updated by register renaming and branch recovery, REG-predictor does not suffer from stale values. Furthermore, by correlating the instructions/variables which repeatedly store the same value as each other, REG-predictor can use the outcome of one instruction as the prediction for another.

Lipasti et al[1] present a last-value predictor (refer Section 2.2.2). They classify static loads into unpredictable, predictable, and constant (highly predictable) loads based on the results of their previous predictions. For predictable loads, predicted value is compared with the actual value obtained from memory hierarchy. For constant loads, a constant verification unit is used to avoid accessing memory hierarchy and still keep the prediction table entries in sync with main memory. They assume a small misprediction penalty (1 cycle), and thus, restarting execution on incorrect predictions does not affect performance.[1] They show that their technique improves performance and allows collapsing true dependencies by completely avoiding memory latency.

Lipasti et al[40] extend last-value predictor scheme[1] to all instructions that write to an integer or floating-point register. They show that a large fraction of these writes can be easily predicted before they are issued and this allows overcoming the barriers of serialization constraints imposed by operand dataflow. This allows speculative and parallel execution of dependent instructions, which facilitates better utilization of hardware and collapsing of critical paths in dependence graphs. They use a value predictor and a scheme for verifying the correctness of predictions. They show that their technique provides significant performance improvement by virtue of exceeding dataflow limit.

Sazeides et al[7] evaluate computational and context-based predictors. They use unbounded predictor tables that are kept updated with correct data values and observe that data values are highly predictable. Also, a context-based predictor is generally more accurate than a stride predictor which in turn, is more accurate than a last-value predictor. Also, predictability of some (e.g., `add`) instructions is much higher than that of others (e.g., `load` and `shift`). The variation in predictability for different instructions is higher for computational predictor and smaller for context predictor. Since context predictor also incurs higher cost, they propose using a hybrid stride-context predictor. Few static instructions are responsible for better predictability of context predictor over stride predictor. Further, only few values are required for predicting a large fraction of dynamic instructions in context-predictor, and hence, the requirement of table size is small.

Goeman et al[36] note that FCM[7] is ineffective in prediction of stride patterns due to interference between stride and non-stride patterns occurring in the second-level prediction table. To avoid this interference, the number of entries taken by the stride pattern needs to be reduced. They propose using differences between values (i.e., strides) instead of values themselves, since with this optimization, stride patterns are mapped to only one entry in the second level table and irregular repeating patterns also remain predictable.

As an example, Figure 8 shows the level-2 table in both FCM and DFCM for a repeating sequence $(1, 2, 3, 4, 5, 6, 7)_n$. The table shows different contexts and the predicted value for them. In FCM table, each stride pattern takes one entry, whereas in DFCM, all stride patterns with the same stride use just one entry. The patterns after a counter reset (e.g., `7, 1, 2`) need one entry each; however, their frequency is much lower, as shown in the third column of the table. Clearly, DFCM can efficiently record and predict stride patterns, even if those patterns have not been repeated yet. DFCM predictor provides higher accuracy than FCM predictor alone and a hybrid of FCM and stride predictor. However, similar to FCM, DFCM also has high prediction latency due to two-level lookup.

Wang et al[11] present stride, two-level, and hybrid value predictors. The first and second levels of the two-level predictor are value history table (VHT) and pattern history table (PHT), respectively. For each instruction, VHT maintains a tag, last-K ($K = 4$) distinct values and a $p \times \log_2 K$-bit pattern indicating outcome of last p instances of the instruction. Since each instruction can have K outcomes, $\log_2 K$ bits are required to record each outcome. Each PHT entry uses K independent saturating counters to maintain a condensed history of previous outcomes for each possible $2p$-bit pattern. For making prediction for an instruction, its value history pattern is used to locate the corresponding PHT entry. The maximum among the K counter values for this PHT entry is selected, and if this is greater than a threshold, then the outcome corresponding to that counter is taken as the predicted value. No prediction is made if the counter is lower than the threshold. They show that stride and two-level predictors provide better accuracy than the last-value predictor. To improve the coverage, they propose two hybrid predictors: (1) last-value and stride and (2) stride and two-level. For example, in hybrid predictor (2), if the counter is lower than the threshold, the stride predictor is used; otherwise, the two-level predictor is used. They show that the hybrid predictors provide higher accuracy than the constituent predictors.

Burtscher et al[15] design a last K-value predictor, e.g., for $K = 4$, the predictor keeps 4 recent loaded values, as shown in Figure 9. The predictor has K components each consisting of same number of slots for storing a 64-bit value. Each component uses counters for CE. Based on the PC of the instruction, all K components predict the value stored in their slot, and the value of the component with highest confidence is taken if it is also larger than a threshold (λ). Thus, prediction is made only if the confidence in it is higher than a threshold. This is shown in Figure 9. The CE-counters of each component are updated based on comparison-result of loaded value with the stored value. The predictor proposed by Wang et al[11] keeps the last K distinct values, whereas the values in their predictor may not be distinct. They observe that their predictor provides higher performance than that in Wang and Franklin[11] and the last-value predictor ($K = 1$). For a fixed storage budget of predictor (i.e., the product of K and the number of slots is

Sequence: $(1, 2, 3, 4, 5, 6, 7)_n$

Context	Value	Access/Iterations
1 2 3	4	1
2 3 4	5	1
3 4 5	6	1
4 5 6	7	1
5 6 7	1	1
6 7 1	2	1
7 1 2	3	1

(A)

Context	Value	Access/Iterations
1 1 1	1	4
1 1-6	1	1
1-6 1	1	1
-6 1 1	1	1

(B)

FIGURE 8 The stride pattern stored in level-2 table of A, FCM and B, DFCM[36] predictors for the same sequence. Clearly, DFCM table incurs small storage overhead than the FCM table

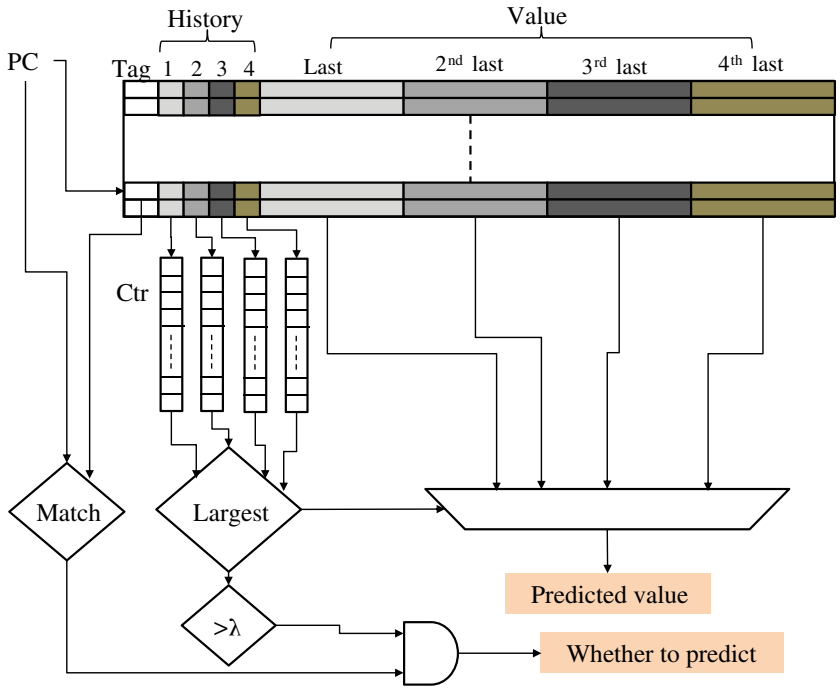

FIGURE 9 Last-*K* value predictor (*K* = 4, Ctr = counter). Note that history 1 corresponds to last-value, history 2 corresponds to 2nd last value, and so on

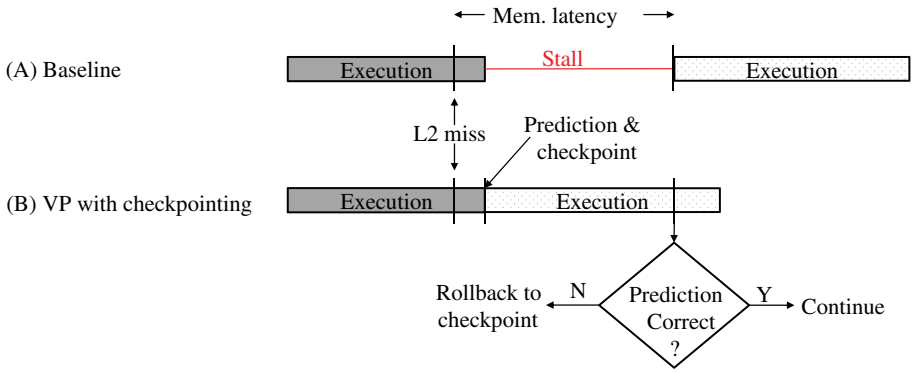

FIGURE 10 Checkpoint-assisted VP approach[43]

fixed), using *K* = 4 provides better performance than using *K* = 1, 2 or 8 since once the information about the frequently executed loads is captured by the predictor, additional slots store information only about the rarely executed (i.e., unimportant) loads.

Ceze et al[43] note that a long-latency load (e.g., L2 miss) clogs the re-order buffer (ROB) since it waits at the head of ROB till the data is returned from memory. This is shown in Figure 10A. They propose using VP and checkpointing to address this limitation. After an L2 cache miss, when the load reaches the head of ROB, its state is checkpointed, the load is retired and the execution continues with the predicted value, as shown in Figure 10B. When the value is returned from memory, it is compared against the predicted value. On a correct prediction, execution continues and the checkpoint is discarded, whereas on an incorrect prediction, execution restarts from the checkpoint. The predictor is placed by the L2 cache controller. The speculative state is stored in L1 cache and cannot be replaced from there, until the speculative region is squashed. VP is stopped when (1) the confidence is predictions is low, (2) outstanding predictions exceed the limit of hardware buffer, (3) speculative state exceeds a fraction of L1 cache size, or (4) the number of speculatively committed instructions since the most recent checkpoint reaches a threshold. They show that their technique provides higher performance improvement than runahead execution.

4.2 | Hybrid predictors

Hybrid predictors provide higher accuracy than solo predictors by using the most suitable predictor for each value pattern. However, due to this, they also incur large hardware overhead. Conversely, for the same total hardware budget, hybrid predictors can use smaller component predictors

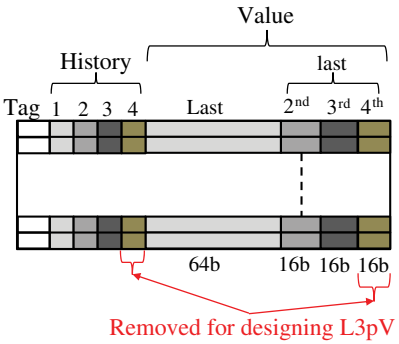

FIGURE 11 VP table in last-4 partial value predictor (L4pV)[60] (remaining part remains the same as shown in Figure 9). The parts pointed by red arrow are removed for obtaining last-3 partial value predictor (L3pV)[60]

which have lower accuracy than their solo predictor versions. Further, to optimize space-usage efficiency, the component predictors of a hybrid predictor should be effective in predicting different data patterns.

Burtscher et al[60] propose a technique to reduce the storage overhead of hybrid predictors. They begin with the last-four value predictor[15] and note that most significant bits of four values are mostly identical. Hence, they store full bits for only one value and 25% bits for the remaining three values which reduces the storage to nearly half. Their "last-4 partial value predictor" (L4pV) is shown in Figure 11. Further, for $K \geq 2$, a last-K value predictor already provides information required for a stride predictor, since stride can be computed as the difference between successive values. This allows making their stride predictor storage-less.

Further, they note that the fourth component of L4pV makes little contribution to the performance. Hence, they omit it, leading to a L3pV predictor as indicated in Figure 11. In place of this, they add two extra CE schemes. One of these is used for stride predictor. For second CE, they use a register-value predictor[14] which is also storage-less. They also use a strategy to prevent infrequently executed loads that alias with frequently executed loads from evicting the latter. Overall, their REG+Stride+L3pV hybrid predictor provides better performance than predictors of much larger size.

Burtscher et al[42] evaluate all hybrid predictors designed from the combination of five predictors: register value (REG), last value (L1V), 2-delta stride (ST2D), last four value (L4V) and FCM. In the hybrid predictor, the predictor with the highest confidence is selected and used for prediction if its confidence is higher than a threshold. They note that, when used alone, REG performs poorly; however, it is a component in most high-performing hybrid predictors since it predicts loads that other predictors cannot. Conversely, effective individual predictors (e.g., FCM) do not always lead to effective hybrids. In fact, for some combinations, a negative interference is seen, e.g., REG+ST2D performs better than REG+ST2D+L1V; and no benefit is obtained on adding ST2D component to REG+L4V+FCM, REG+L1V+FCM and REG+L1V predictors. This is because including a predictor in the hybrid predictor complicates the selection process and the selection-related losses may nullify predictability advantage of the added predictor. They also find that REG+L1V and REG+ST2D predictors, although being small, are most efficacious. Overall, hybrid predictors can provide much larger performance than individual predictors, which underscores the fact that hybrid predictors are important for exploiting different types of locality patterns existing in applications.

4.3 | Global value history based predictors

While most works use local context (i.e. context formed by the same instruction), some works use global context or value history (e.g., context formed by the instruction on different paths or by multiple instructions) [35,46,49,69] to make prediction for an instruction. We now review them.

Zhou et al[35] note that the value sequence generated by previous executions of the same instruction creates a "local value history" (LVH), whereas the value sequence generated by all dynamic instructions creates a "global value history" (GVH). A long GVH encompasses LVH also. They show that GVHs show very strong stride-pattern locality and several instructions which are difficult-to-predict using LVH can be easily predicted using GVH-based predictors. They propose a predictor for exploiting stride-based pattern in GVH which is shown in Figure 12. Let the values produced by the dynamic instruction j be v_j. They record the values of completed instructions in a table in order of their execution. A prediction table records the selected distance (p for v_M-v_{M-p}) used for prediction and deltas between instruction's outcome and outcome of m instructions which completed just before it (i.e., x_M- x_{M-j}, for $j = 1$ to m).

To make a prediction, the table is indexed using the PC of the instruction (❶). Then, the value recorded at entry p (corresponding to the "distance" variable of table entry) of global value queue (GVQ) is retrieved (❷ on the right) along with the stride value (diff_p) for that "distance" value (❷ on the left). The sum of these two quantities (❸) provides the prediction (❹).

The table is updated as follows. On completion of an instruction, the deltas between its outcome (① on the right) and the values recorded in GVQ (① on the left) are computed. An m order predictor has m delta values. These deltas (② on the right) are compared with those stored in the corresponding entry of the prediction table (② on the left). On a match (③), the matching distance is recorded in the "distance" variable; otherwise,

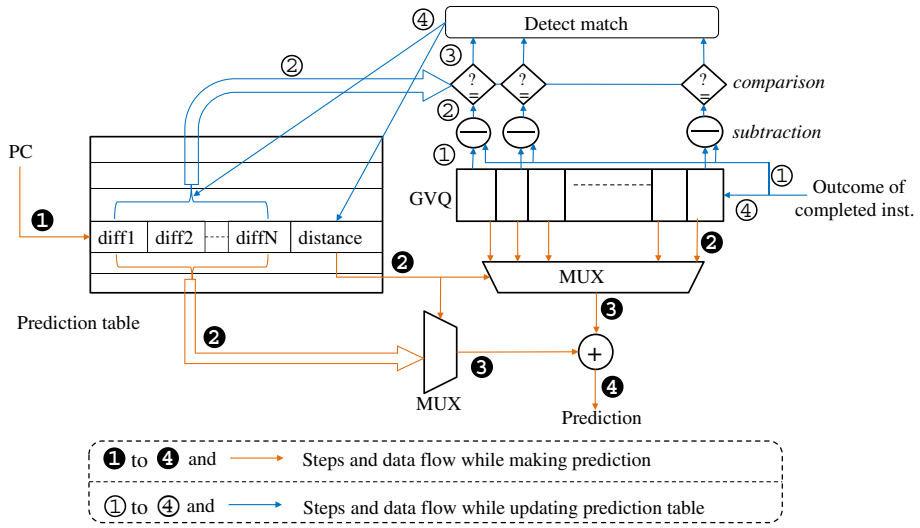

FIGURE 12 Prediction and updating of table in GVH-based stride predictor[35] (inst. = instruction)

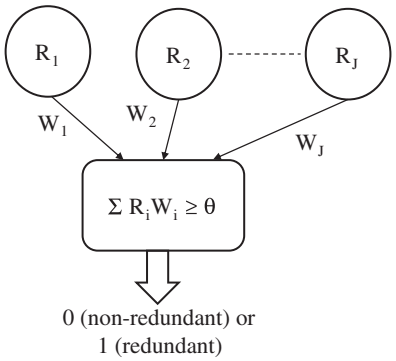

FIGURE 13 Perceptron design in the work of Seng et al[70]

computed deltas are recorded in the prediction table without updating the "distance" variable (④ on the left). Also, the present outcome is shifted in GVQ (④ on the right).

A challenge in using their scheme is that due to the value delay, the correlated values may not be present at the time of making the prediction, and this affects a GVH-predictor more than an LVH-predictor. To address this, they propose using the speculative values. They evaluate their scheme with both all value producing instructions and only the load instructions. They show that their technique provides higher accuracy and coverage compared to stride and context predictors based on LVH.

Nakra et al[46] note that most VP techniques speculate based on the values produced by the *same* instruction, and thus, they utilize local context only. They propose a technique which utilizes global context by predicting values of an instruction based on the behavior of *other* instructions. They propose two schemes. First, different values for an instruction are speculated on different paths which are separated using branch history. This scheme uses the recent branch results with instruction address to predict the next value of the instruction. Second, values are predicted based on the correlation between values produced by nearby instructions in the dynamic instruction stream. They also propose strategies to integrate path information for improving effectiveness of stride predictor. They show that their technique improves prediction accuracy compared to the local-context based VP techniques.

Thomas et al[69] note that traditional predictors use context formed from multiple recent instances of a static instruction; however, this does not always provide required information for accurate prediction. They propose a technique which improves predictability of difficult-to-predict instructions by leveraging predictability of their closest easily-predictable producer instructions in the dataflow graph. Their technique uses traditional context for easily-predictable instructions and better context for difficult-to-predict instructions. Their technique leads to significant improvement in predictor accuracy.

Seng et al[70] propose a perceptron-based predictor for predicting redundant instructions. As shown in Figure 13, J recently committed instructions form input to a perceptron and the input value is 1 or −1, depending on whether an instruction is redundant or not. If the weighted sum of inputs is greater than a threshold, the instruction is predicted to be redundant. Based on the correctness of predictions, weights are individually adjusted. They use a table of perceptrons and the perceptron for an instruction is selected by indexing with the lower bits of instruction address. All perceptrons operate independently. Their predictor design lowers storage budget compared to the design of Tullsen et al.[14] Also, while the REG-predictor[14]

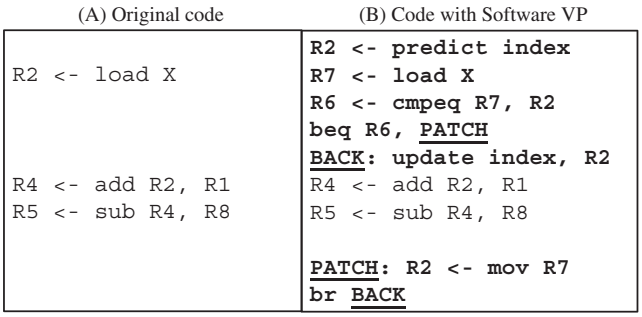

FIGURE 14 A, Original code where two instructions are dependent on a load instruction; B, Software VP with CE approach.[29] The branch predictor tries to estimate whether the actual outcome (R7) is same as the predicted one (R2)

works based on local history, their predictor uses global history (i.e., the outcome of previous *J* instructions). They note that REG-predictor and their predictor perform better for different applications, although the average performance of their predictor is better.

4.4 | Software-based predictors

Larson et al[29] present a compiler-directed VP technique to reduce the hardware overhead. They propose using branch predictor to estimate confidence of VP, since the branch predictor already includes mechanism for CE and misprediction recovery. Their technique encloses the VP location with a branch instruction which measures the confidence in the prediction. If the branch predictor predicts branch to be not taken, it indicates that the confidence in accuracy of VP is high and speculative progress can be made. If branch predictor predicts branch to be taken, confidence in VP is low and it proceeds to a CCode which waits for execution of the load instruction. If the branch prediction itself is incorrect, the corresponding overhead is incurred.

Figure 14 illustrates their technique. In the original code, two instructions have chained dependence on a load instruction. Their technique replaces the destination register of the load with a free register (`R7`). The prediction is stored in the original destination of the load (`R2`). Further, compare (`cmpeq`) and branch (`beq`) instructions are inserted for validating the prediction. In case of correct prediction, the program proceeds normally with the predicted value, but for an incorrect prediction, the branch is taken to the `PATCH` code, where the correct value is placed in the destination register (`R2 ← mov R7`).

They use a stride predictor and find its stride by profiling. They study a hardware-supported and a compiler-only version of their technique of which the first version provides higher performance. In the first version, predictor information is kept in a hardware table, whereas in the second version, this information is kept in memory and is brought to registers at the time of entering into the function. Candidates for VP are selected based on whether the benefit from VP is higher than its penalty. Also, for long chains of dependent instructions, their technique selects instructions which frequently lie at middle of the chain since this allows splitting the chain in half for enabling parallel execution. They note that for cases where VP was correct but branch predictor indicated incorrect prediction, useful progress made based on VP needs to be discarded. To avoid it, they add a new branch instruction which does not recover from such a misprediction. Their approach of using branch-predictor as a CE-scheme for VP boosts performance with minimal hardware overhead.

Fu et al[8] propose a compiler-based technique for VP and scheduling. In their technique, the compiler identifies long dependence chains and uses VP to break this into smaller chains. Information for VP is obtained using value profiling. They introduce two new instructions for interfacing with the VP table: first, for loading the predicted value into a register and, second, for updating the VP table with the actual outcome. Compiler generates compensation code (CCode) and issues instructions for updating the VP table which reduces their hardware overhead. For different predicted values, compiler assigns different indices into VP table. This reduces table conflicts compared to a hardware-only technique which generates VP table index from some bits of the PC. Their technique provides large performance improvement and can be applied to both dynamically and statically-scheduled processors.

Fu et al[64] extend their value speculation scheduling scheme[8] to a software-only technique which does not require ISA extensions or VP hardware and thus can be applied to present architectures also. Since software VP requires higher amount of code instrumentation than in the previous technique,[8] to restrict the increase in code-size, they use only a simple static stride prediction approach. Thus, although their technique cannot use complex and more accurate predictors such as context predictors, the stud predictor used by them provides higher accuracy than simpler predictors such as register and last-value predictors.

Compared to the previous scheme[8] which loads the predicted value from a hardware table, to obtain the predicted value, their technique adds a constant stride to the register which holds the previous value. The static stride is found through value profiling. Thus, this technique uses the already-present `ADD` instruction to obtain the prediction instead of introducing a new instruction for loading the prediction from a hardware table. Their technique improves processor performance.

Comparison between hardware- and software-based VP techniques: The compiler has broader picture of the application data-flow than what is available to a hardware-only scheme. Hence, the compiler can exploit larger scope of scheduling. Also, unlike in the hardware-only VP techniques, in static

VP techniques (e.g., Fu et al[8,64]), the predictor is not on the critical path. Due to these reasons, in static VP techniques, the dependent instructions can begin as soon as possible before the predicted instruction that they depend on, as shown in Figure 1. Also, compiler can select better candidates for VP, reducing the number of accesses to VP tables. Another benefit of software VP techniques is that the CCode is produced automatically which reduces the need of detailed hardware recovery schemes.

The limitation of static scheduling techniques is that the speculative instructions are committed as speculative, and hence, they invariably need predicted values. By comparison, in hardware-only techniques, the decision about speculatively executing instructions can be dynamically made depending on the confidence in the prediction. This allows avoiding low-confidence predictions which reduces the misprediction overhead. Especially, the software-only VP technique[64] can use only simple predictors and hence, it cannot predict accurately under complex data patterns.

5 | SELECTIVE VALUE PREDICTION TECHNIQUES

As shown in Table 2, several works perform selective VP to reduce the pressure on VP tables and reduce mispredictions. These works may predict only critical instructions (Section 5.1), load instructions (Section 5.2), or load instructions missing in cache (Section 5.3). Similarly, for finding highly predictable instructions, some works use static classification (Section 5.4) or CE schemes (Section 5.5). We now summarize them.

5.1 | Predicting only critical instructions

Calder et al[12] present a technique for selective VP which is useful when the misprediction latency is high and the predictor capacity is limited. They use a hybrid predictor with two-delta stride and context predictors. Their predictor updates values and strides speculatively and an incorrect update is handled in the instruction-commit stage of the pipeline. CE counters are updated during writeback when the accurate outcome is available. They add two counters to the VP table. The first counter adds hysteresis to mitigate capacity thrashing by enabling highly-predictable instructions to stay in table. The second counter allows updates to the confidence counter and updates of the VP table only when an instruction has warmed up its prediction information.

To reduce the pressure on the tables, they evaluate multiple strategies for selecting which instructions can write to the tables: (1) *no-filtering:* all register defining instructions (2) *Sourced:* only instructions which define registers which are actually used by another instruction in the current instruction window (3) *Loads:* only load instructions (4) *Path-M:* only instructions on critical path with longest path-length greater than M cycles (e.g., $M = 0, 10, 20, 50$ cycles). They observe that *Loads* filter provides ~75% of potential of value prediction while requiring prediction of only ~30% instructions. Similarly, Path-0 filter provides same performance as predicting all instructions while significantly reducing the number of predicted values. They also apply filtering to instructions that can use a predicted value which reduces the impact of misprediction. Use of predicted values by instructions on the longest path with low-confidence is more advantageous than that by instructions not on the longest path.

Fields et al[72] model the data and resource dependencies in a program using weighted dependence graph. The longest weighted path in this graph shows the critical path. They use this information to focus processor management policies on critical instructions. Firstly, scarce resources (e.g., port, functional units, prediction unit slots) are preferentially allocated to critical instructions. Further, value speculation is avoided for non-critical instructions. This reduces the mispredictions significantly and also improves performance.

5.2 | Predicting only load instructions

Load instructions account for nearly 20% of all instructions and they incur the largest latency. Out of several result-producing instructions, loads show highest predictability and they are also generally in critical paths.[8] Hence, by predicting only loads, the complexity of predictor and the number of predictions required can both be reduced.

Ibrahim et al[54] note that barrier synchronization leads to large latency penalty in parallel applications. For example, in Figure 15A, the computation $Y \leftarrow X$ on processor P2 can be ideally performed after the X value is communicated from P1 to P2. However, due to the presence of barrier, it is actually performed at a much later time. They propose using VP to speculatively execute instructions after the barrier without violating data dependencies. As shown in Figure 15B, by predicting the value of X, the computation on P2 can be performed much earlier by using the predicted value of X (X'). Only the loads of shared memory references are responsible for synchronization and hence, prediction is made for only these loads which occur before the barrier. The limitation of this approach is that multiple predictions need to be made to avoid synchronization delays since validation can be performed only when all the threads reach the barrier. Hence, with increasing number of threads, the number of predictions required

TABLE 2 Selective VP techniques. Predicting only …

Category	References
Load instructions	1,8,12,14,27,38,41,44,47,64,71
Instructions on critical path	12,29,72
Loads that miss in cache	24,47
Highly-predictable instructions	Found from static classification[8,10,47,64] and from CE scheme (most others)

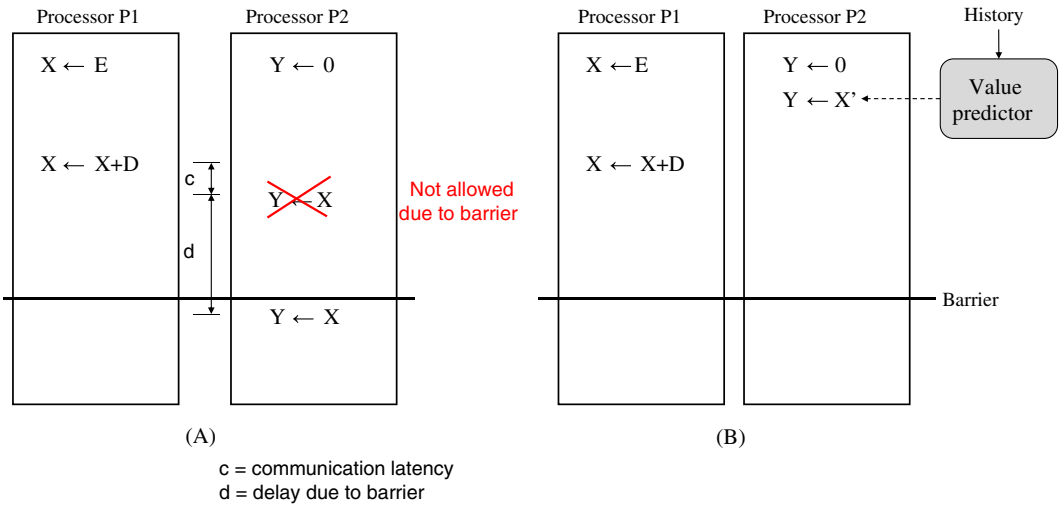

FIGURE 15 A, Dependency arising from barrier and B, breaking this dependency using VP[54]

and the overhead of validation also increases. Also, unlike other approaches (e.g., producer identification[54]), VP does not allow early recovery from misprediction. They observe that VP provides smaller reduction in synchronization overhead than producer identification.

5.3 | Predicting only loads missing in cache

Gellert et al[24] present a technique where the predictor is accessed only for loads that miss in L1 cache. This reduces the latency and energy penalty of accessing predictor which results in improved performance and energy efficiency. Further, it allows reducing data cache capacity while maintaining performance.

5.4 | Use of static classification

Static classification can help in finding highly predictable instructions[8,10,47] and the most suitable predictor component in the hybrid predictor for each instruction based on its predictability pattern.[10,47,57] Unlike dynamic schemes such as use of CE, static classification scheme does not incur warming-up overhead. The limitation of static classification is that the information about each instruction need to be encoded in the binary which increases its size. Also, modifying the binary may not be possible in some scenarios.

Gabbay et al[57] propose using program profiling to boost the efficacy of VP based on the observation that for several workloads, a change in input does not have large impact on prediction accuracy.[47] They profile the application with test inputs and record whether an instruction is predictable using stride or last-value predictor. For a predictable instruction, if the ratio of correct non-zero stride-based predictions to total correct predictions is larger than a threshold, then it is marked as stride predictable, otherwise as last-value predictable. Using compiler, this information is encoded as directives in the instruction opcode. Based on this directive, at runtime, an instruction is predicted using stride or last-value predictor and an instruction without a directive (i.e., unpredictable) is not predicted. Their technique allows reducing misprediction overhead and improving utilization of VP-table by keeping only predictable instructions in it. Their technique outperforms hardware-based classification for most workloads.

Burtscher et al[47] present a compiler approach for identifying loads that are predictable and miss in cache. They divide loads into 20 groups based on parameters such as its type (e.g., array or scalar), memory region accessed (heap/stack/global), and the loaded value (pointer/non-pointer). They study cache behavior and predictability of these groups. They note that load-value predictability or unpredictability of the groups remain same across applications. Also, a few (for example, six) groups contribute to nearly half the loads and a large fraction (90%) of cache misses. Thus, using this compiler information, VP can be enabled for these loads only, which reduces conflicts for VP tables. Also, a hybrid predictor with static selection of predictor can be used to exploit different predictability patterns. They also note that well-known predictors, e.g., FCM[7] and DFCM,[36] are effective for loads that are not crucial for performance. Their results hold for both Java and C applications and for different inputs. Thus, by virtue of relating static application behavior with predictability and cache performance, their approach allows making VP decisions in compiler.

Zhao et al[10] present a scheme for statically identifying and classifying instructions into multiple value predictability patterns. This information is saved with the instructions and is used at runtime to ascertain the most suitable predictor for VP. Their profiler records outcomes of up to 100 consecutive instances of every static instruction. By postprocessing these outcomes, each static instruction is classified into one of the following categories: last-value, single-stride, context, mixed-pattern, and unpredictable. These are shown in Figure 3. They note that classifying instructions in just three categories, viz, last-value, single-stride and "others" provides most of the benefit of the full classification since the first two categories occur most frequently due to loop-invariant and loop-induction computations, respectively. Their technique requires information available in a single function only at a time. Their technique provides better performance than a dynamic scheme.

5.5 | Use of CE schemes

Most research works use a CE scheme for improving VP accuracy. CE scheme can be based on saturating counters[1,2,11,73] or prediction-history/branch-predictor.[27,29]

Burtscher et al[27] note that since both branch predictors and CEs make binary decisions, the ideas from branch prediction field can be used to design CEs. They evaluate two approaches for CEs, viz, use of prediction-outcome history and saturating counters. These approaches work on the insight that successful prediction of an instruction in past indicates its successful prediction in future. The history based approach records most recent success/failure (1/0) outcomes. They use profiles to decide the history patterns when a prediction should be made. Saturating counters are incremented/decremented on each correct/incorrect prediction till they saturate at fixed higher/lower values, respectively. A higher counter value indicates higher confidence. However, counters suffer from saturation effects and do not record sequence information, and hence, they use the history-based approach. They show that four-bit histories are sufficient for high accuracy and with increasing number of history bits, the accuracy and coverage improves further. Also, even the simple last-value predictor, when augmented with their CE approach, provides higher performance than other complex predictors.

6 | VP FOR VARIOUS SYSTEMS AND OBJECTIVES

6.1 | VP in multithreaded/multicore processors

On using VP in a single-thread context, the long memory latency cannot be fully hidden since after VP for a long latency load, subsequent instructions cannot commit before return of the load, and hence, speculative uncommitted instructions soon fill the instruction window. Also, only one prediction can be made for any dynamic instruction. Using VP in multithreaded context allows addressing these limitations,[5,24,38,48,59] since the extra thread can progress on speculative path. We now review such works.

Tuck et al[38] propose using VP in a multithreaded context where at the time of prediction, a new thread is spawned to follow speculative execution. This thread can commit instructions from its instruction-window and thus make quick progress. When the load returns, either the main or speculative thread is discarded depending on whether the prediction was correct or incorrect, respectively. Only memory writes need to be buffered, and upon confirmation of a prediction, only these store buffers are released, whereas in single-threaded VP, all speculative instructions need to be serially committed. In their approach, only the size of speculative store buffer and not instruction queue and reorder buffer, limit the speculation distance. By decoupling speculative and non-speculative streams, their multithreaded VP approach allows greater progress of speculative stream, creating the impact of a larger instruction window. Their approach outperforms single-threaded VP approaches and is also effective with floating-point computations.

Liu et al[3] note that in many-core systems, VP can lower both computation and communication latency. Communication latency is defined as the time in satisfying true data dependence between instructions/threads for exploiting ILP/TLP, respectively. VP eliminates communication latency by allowing both consumer and producer to execute in parallel. Conventionally, the maximum speedup achievable on parallelizing an application is shown by Amdahl's law; however, they show that fine-grained VP enables exceeding this limit by allowing parallelization of the serial portions of the application, making VP highly useful for many-core systems. This advantage provided by VP depends on data redundancy present in the application, which can be determined experimentally or using information theory approach. They show that even simple predictors, e.g., last-value or context predictors, provide large performance improvement for parallel applications. Their approach provides significant performance improvement compared to leveraging only the intrinsic parallelism.

Wu et al[59] note that some instructions are predictable but not reusable, whereas opposite is true for other instructions. They propose an integrated technique for computation reuse[74] and VP. Their technique leverages VP of applications at the level of a region. The compiler finds computation regions amenable to VP or computation reuse. At runtime, the outputs of a region are reused or predicted as a whole. They use speculative multithreading hardware for efficiently using the same hardware for both computation reuse and VP. It has two cores, called main and checker cores, respectively. The hardware has buffers for remembering previous execution instances of specified computation regions. Specifically, it remembers the input registers and their values while entering the region and output registers and their values while exiting the region. On detecting the region again, their technique tries to reuse the computation; however, if the inputs do not match any instance, the stored information is used for VP. The predicted values are used for speculatively continuing the execution on the main core, whereas the computation region is forwarded to a checker core for validating its output against the predicted output. Successful validation leads to committing the results from main core whereas in case of a failure, execution is restarted from the correct exit of the mispredicted region, using the correct output from the checker core. Figure 16B summarizes their implementation for overlapping the execution of the computation region with speculative execution of the subsequent code. For the sake of comparison, Figure 16A shows the implementation of computation reuse on a single-threaded architecture. Their combined approach outperforms use of VP or computation reuse alone.

Li et al[48] propose a VP technique for enhancing speculative computation in parallel-threaded applications. They use a two-core system. One core, termed master, executes in non-speculative mode and the other core executes in speculative mode. Both cores share the memory hierarchy but have separate RF and application state. On reaching a "fork" instruction, the master core spawns a thread which runs on the speculative core beginning with a specified address. Both master and speculative threads run in parallel till the master thread reaches the same start address of the

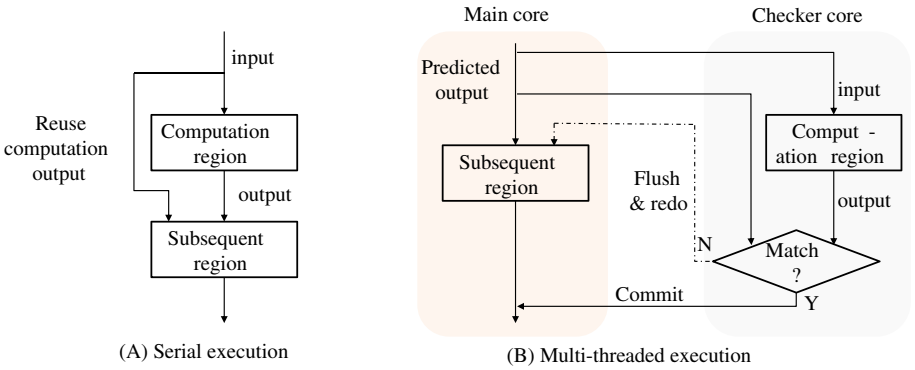

FIGURE 16 Computation reuse and/or VP on serial and multithreaded architectures[59]

speculative thread. At this point, dependence violation is checked and based on it, the master core either commits the results or begins recovery. Their compiler performs selective value profiling. Based on analysis of data predictability, it uses specific VP schemes at the location where the values will be used. Before forking the speculative thread, the master thread predicts the value for it and communicates the value to it. The compiler uses a cost-benefit analysis to select variables for prediction which have high prediction accuracy and require re-execution of smallest number of instructions on misprediction. The compiler also tries to reduce code-size of the predictor. By analyzing the dataflow graph, the compiler can also select the variables for prediction which are not directly predictable but depend on other variables that can be predicted. By virtue of allowing more speculative computation, their technique improves performance of parallel-threaded applications.

6.2 | VP in trace processors

Lee et al[71] note that VP-based techniques assume that each fetched instruction accesses the VP table simultaneously in the instruction fetch stage. However, in wide-issue (e.g., 8 to 16 width) superscalar processors, many branch predictions and concurrent accesses to VP-table are required which necessitates a table with several read/write ports. Using a banked-design of the table[32] can lead to bank conflicts. Further, accessing VP table in fetch stage leads to both beneficial and unbeneficial instructions accessing the VP table since instructions are not decoded in the fetch stage.

To address these limitations, they propose a technique for processors with trace cache which efficiently integrates multiple branch predictions and speculative execution. In their technique, VP happens in the same stage when VP table is updated (e.g., writeback stage) and predicted values are fed to VP buffer of the trace cache. At the time of fetching instructions from the trace cache, speculated values in the buffer are also read. They use a hybrid predictor with dynamic classification.[19] Their technique reduces bandwidth to VP table since the instruction-types are already known before accessing the VP table. For example, VP is performed only for load instructions and those having data-dependent instructions in the same trace cache block. Their technique improves performance while reducing the hardware overhead of VP. However, the applicability of their technique is limited to trace processors.

Bhargava et al[25] study VP in a wide-issue high-frequency processor with realistic values of predictor port-count, latency, and energy values. They note that performing VP at instruction-fetch or after instruction-decode suffers from latency penalty of predictors. A lower-latency alternative to these two approaches is to perform VP right after retirement of the previous instance of an instruction. In this approach, the predicted values are stored in a separate "predicted value cache" (PVC) and are used by the next instance of the instruction. However, this approach suffers from staleness and need of accessing a centralized VP table. They propose an energy and latency aware VP technique which keeps predicted values in trace form and performs trace-based updates in the fill unit. For allowing this, PVC also holds stride information with the predicted value. Their technique replaces the centralized VP table with a queue of prediction traces. This queue consumes lower energy since it is accessed less frequently, needs only one read port and one write port and is direct mapped. They experiment with a stride and a two-delta stride predictor. Their technique reduces energy overhead of VP while maintaining high performance, although complex predictors such as the context predictor cannot be implemented with their technique due to simplified hardware.

6.3 | VP for trace-compression

Burtscher et al[50] note that an extended program trace (i.e., that containing both PCs and additional data-items such as register-contents, effective addresses, etc.) is more difficult to compress than a PC-only trace since these data-items vary over larger range and repeat less frequently. They propose using VP to compress such traces effectively. They use multiple prediction schemes, e.g., last-*K* and global/local correlation last-value predictors for extended data, stride 2-delta, FCM and DFCM VP for both PCs and extended data, etc. To perform compression with any predictor, each trace entry is compared with the predicted value and "1" is written on a correct prediction and "0" followed by the trace entry is written otherwise. Their compression algorithm first compresses PCs using different predictors and in case more than one predictor is correct, chooses one with the smallest length. Predictor number is encoded using dynamic Huffman encoder. For unpredictable PCs, only $\log_2$ *Range* bits are written since

the remaining bits are zero. For unpredictable extended data-items, the number for the predictor whose prediction is closest to the actual value is written followed by the difference between the actual and the predicted value.

They also perform several optimizations to the predictors to improve the compression ratio. Implementing VP in software allows using more complex predictors than those possible in a hardware implementation. Their algorithm achieves higher compression ratio than the traditional algorithms and performs compression in a single-pass while requiring only fixed amount of memory. Burtscher[55] also presents another technique which uses VP to highlight and augment patterns in the traces to allow them to be better compressed using traditional compression algorithms. Compared to traditional compressors, their approach provides higher compression ratios and reduced latency of compression and decompression.

6.4 | VP for prefetching

Zhou et al[51] note that for memory-intensive workloads with substantial pointer-chasing, data-dependency between the loads forces serial execution. VP can allow parallelizing these loads, and thus, for such workloads, VP is mainly useful for exploiting memory level parallelism (MLP) and not ILP. They propose using VP only for speculative pre-execution for prefetching to L1 data cache. Conventional VP techniques commit/squash the speculative state upon a correct/incorrect prediction (respectively), whereas their technique uses speculated results for prefetching only and does not commit it. This avoids the need of VP-validation and misprediction recovery schemes. Hence, their technique requires smaller changes to hardware and also enables memory disambiguation for prefetching. In case of correct prediction, their pre-execution approach does not remove the requirement of re-execution; however, such re-executions happen fast since the data is already in cache. Their technique enhances MLP and improves performance even with a simple stride predictor.

7 | TECHNIQUES FOR REDUCING OVERHEAD OF VP

In this section, we discuss the works aimed at reducing modifications to the OOO engine (Section 7.1), reducing predictor storage and port-count requirement (Section 7.2), reducing VP misprediction penalty (Section 7.3). We also discuss techniques for exploiting narrow values (Section 7.4), using partial-tags (Section 7.5), using bit-sharing (Section 7.6), and using cache for predicted values (Section 7.7). Some of these techniques reduce coverage,[28] while others increase it.[26] Table 3 provides the overview of these works.

7.1 | Reducing modifications to OOO engine

Perais et al[37,76] present a VP technique which combines performance benefits of out-of-order designs with energy benefits of in-order designs. Figure 17 shows the pipeline diagram and modifications made by their technique. Their technique uses both early- and late-execution approach to reduce modifications to OOO engine. On using VP, the operands of several single-cycle instructions become ready in the front-end. They propose early in-order execution of these instructions in parallel with rename by using immediate or/and predicted operands. These instructions are executed by the OOO engine which allows reducing the instruction window size. They show that with very-accurate predictors, misspeculations need not be detected at execute stage. Instead, validation may be performed in-order, during commit stage; and on a misspeculation, the pipeline can be fully squashed. This obviates the need of replaying instructions directly from instruction queue on a misspeculation. This lowers the complexity and ensures that the operands of committed early-executed instructions are correct.

Similarly, they offload the execution of predicted instructions to a separate in-order late execution stage where "select and wakeup" does not happen since the validation of predicted results can be performed at commit time. Instructions dependent on predicted instructions use the predicted results instead of waiting in the scheduler. Several high-confidence branches have high predictability, and hence, they are also assigned to the late-execution stage. Their predictor is termed as value tagged geometric (VTAGE) predictor.

In summary, their technique performs both prediction-validation and misprediction recovery outside OOO engine, at commit stage instead of execution stage. Also, for misprediction recovery, their technique allows using pipeline squashing, which is much simpler than selective replay. While maintaining similar performance as an OOO processor with VP, their VP technique reduces issue width and complexity of RF. They also propose hybridizing their VTAGE predictor with the two-delta stride predictor.[16] Overall, their technique allows using VP without altering the OOO engine.

TABLE 3 Techniques for reducing VP implementation overhead

Category	References
Exploiting narrow values	22,28,44,75
Partial tag	26,27
Bit sharing	28
Using limited bits in stride	20,58
Caching predicted values	25,34

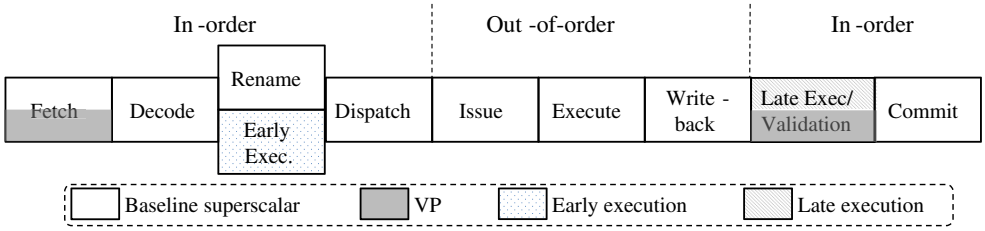

FIGURE 17 Pipeline diagram in the technique of Perais et al[37]

7.2 | Reducing port-count and storage requirement

A limitation of the technique of Perais et al[37] is the requirement of multi-ported structures and large predictor table. Perais et al[58] propose another technique to address these limitations. Their technique relates speculated values with a fetch block instead of unique instructions, similar to the current instruction-fetch scheme. This allows predicting several instructions in each cycle while using only single-ported components, since only a single predictor entry is used for all speculations related to an instruction fetch. By just once accessing the predictor with PC of instruction fetch block, entire set of predictions can be obtained.

Modern superscalar processors fetch and decode multiple instructions in each cycle and in variable-length ISAs, the number of values produced by an instruction are known after many cycles. Hence, a value predictor entry cannot be associated with a unique PC. To perform VP for all outcomes of all instructions fetched in a cycle, a multi-ported VP structure is required. To avoid this, in their technique, the predictor is accessed using the "fetch-block PC," i.e., PC right-shifted by $\log_2(SizeOfFetchBlock)$ bits and some global information. Thus, each predictor entry now stores multiple values.

They further propose a differential-VTAGE (D-VTAGE) predictor which stores strides instead of whole values (similar to DFCM[36]). It uses partial strides (e.g., 8 bits) and hence has a moderate storage budget comparable to branch predictor or I-cache. They also show that block-based VP allows implementing the checkpoint-scheme required for supplying last predicted/computed values to D-VTAGE predictor with low overhead. Their experiments confirm the effectiveness of their technique.

Huang et al[62] note that the inputs and outputs of a chain of instructions show high regularity and predictability. To exploit such value locality, they propose performing value reuse at the granularity of a basic block. Thus, performing prediction and reuse at basic block granularity instead of instruction granularity reduces hardware overhead and also improves performance. They use compiler to mark the instructions that are live at the completion of a basic block. For each basic block, they store upward-exposed inputs and live outputs from the last four executions in a block history buffer. Then, if the present inputs to a basic block are same as any of the last four inputs, the instructions in the BB need not be executed and that output itself can be taken. They note that compiler schemes such as function inlining and loop-unrolling do not impact the value locality. Overall, their technique improves performance significantly.

7.3 | Reducing misprediction penalty

Nakra et al[65] note that in "very large instruction word" (VLIW) processors, the code is statically scheduled and hence, VP can be used only with the instructions selected at compile-time. On a misprediction, incorrectly predicted instructions are re-executed and this re-executed code is referred to as "compensation code" (CCode). Their technique seeks to reduce the performance penalty due to CCode. It generates CCode dynamically during execution of VLIW code with value-predicted operations. At runtime, whenever VP is done, CCode is generated and stored in a dedicated storage. This alleviates the need of statically scheduling CCode with the remaining VLIW code. This also avoids the increase in code-size, alteration in instruction schedule and pollution of instruction cache by the CCode blocks. Further, they use a separate "CCode engine" which runs concurrently with the VLIW engine. These engines are synchronized to ensure that VLIW engine is stalled on a mis-speculation only if it needs a value that has not yet been generated by the CCode engine. Otherwise, VLIW engine continues running and CCode engine concurrently re-executes any mispredicted operations. This reduces the impact of CCode on the execution of VLIW code. Their experiments confirm the effectiveness of their technique.

7.4 | Exploiting narrow values

In hybrid predictors, different predictor components are aimed at different *value patterns* (e.g., strided vs. constant), whereas in the techniques discussed next,[22,44] different predictor tables are aimed at values with *different widths* (e.g., narrow vs. wide).

Sato et al[22] exploit narrow-width values for reducing the hardware overhead of VP tables. They note that most highly predictable instructions have narrow values and the instructions with wide values are already difficult to predict. Based on this, their technique uses a narrow and a wide predictor. Narrow predictor keeps only low-order bits of values in the history table and is used for predicting narrow values (e.g., 8b). Wide predictor is used for predicting wide values (e.g., 32b). Narrow and wide predictors have fully-associative and direct-mapped VHTs, respectively. Both predictors

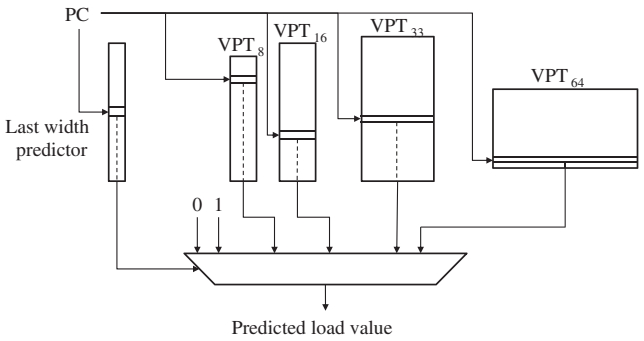

FIGURE 18 The width-partitioned last-value predictor.[44] VPT_k refers to the VP-table with a width of *k*-bits

use the same prediction scheme. For any instruction, the history is stored in only one of the two predictors. Since most instructions are predicted by the narrow predictor, the total hardware budget is reduced. Their technique maintains performance while incurring only small loss in predictability.

Loh et al[44] propose a technique for reducing energy by exploiting narrow width values and locality property of data-widths. They use six width-ranges, e.g., 0, 1, 2-8, 9-16, 17-33, and 34-64 bits. Since the first two ranges (0 and 1 bits) have only a single value, no tables are required for them. Hence, they use four tables for the last four ranges and the widths of these tables are the maximum bits of those ranges (i.e., 8, 16, 33, and 64, respectively), as shown in Figure 18. The number of entries in these tables can be different. Their technique predicts the data-width using a last-width predictor and then retrieves value from only the corresponding table. Last-width prediction works on the observation that the width of the outcomes of successive dynamic instances of a static instruction are generally the same. For width-prediction, they use a PC-indexed table which records the most recent data-width of a load value and predicts this to be the width of the next load value. Their technique saves energy by virtue of reducing storage space and accessing only a single smaller table (instead of a full 64-bit table). The limitation of their technique is that it predicts the width using last-width prediction approach, which incurs storage overhead of an additional table. Also, when the widths of successive values change frequently, the effectiveness of last-width prediction reduces greatly.

7.5 | Using partial tags

Sato et al[26] propose using only few bits of the instruction address (i.e., tag) in the VP-table since a small loss in prediction accuracy is acceptable. This approach leads to aliasing and reduced prediction accuracy, although it increases the prediction coverage. With increase in table size, the reduction in prediction accuracy due to decreasing tag size also reduces since these predictors have few conflicts. They observe that although no-resolution (i.e., 0-bit tag) reduces performance significantly, 2-bit tag provides more than 80% performance of the full tag size. Similarly, since a wrong VP only affects performance and not application-correctness, Burtscher et al[27] omit tag and valid bits in a value-predictor to reduce hardware overhead and complexity, since the tag/valid bits provide only small increase in accuracy.

7.6 | Using bit-sharing

Salehi et al[28] propose two techniques to reduce the storage overhead of VP. First, since the predicted values are narrow, they predict at 8-bit granularity instead of 32-bit granularity, although due to this, values more than 8-bit wide cannot be predicted. Still, they show that the reduction in coverage is small. Thus, while other techniques [22,44] use multiple tables to predict both narrow and wide values, the technique of Salehi et al[28] predicts only narrow values. Second, since some values appear more frequently than others, they propose sharing bits among the predictor entries. They propose two variants of the technique, sharing multiple bits between two consecutive entries or sharing one bit between multiple entries. While bit sharing reduces storage, it also reduces the predictor coverage. Another limitation of bit sharing is that any predictor entry that shares the bit can update the shared bits, which changes the value for the remaining sharing partners also. Their techniques reduce the storage overhead of VP significantly while incurring negligible decrease in the predictor coverage.

7.7 | Using a cache for predicted values

Lee et al[34] propose techniques to address two challenges in VP: large number of accesses to VP table and the latency of updating VP table. They use a "predicted value cache" (PVC) which holds predicted values and is organized exactly like the I-cache. This allows predicting value of each instruction in the same way it is fetched from the I-cache. PVC provides required bandwidth to access multiple predictions required in each cycle. They use a hybrid value predictor with dynamic classification. Each prediction table has 2 read/write ports and a queue for buffering the accesses. They also use a VP mechanism[71] decoupled from the fetch stage. This enables using VP at a later (writeback) stage and reducing the time between VP updates and their use. PVC allows accessing multiple prediction values without the overhead of accessing both the branch prediction tables and VP tables. It also alleviates bank conflicts between multiple accesses to the VP table since both I-cache and PVC have the same organization. To address the

problem of delayed update, they examine schemes such as speculative update and use of age counters to identify the stale values, however, the effectiveness of these schemes is limited due to the latency of dynamic classification.

8 | CONCLUSION AND FUTURE WORK

In this paper, we surveyed several techniques for leveraging value locality and performing value prediction. We organized the works based on several key parameters and discussed the techniques for improving VP accuracy and reducing its implementation overhead. We close the paper with a brief mention of future research directions and challenges.

8.1 | Exploiting VL in other scenarios

The research works reviewed here study VL arising between the outcomes of different *dynamic* instances of the same *static* instruction. Similarly, instructions executed in a dynamic instruction window typically produce the same result, especially in case of integer results. VL may also arise due to register-to-register move operations, trivial computations (e.g., A*B where A or B =0), etc. This type of VL is discussed in detail by a recent survey.[30]

Wen et al[61] show that *different instructions* may also write the *same value* to a location (e.g., a register) which leads to value locality. Examples of this type of VL are shown in Figure 19. Their technique tracks the values stored in every storage location in memory and registers due to the data movement and computations. Then, it checks whether a newly generated value is same as the previously stored value at that location. If so, the operation is marked as redundant. Their technique also records the frequency of such redundancy and then focuses on locations where redundancy occurs with high frequency. Their technique removes redundant operations and if the values stored in nearby locations are identical, the computation result from one location is reused for another. Evidently, the optimizations possible by exploiting VL are much richer than that could be summarized in this paper. We hope future research will explore them in greater detail.

8.2 | VP in shared-memory systems

Prediction approach is also useful for shared memory systems, for instance, in reducing the communication overhead, improving efficiency of coherence protocols, etc. The following works illustrate this idea.

Mukherjee et al[67] note that coherence message patterns connected to specific cache block addresses remain stable over program execution and hence, can be predicted with high accuracy. They present a two-level predictor for predicting such patterns for improving the efficiency of coherence protocols. In their predictor, the first and second level tables are termed as "message history" and "pattern history" tables (MHT and PHT), respectively. Their predictor indexes into MHT using the cache block address and obtains a single or multiple <sender-processor, message-type> tuples corresponding to the last few coherence messages received for that cache block. The sender processor signifies the specific sharers in the sharing pattern and message-type shows the coherence actions for a sharing pattern. Using these tuples, PHT is accessed and a <processor, message-type> prediction for the source and type of the next coherence message for a cache block is obtained. They show that their predictor achieves high accuracy.

In shared memory systems, the destination set refers to the group of processors receiving a given coherence request. In snooping protocols, the request is sent to all the processors for reducing the latency whereas in directory protocols, a directory is consulted to send request to only a minimal destination set for reducing the bandwidth. In hybrid protocols, the destination set is predicted for achieving the best of the two protocols. These predictors speculate which processors will see a given coherence request for a certain sharing pattern. For example, for MOESI write-invalidate protocol, a write-request (i.e., request for exclusive) finds the owner and all the sharers whereas a read-request (i.e., request for shared) finds the current owner. Martin et al[66] propose multiple destination-set predictors and evaluate them for multicast snooping protocols. They show that their predictors allow exercising tradeoff between latency and bandwidth.

Acacio et al[68] note that a large fraction of L2 misses in shared memory systems are "cache-to-cache transfer misses." In cache coherent non-uniform memory architecture (CC-NUMA) designs, directory access to find the owner node (the node which stores the single valid copy of the block) happens in critical path of these misses which increases their latency significantly. Figure 20A shows the handling of such "3-hop" misses in traditional systems. On receiving a request from a node (P1), the home directory (P2) forwards the request to its owner node (P3). The owner node sends a copy of the block to the requesting node (P1), along with a message to the directory (P2) and also updates the state of its own copy of the

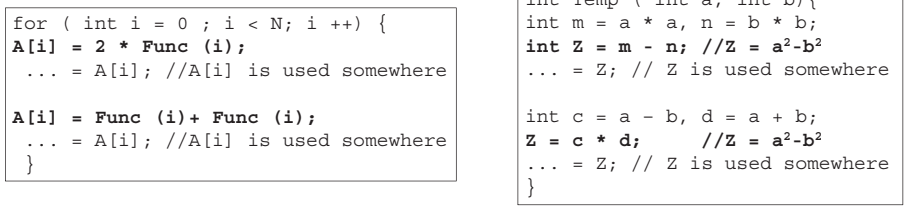

FIGURE 19 Examples of VL arising due to redundant computations or multiple instructions leading to the same final result[61]

block. Acacio et al[68] note that only few instructions are responsible for such 3-hop misses and the candidate owner nodes of missing blocks are also small in number and generally the same. Based on this, they propose a technique by which the requesting node predicts the owner node. Then, as shown in Figure 20B, it directly sends the request to the owner core similar to snooping-based protocols. This converts 3-hop misses into 2-hop misses which provides large improvement in performance.

Clearly, prediction is a versatile and powerful optimization tool. Further research on VP in shared memory systems should leverage VP to overcome synchronization overheads while also addressing semantics issues.[77]

8.3 | VL in GPUs and VP techniques for GPUs

VL also arises in GPUs.[78] In the single-instruction multiple-thread (SIMT) execution model of GPUs, multiple (e.g., 32) threads form a group called a "warp" and all the threads in a warp execute the same instruction. In several cases, the results of all the thread-computations may be same, e.g., due to constant/zero operands, inherent redundancy in data (e.g., image/video data), same control operations on the threads, etc. Thus, in GPUs, the same instruction *concurrently* executed by different threads of a warp may produce the same result which leads to VL. Here, VL can be exploited using the computation reuse idea, i.e., without requiring the speculative VP technique. For example, such instructions can be executed on a separate scalar pipeline and its operands/results can be stored in a scalar RF.[79] This saves both computation and storage energy.

Some researchers have proposed VP techniques for GPUs. Sun et al[53] present a software-based VP technique for GPUs for boosting applications without data parallelism since they cannot benefit from the parallel hardware of GPU. They use a hybrid stride+context predictor. Based on the history values, *K* data values of the kernel are predicted and the kernel is computed with *K* parallel threads on GPU. The computation results of the kernel are recorded in a prediction table. Before beginning of a new iteration, the fresh data value is searched in the table and on a hit, the results are retrieved from the table. On a miss, if the miss-count is found to exceed a threshold, the table update request is issued. Several GPU threads are started for obtaining new kernel results based on the new predicted values. These new results update the prediction table. The GPU computes the kernel with predicted input values while CPU identifies scope for speculating based on the table. If the output of a kernel depends on both the current input and the previous inputs, they are stored as the combination of multiple inputs in a single prediction entry in the table. Figure 21 summarizes the working of their technique. Despite the overhead of software-based VP and additional CPU-GPU communication, their technique improves performance of parallelized kernel loops with data dependencies and complex pointer operations.

Several features of GPUs make them suitable for implementing VP. While CPUs require few micro or even milliseconds to create a thread, on GPUs, thousands of threads can be generated in few cycles. Also, GPUs can provide much higher performance per watt compared to CPUs and hence, VP can be more energy efficient on GPUs than on CPUs. Also, achieving high memory bandwidth and large parallelism is even more important in GPUs than in CPUs and hence, VP is very promising for GPUs. However, CPUs and GPUs differ fundamentally in terms of their architectures and optimization objectives. Hence, the techniques proposed in CPU context may not be applicable or optimal when used in GPUs. Given this, porting the existing VP techniques to GPUs and designing novel GPU-specific VP techniques will be a major challenge for researchers in near-future.

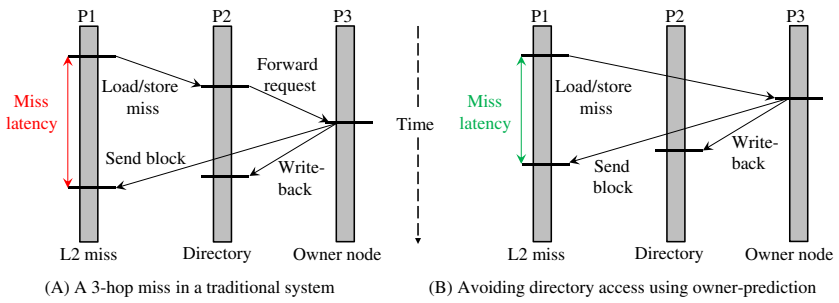

FIGURE 20 Handling of a cache-to-cache transfer miss in A, traditional CC-NUMA system and B, a CC-NUMA system with owner prediction[68]

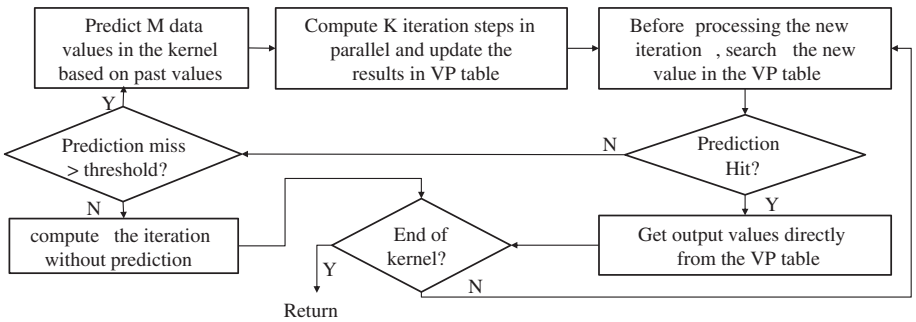

FIGURE 21 Design flow of the technique of Sun et al[53]

8.4 | Approximate VP techniques

In contrast with accurate VP, approximate VP[61,80] allows significantly lowering the implementation overhead of VP, e.g., rollback on misprediction can be avoided and the storage for predictor tables can be reduced.

San Miguel et al[81] note that fetching data on a load miss from the next cache level or memory incurs high latency. By exploiting approximable nature of applications, the load values can be predicted which allows the processor to continue without waiting for a response. They present an load value approximation technique for graphics programs. Compared to the conventional load VP techniques, where a line needs to be fetched *on every cache miss* to verify the prediction, their technique fetches the blocks *occasionally* merely for training the approximator. This reduces the memory accesses significantly. Also, given the error-tolerant nature of graphics applications, if the values predicted by the approximator does not match the exact value, a rollback is not required. Using CE scheme, approximation is made only when the accuracy of approximator is sufficiently high. While incurring negligible loss in in output quality, their technique provides large improvement in performance and energy efficiency.

A key challenge in approximate VP, however, is that it is only applicable to error-tolerant applications/code-portions, e.g., graphics applications. Developing strategies to extend the scope of approximate computing to VP techniques will be an interesting and rewarding challenge for the architects.

8.5 | Synergistic integration with related techniques

To improve the accuracy of VP and reduce its overhead, VP efficiency can be enhanced by integrating it with related techniques and leveraging the ideas from related fields. For example, the amount of VL present in the application can be enhanced using compiler management techniques such as instruction scheduling, memory layout restructuring, code reordering, etc.[41,59,82,83] For example, instructions that consume a potential predicted value should remain close to their producer which allows using the predicted values before the computation of actual results.[41] Further, in branch prediction field, neural branch predictors have been shown to provide high accuracy. As such, further research into neural based value predictors such as perceptron, path-based and back-propagation predictors can help in improving VP accuracy.[84] Further, many cores on modern processors remain idle which can be leveraged for executing a helper thread. This thread executes reduced (or full) version of the program and as such, always runs ahead of the main thread. Hence, it pre-computes the value which achieves the effect of VP while providing the values in time.[52,85] Clearly, a vast space of opportunity remains yet to be explored.

8.6 | Need for comprehensive evaluation of VP techniques

Several research works evaluate only prediction accuracy and not performance.[7,11,36,46,69] However, it has been shown that high VP accuracy does not necessarily lead to improved performance.[19] This is because most studies ignore the impact of recovery schemes and the common predictors such as FCM[7] and DFCM[36] are effective for loads that are not crucial for performance.[47] Hence, evaluating VP accuracy without evaluating its impact on performance may provide a misleading picture of the potential of VP. To address this issue, future studies need to evaluate the VP techniques on both accuracy and performance.

8.7 | Integration of VP in real processors

Except few techniques,[50,55,61] remaining techniques summarized in the paper have been evaluated only on the simulators. Implementation and evaluation of VP based techniques on real processor is required to prove their feasibility and gain deeper insights. Since real-world processors employ several management techniques for register file, caches, etc., implementation of VP in processors will also depend on the effective integration of VP with existing processor management policies.

ACKNOWLEDGEMENTS

This work was supported in part by the Science and Engineering Research Board (SERB), India under grant ECR/2017/000622.

ORCID

Sparsh Mittal http://orcid.org/0000-0002-2908-993X

REFERENCES

1. Lipasti MH, Wilkerson CB, Shen JP. Value locality and load value prediction. *ACM SIGPLAN Notices*. 1996;31(9):138-147.
2. Gabbay F, Mendelson A. Using value prediction to increase the power of speculative execution hardware. *ACM TOCS*. 1998;16(3):234-270.
3. Liu S, Gaudiot JL. Potential impact of value prediction on communication in many-core architectures. *IEEE Trans Comput*. 2009;58(6):759-769.
4. Lepak KM, Lipasti MH. On the value locality of store instructions. Paper presented at: ISCA; Vancouver, British Columbia, Canada; 2000.
5. Marcuello P, González A, Tubella J. Thread partitioning and value prediction for exploiting speculative thread-level parallelism. *IEEE Trans Comput*. 2004;53(2):114-125.

6. Sam NB, Burtscher M. Complex load-value predictors: Why we need not bother. Paper presented at: Workshop on Duplicating, Deconstructing, and Debunking; Madison, Wisconsin, USA; 2005.
7. Sazeides Y, Smith JE. The predictability of data values. Paper presented at: MICRO; Research Triangle Park, North Carolina, USA; 1997.
8. Fu C, Jennings MD, Larin SY, Conte TM. Value speculation scheduling for high performance processors. Paper presented at: ASPLOS; San Jose, California, USA; 1998.
9. Mittal S. A survey of recent prefetching techniques for processor caches. *ACM Comput Surv*. 2016;49(2):35:1-35:35.
10. Zhao Q, Lilja DJ. Static classification of value predictability using compiler hints. *IEEE Trans Comput*. 2004;53(8):929-944.
11. Wang K, Franklin M. Highly accurate data value prediction using hybrid predictors. Paper presented at: MICRO; Research Triangle Park, North Carolina, USA; 1997.
12. Calder B, Reinman G, Tullsen DM. Selective value prediction. Paper presented at: ISCA; Atlanta, Georgia, USA; 1999.
13. Sodani A, Sohi GS. Understanding the differences between value prediction and instruction reuse. Paper presented at: MICRO; Dallas, Texas, USA; 1998.
14. Tullsen DM, Seng JS. Storageless value prediction using prior register values. Paper presented at: ISCA; Atlanta, Georgia, USA; 1999.
15. Burtscher M, Zorn BG. Exploring last n value prediction. Paper presented at: PACT; Newport Beach, California, USA; 1999.
16. Eickemeyer RJ, Vassiliadis S. A load-instruction unit for pipelined processors. *IBM J Res Dev*. 1993;37(4):547-564.
17. Yeh TY, Patt YN. A comparison of dynamic branch predictors that use two levels of branch history. Paper presented at: ISCA; Orlando, Florida, USA; 1993.
18. Sazeides Y, Smith JE. Modeling program predictability. Paper presented at: ISCA; Barcelona, Spain; 1998.
19. Rychlik B, Faistl J, Krug BP, et al. Efficient and accurate value prediction using dynamic classification. CMμART-1998-01, Carneige Mellon University; Pittsburgh, Pennsylvania, USA; 1998.
20. Rychlik B, Faistl J, Krug B, Shen JP. Efficacy and performance impact of value prediction. Paper presented at: PACT; Paris, France; 1998.
21. Thomas R, Franklin M. Characterization of data value unpredictability to improve predictability. Paper presented at: International Workshop on Workload Characterization (WWC); Austin, Texas, USA; 2001.
22. Sato T, Arita I. Table size reduction for data value predictors by exploiting narrow width values. Paper presented at: International Conference on Supercomputing; Santa Fe, New Mexico, USA; 2000.
23. González J, González A. The potential of data value speculation to boost ILP. Paper presented at: ICS; Melbourne, Australia; 1998.
24. Gellert A, Palermo G, Zaccaria V, Florea A, Vintan L, Silvano C. Energy-performance design space exploration in SMT architectures exploiting selective load value predictions. Paper presented at: DATE; Paris, France; 2010.
25. Bhargava R, John LK. Latency and energy aware value prediction for high-frequency processors. Paper presented at: ICS; New York, NY, USA; 2002.
26. Sato T, Arita I. Partial resolution in data value predictors. Paper presented at: ICPP; Toronto, Canada; 2000.
27. Burtscher M, Zorn BG. Prediction outcome history-based confidence estimation for load value prediction. *J Instruction-Level Parallelism*. 1999;1.
28. Salehi M, Baniasadi A. Storage-aware value prediction. Paper presented at: Euromicro Conference on Digital System Design; Lille, France; 2010.
29. Larson E, Austin T. Compiler controlled value prediction using branch predictor based confidence. Paper presented at: MICRO; Monterey, California, USA; 2000.
30. Mittal S. A survey of techniques for designing and managing CPU register file. *Concurrency Computat: Pract Exper*. 2017;29(4):e3906. https://doi.org/10.1002/cpe.3906
31. Markovski J, Gusev M. Why value prediction is limited? Paper presented at: International Conference on Information Technology Interfaces; Cavtat, Croatia; 2004.
32. Gabbay F, Mendelson A. The effect of instruction fetch bandwidth on value prediction. Paper presented at: ISCA; Barcelona, Spain; 1998.
33. Zhao Q, Lee SJ, Lilja D. Using hyperprediction to compensate for delayed updates in value predictors. *IEE Proc-Comput Digital Tech*. 2005;152(5):596-608.
34. Lee SJ, Yew PC. On some implementation issues for value prediction on wide-issue ILP processors. Paper presented at: PACT; Philadelphia, Pennsylvania, USA; 2000.
35. Zhou H, Flanagan J, Conte TM. Detecting global stride locality in value streams. Paper presented at: ISCA; San Diego, California, USA; 2003.
36. Goeman B, Vandierendonck H, De Bosschere K. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. Paper presented at: HPCA; Nuevo Leone, Mexico; 2001.
37. Perais A, Seznec A. EOLE: Paving the way for an effective implementation of value prediction. Paper presented at: ISCA; Minneapolis, Minneapolis, USA; 2014.
38. Tuck N, Tullsen DM. Multithreaded value prediction. Paper presented at: HPCA; San Francisco, California, USA; 2005.
39. Sato T, Hamano A, Sugitani K, Arita I. Influence of compiler optimizations on value prediction. Paper presented at: International Conference on High-Performance Computing and Networking; Amsterdam, The Netherlands; 2001.
40. Lipasti MH, Shen JP. Exceeding the dataflow limit via value prediction. Paper presented at: MICRO; Paris, France; 1996.
41. Endo F, Perais A, Seznec A. On the interactions between value prediction and compiler optimizations in the context of EOLE. *ACM Trans Archit Code Optim*. 2017.
42. Burtscher M, Zorn BG. Hybrid load-value predictors. *IEEE Trans Comput*. 2002;51(7):759-774.
43. Ceze L, Strauss K, Tuck J, Torrellas J, Renau J. CAVA: Using checkpoint-assisted value prediction to hide L2 misses. *ACM Trans Archit Code Optim (TACO)*. 2006;3(2):182-208.
44. Loh G. Width prediction for reducing value predictor size and power. Paper presented at: Value Prediction Workshop; San Diego, California, USA; 2003.
45. Gabbay F. Speculative execution based on value prediction. Technical Report 1080, Technion - Israel Institute of Technology; Haifa, Israel; 1996.
46. Nakra T, Gupta R, Soffa ML. Global context-based value prediction. Paper presented at: HPCA; Orlando, FL, USA; 1999.
47. Burtscher M, Diwan A, Hauswirth M. Static load classification for improving the value predictability of data-cache misses. *ACM SIGPLAN Notices*. 2002;37(5):222-233.

48. Li XF, Du ZH, Zhao Q, Ngai TF. Software value prediction for speculative parallel threaded computations. Paper presented at: Value Prediction Workshop; San Diego, California, USA; 2003.
49. Ghandour WJ, Akkary H, Masri W. Leveraging strength-based dynamic information flow analysis to enhance data value prediction. *ACM Trans Archit Code Optim (TACO)*. 2012;9(1):1:1-1:33.
50. Burtscher M, Jeeradit M. Compressing extended program traces using value predictors. Paper presented at: PACT; New Orleans, LA, USA; 2003.
51. Zhou H, Conte TM. Enhancing memory-level parallelism via recovery-free value prediction. *IEEE Trans Comput*. 2005;54(7):897-912.
52. Tanaka Y, Ando H. Reducing register file size through instruction pre-execution enhanced by value prediction. Paper presented at: ICCD; Lake Tahoe, California, USA; 2009.
53. Sun E, Kaeli D. Aggressive value prediction on a GPU. *Int J Parallel Program*. 2014;42(1):30-48.
54. Ibrahim KZ, Byrd GT. On the exploitation of value prediction and producer identification to reduce barrier synchronization time. Paper presented at: International Parallel and Distributed Processing Symposium. IEEE; San Francisco, California, USA; 2001.
55. Burtscher M. Vpc3: A fast and effective trace-compression algorithm. *Paper presented at: ACM SIGMETRICS Performance Evaluation Review*. 2004;32.
56. McFarling S. Combining branch predictors. Technical Report, Digital Western Research Laboratory; Palo Alto, California, USA; 1993.
57. Gabbay F, Mendelson A. Can program profiling support value prediction? Paper presented at: MICRO; Research Triangle Park, North Carolina, USA; 1997.
58. Perais A, Seznec A. BeBoP: A cost effective predictor infrastructure for superscalar value prediction. Paper presented at: HPCA; Burlingame, California, USA; 2015.
59. Wu Y, Chen DY, Fang J. Better exploration of region-level value locality with integrated computation reuse and value prediction. Paper presented at: ISCA; Göteborg, Sweden; 2001.
60. Burtscher M, Zorn BG. Hybridizing and coalescing load value predictors. Paper presented at: ICCD; Austin, Texas, USA; 2000.
61. Wen S, Chabbi M, Liu X. REDSPY: Exploring value locality in software. Paper presented at: International Conference on Architectural Support for Programming Languages and Operating Systems. ACM; Xi'an, China; 2017.
62. Huang J, Lilja DJ. Exploiting basic block value locality with block reuse. Paper presented at: HPCA; Raleigh, North Carolina, USA; 1999.
63. Fu C, Conte TM. Value speculation mechanisms for EPIC architectures. Technical Report, North Carolina State University; Raleigh, North Carolina, USA; 1998.
64. Fu C, Jennings MD, Larin SY, Conte TM. Software-only value speculation scheduling. Technical Report, North Carolina State University; 1998.
65. Nakra T, Gupta R, Soffa ML. Value prediction in VLIW machines. Paper presented at: ISCA; Atlanta, Georgia, USA; 1999.
66. Martin MM, Harper PJ, Sorin DJ, Hill MD, Wood DA. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. Paper presented at: ISCA; San Diego, California, USA; 2003.
67. Mukherjee SS, Hill MD. Using prediction to accelerate coherence protocols. Paper presented at: ISCA; Barcelona, Spain; 1998.
68. Acacio ME, González J, García JM, Duato J. Owner prediction for accelerating cache-to-cache transfer misses in a CC-NUMA architecture. Paper presented at: ACM/IEEE Supercomputing Conference; Baltimore, Maryland, USA; 2002.
69. Thomas R, Franklin M. Using dataflow based context for accurate value prediction. Paper presented at: PACT; Barcelona, Catalunya, Spain; 2001.
70. Seng J, Hamerly G. Exploring perceptron-based register value prediction. Paper presented at: Second Value Prediction and Value-Based Optimization Workshop, held in conjuction with ASPLOS; Boston, MA, USA; 2004.
71. Lee SJ, Wang Y, Yew PC. Decoupled value prediction on trace processors. Paper presented at: HPCA; Toulouse, France; 2000.
72. Fields B, Rubin S, Bodík R. Focusing processor policies via critical-path prediction. Paper presented at: ISCA; Göteborg, Sweden; 2001.
73. Sazeides Y, Smith JE. Implementations of context based value predictors. Technical Report ECE-97-8, University of Wisconsin-Madison; Madison, Wisconsin, USA; 1997.
74. Sodani A, Sohi GS. Dynamic instruction reuse. Paper presented at: ISCA; Boulder, CO, USA; 1997.
75. Balakrishnan S, Sohi GS. Exploiting value locality in physical register files. Paper presented at: International Symposium on Microarchitecture; San Diego, CA, USA; 2003.
76. Perais A, Seznec A. Practical data value speculation for future high-end processors. Paper presented at: HPCA; Orlando, FL, USA; 2014.
77. Boehm HJ, Demsky B. Outlawing ghosts: avoiding out-of-thin-air results. Paper presented at: Workshop on Memory Systems Performance and Correctness; Edinburgh, Scotland; 2014.
78. Mittal S. A survey of techniques for architecting and managing GPU register file. *IEEE Trans Parallel Distrib Syst (TPDS)*. 2017;28(1):16-28.
79. Gilani SZ, Kim NS, Schulte MJ. Power-efficient computing for compute-intensive GPGPU applications. Paper presented at: HPCA; Shenzhen, China; 2013.
80. Mittal S. A survey of techniques for approximate computing. *ACM Comput Surv*. 2016;48(4):62:1-62:33.
81. Miguel JS, Badr M, Jerger EN. Load value approximation. Paper presented at: MICRO; Cambridge, United Kingdom; 2014.
82. Zhang Y, Yang J, Gupta R. Frequent value locality and value-centric data cache design. *Paper presented at: ACM SIGOPS Operating Systems Review*. 2000;48.
83. Ramirez A, Larriba-Pey JL, Valero M. The effect of code reordering on branch prediction. Paper presented at: PACT. IEEE; Philadelphia, PA, USA; 2000.
84. Jiménez DA, Lin C. Neural methods for dynamic branch prediction. *ACM Trans Comp Syst (TOCS)*. 2002;20(4):369-397.
85. Zilles C, Sohi G. Execution-based prediction using speculative slices. Paper presented at: ISCA; Göteborg, Sweden 2001.

How to cite this article: Mittal S. A survey of value prediction techniques for leveraging value locality. *Concurrency Computat: Pract Exper*. 2017;29:e4250. https://doi.org/10.1002/cpe.4250