Copyright

by

Pawan Balakrishna Joshi

2019

The Thesis Committee for Pawan Balakrishna Joshi
Certifies that this is the approved version of the following Thesis:

Techniques for Advancing Value Prediction

APPROVED BY
SUPERVISING COMMITTEE:

Calvin Lin, Supervisor

Mattan Erez

Techniques for Advancing Value Prediction

by

Pawan Balakrishna Joshi

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2019

Dedicated to my beloved parents

Abstract

Techniques for Advancing Value Prediction

Pawan Balakrishna Joshi, M.S.E.

The University of Texas at Austin, 2019

Supervisor: Calvin Lin

Sequential performance is still an issue in computing. While some prediction mechanisms such as branch prediction and prefetching have been widely adopted in modern, general-purpose microprocessors, others such as value prediction have not been accepted due to their high area and misprediction overheads. True data dependences form a major bottleneck in sequential performance and value prediction can be employed to speculatively resolve these dependences. Accurate predictors [1] [2] have been shown to provide performance benefits, albeit requiring large predictor state. We argue that a first step in making value prediction practical is to manage the metadata associated with the predictor effectively. Inspired by irregular prefetchers that store their metadata in off-chip memory, we propose the use of an improved prefetching mechanism for value prediction that not only provides performance benefits but also a means to off-load predictor state to the memory hierarchy. We show an average of 5.3% IPC improvements across a set of Qualcomm-provided traces [3].

The result of a static instruction can be predicted by mapping runtime context information to the value produced by the instruction. To that end, existing value predictors either use branch history contexts [2] or value history contexts [1] to make predictions. As long histories are needed to achieve high accuracy, these approaches slow down the training time of the predictor, negatively impacting coverage. We identify that branch and value histories both provide distinct advantages to a value predictor, and therefore combine them in a novel predictor design called the Relevant Context-based Predictor (RCP) that maintains high accuracy while improving training time. We show an average of 38% speedup over a baseline that performs no value prediction on the Qualcomm-provided traces.

Table of Contents

Abstract ..... **v**

List of Tables ..... **x**

List of Figures ..... **xi**

Chapter 1: Introduction **1**

1.1 The Problem: Data Dependences Limit Sequential Performance 2

1.2 Motivation For The Work 2

1.3 Key Contributions 4

1.4 Organization of the Thesis 5

Chapter 2: Architecture of Modern General-Purpose Microprocessors **6**

2.1 In-Order Pipelining 7

2.2 Memory Hierarchy 9

2.3 Dependence Handling in Pipelining 11

2.3.1 Control Dependences 11

2.3.2 Data Dependences 12

2.4 Out-of-Order Execution 13

Chapter 3: Value Prediction **16**

3.1 Value Prediction For Single-Thread Performance 16

3.2 Value Prediction Mechanisms 18

3.2.1 Computation-based Predictors 19

3.2.1.1 Last Value Predictor 19

3.2.1.2 Stride Predictor 21

3.2.2 Context-based Predictors 21

3.2.2.1 Finite Context Method (FCM) Predictor 22

3.2.2.2 Differential FCM..23

3.2.2.3 DFCM++ 24

3.2.2.4 VTAGE and EVES 24

3.2.3 Store-Load Value Predictors..25

3.3 Prediction Validation and Recovery 26

3.4 Challenges in Value Prediction..27

Chapter 4: Managing Predictor Metadata 29

4.1 Irregular Stream Buffer (ISB) for Value Prediction 30

4.1.1 Irregular Stream Buffer 31

4.1.2 Enhancements to ISB 32

4.2 Evaluation..33

4.3 Results..34

4.4 Discussion:..36

Chapter 5: Handling Divergence 37

5.1 Understanding Control Flow in Programs 37

5.1.1 Divergence Handling using Branch Contexts 39

5.1.2 Divergence Handling using Value Contexts 40

5.2 Combining EVES and DFCM++:..41

5.3 Employing Relevant Context Information: The Relevant Context-based Predictor (RCP)..44

Chapter 6: Conclusion 49

6.1 Limitations and Future Work 49

6.2 Conclusion of the Thesis 50

BIBLIOGRAPHY 52

List of Tables

Table 1: Baseline microarchitecture for simulation 33

Table 2: Comparing our predictor against EVES, DFCM++ 46

List of Figures

Figure 1: Slower training with increasing branch history length 3

Figure 2: Slower training with increasing value history length 4

Figure 3: Single-cycle Microarchitecture 7

Figure 4: Pipelined execution 8

Figure 5: Memory System in a Processor 10

Figure 6: Value prediction to improve performance 16

Figure 7: RAW dependence distances 17

Figure 8: Last value predictor 19

Figure 9: Stride predictor 20

Figure 10: Updating the stride predictor 20

Figure 11: Finite Context Method 22

Figure 12: Structural address space [8] 30

Figure 13: ISB variants, accuracy and coverage 34

Figure 14: ISB variants, speedup 35

Figure 15: Linked list traversal: correlated value stream 38

Figure 16: Graph traversal, local and global divergence 38

Figure 17: Prediction using branch and value contexts 39

Figure 18: Coverage on combining predictors 42

Figure 19: Accuracy on combining predictors 42

Figure 20: Speedup on combining predictors 43

Figure 21: Correct prediction by each component of hybrid 43

Figure 22: Variance of branch contexts 46

Figure 23: Variance of value contexts 47

Figure 24: Variance upon combining contexts 47

Figure 25: Predictability of values for different contexts 48

Figure 26: Speedup over no VP, obtained using {64-bit branch hist, 1 value} context ... 48

Chapter 1: Introduction

Computers are pervasive and vital and in the present day: from mobile devices to connect to the internet to the forecasting complex phenomenon such as the weather, they are used to perform a broad range of tasks. With the amount of data being collected to be processed increasing, there is a sustained need for advances in high-performance computers. Over the last five decades, advances in the performance of processors have come from two key directions. First, as processors are clocked machines, increasing the frequency of the clock allowed the computations to be done faster. However, as the dynamic power consumption of the processor circuit increases with the frequency, today's processors are designed using a maximum clock frequency of 4-5 GHz.

Second, as device technology shrank every few years from micrometer to nanometer sizes, an increasing number of transistors could be crammed onto the same semiconductor substrate [4]. This allowed the designers to implement complex processor designs that make use of *pipelined, superscalar* and *out-of-order* execution with *branch prediction* to improve single core performance. However, as single core performance gave diminishing returns for the number of transistors expended, the paradigm of multicore computing was introduced to increase the performance of a processor by parallelly executing pieces of the program. However, the multicore computing approach has fundamental limitations.

1.1 The Problem: Data Dependences Limit Sequential Performance

Not all programs are amenable to be computed in parallel across multiple cores due to the nature of the algorithm and such programs would see no improvement in performance on using more than one core. Further, even parallelizable programs have a sequential portion of the algorithm. The sequential fraction of the program limits the maximum performance attainable through using multiple cores, as stated by Amdahl's Law [5]. For instance, if the sequential fraction of the program is 20% and the rest can be readily parallelized, a maximum speedup of 5x can be achieved even on using an infinite number of cores. Therefore, sequential performance remains a key bottleneck in improving performance of modern processors, multicore or otherwise.

While modern processors employ techniques such as branch prediction and out-of-order execution to improve sequential performance, they strictly obey data dependences between instructions. Specifically, they stall execution of the dependent chain of instructions until the producer is executed. This is a major bottleneck in modern processors, especially given the slow main memory latency scaling - load instructions that miss the cache hierarchy can experience hundreds of processor cycles of latency. It is therefore imperative that we employ a technique to resolve data dependences in programs.

1.2 Motivation For The Work

The technique of value prediction resolves data dependences in hardware through predicting the result of the producer instruction and speculatively executing the dependent instructions. Several approaches exist to predict the value of an instruction. While EVES [2] uses instruction PC and branch information to make predictions, the FCM-style predictors [1], [6], [7] use PC and PC-local value history information.

Unfortunately, previous work suffers from two problems. First, existing predictors demand a large on-chip storage - they need tens to hundreds of kilobytes to store their metadata [3]. This severely impedes the adoption of value prediction in commercial designs. We argue that the predictor metadata can be stored in a distributed manner in the memory hierarchy, while *caching* only the important predictor metadata in the predictor structure. Second, even for unlimited-sized predictors, existing algorithms do not efficiently learn the values produced by instructions. As programs exhibit control flow divergence, existing predictors resort to either using long branch or long value histories to accurately predict values. While providing better accuracy, the use of long context lengths adversely affects training time and hence coverage, which is the fraction of values predicted among all the eligible values. Figure 1 and Figure 2 show the poor coverage of the predictors that use longer branch [2] and value histories [1] respectively, measured using the Championship Value Prediction (CVP) infrastructure [3]. This poor coverage limits the speedup obtained by the predictor.

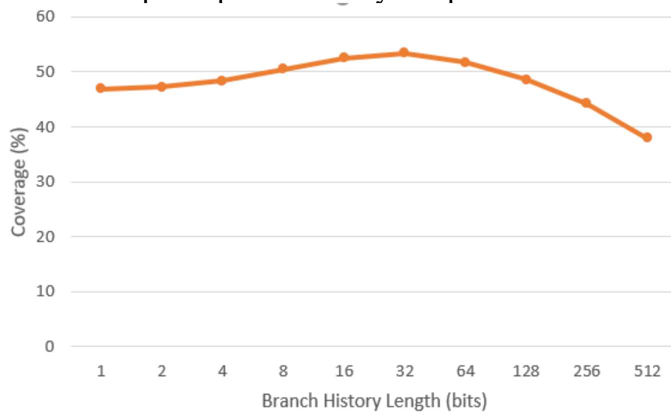

Figure 1: Slower training with increasing branch history length

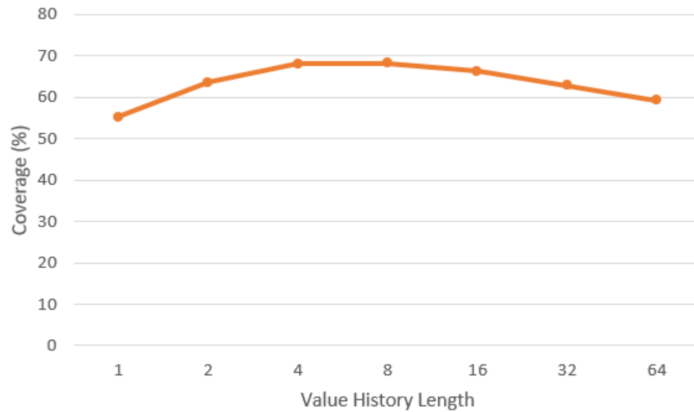

Figure 2: Slower training with increasing value history length

1.3 Key Contributions

In this thesis, we make two key contributions to advance the field of value prediction:

- To manage the high area requirements of a value predictor, we propose the use of an enhanced version of the irregular prefetching algorithm, ISB [8], that is capable of off-loading the predictor metadata to the memory hierarchy.
- We propose the use of *relevant contexts* to handle divergence in programs, obviating the need to use either long branch or long value histories[1]. By clearly identifying the advantages provided by both branch and value histories, we combine them in a novel predictor design called the Relevant Context-based Predictor (RCP) that maintains accuracy while improving training time, and hence coverage.

---

[1] Some of the conclusions on improved divergence handling capabilities using a combination of context information were arrived at independently by Subramanian in their thesis as well. While this work derives them based on identifying the distinct benefits of value and branch histories in handling different types of divergence, their thesis derives its conclusions from an analysis of the gaps in performance between realistic and oracle divergence handling capabilities.

1.4 ORGANIZATION OF THE THESIS

The rest of this thesis is organized as follows: Chapter 2 introduces the architecture of a modern, general-purpose microprocessor; Chapter 3 motivates value prediction and describes previous work in the field; Chapter 4 describes the mechanism to off-load predictor metadata using the prefetcher ISB for value prediction and several enhancements made; Chapter 5 describes the handling of divergence in programs by combining branch and value contexts; Chapter 6 enlists the future directions and concludes the thesis.

Chapter 2: Architecture of Modern General-Purpose Microprocessors

Modern microprocessors are programmable machines that allow the user to provide software instructions to be executed to achieve the desired computation. The function a microprocessor performs is controlled by the binary inputs provided to the digital logic circuits within it. For example, to simply add two integers the arithmetic and logic unit present in the microprocessor needs to be provided the two input operands, a destination to store the result and a unique binary code for indicating that the operation to be performed is an addition. To facilitate programming, the microprocessor provides an interface to the user known as the *Instruction Set Architecture* or ISA. The ISA specifies the set of instructions and the set of registers that can be used by instructions to read operands and write their results. Typically, each instruction has an *opcode*, a set of *source* operands and a set of *destination* operands.

A computer program is a sequence of instructions specified to achieve a desired computation. When executing a program, the microprocessor's state that is visible to the programmer is called *architectural state*, or software-visible state. This includes the values of the registers defined by the ISA, including the *program counter* (PC) which indicates the address of the instruction in memory, and the state of the memory. During execution, the instructions in the program read the architectural state as input, perform computation and write the updated architectural state back, one after the other. This implies that the architectural state is atomically updated by the instructions in program order. This sequential model of execution is termed the *von Neumann architecture*.

The underlying hardware that executes the instructions, called the *microarchitecture*, may have state that is not software-visible. We call this state the *speculative state*.

2.1 IN-ORDER PIPELINING

Program execution on a microprocessor follows the following general flow. The instruction is *fetched* from memory using the address given by the current program counter, and the program counter is then updated to point to the next sequential instruction in the program. This constitutes the *Fetch Stage*. The instruction is then *decoded* into source register *ids*, destination register *ids* and opcodes, and the control signals to drive the functional units are generated. This forms the *Decode Stage*. In the next stage called *Execute*, the processor reads the source operands from the *register file* and uses one or more of its functional units to perform the computation. In case of instructions that access memory, such as load and store instructions, the address computation of the load/store is done in this stage. In the *Memory* stage of the computation memory is accessed by the instruction to read data. Finally, in the *Writeback* stage the instruction commits the result to the register file.

The five stages described above form the basis for instruction execution. As processors are clocked machines, one can simply design a processor that executes one instruction every clock cycle. In this *single-cycle* microarchitecture, the duration of the

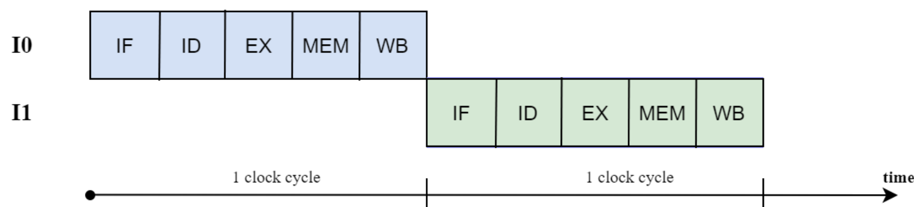

Figure 3: Single-cycle Microarchitecture

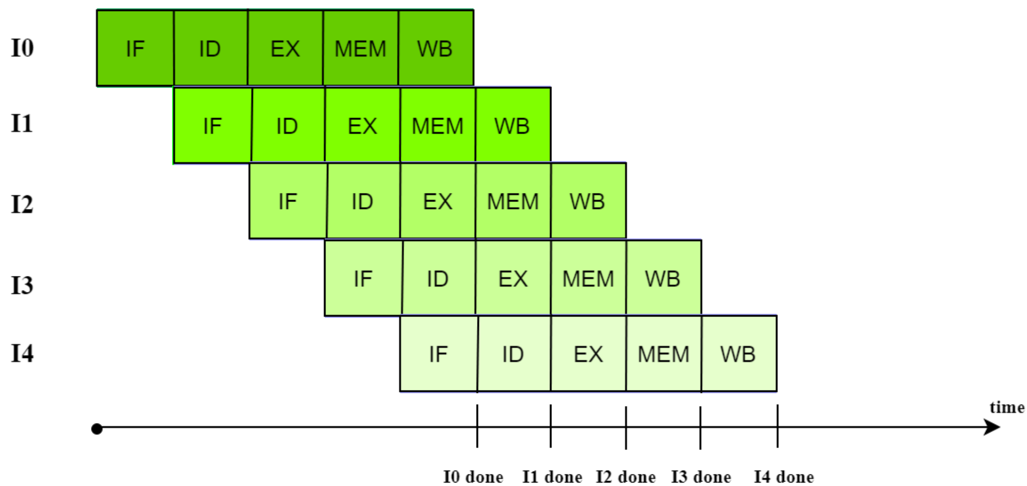

Figure 4: Pipelined execution

clock cycle is specified such that it is long enough to accommodate all the five stages of computation for the slowest instruction, as shown in Figure 3.

The problem with the single-cycle microarchitecture is that at any given moment in time, the logic of only one of the five the different stages of computation is being utilized. This is inefficient as four-fifths of the resources are unused in every stage. To resolve the inefficiency of this design, a *multi-cycle* microarchitecture with pipelining was introduced. In a pipelined microarchitecture, an instruction is executed over multiple cycles with each stage of computation consuming one cycle to complete. Further, once a stage finishes its computation on one instruction, it is free to process the next instruction in the program in the next cycle. This allows for efficient utilization of all stages in every cycle of execution. Figure 4 illustrates this concept, where once the pipeline is full, one instruction is processed every cycle.

Contrary to the single-cycle design, the clock cycle need not be as long as the slowest instruction but instead needs to be only as long as the slowest stage of the pipeline. This allows for higher frequency clocks and better performance. For example,

assuming each stage of computation takes one cycle to execute, the single-cycle microarchitecture would finish one instruction in every 5 clock cycles. However, the pipelined design would finish one instruction every cycle once the pipeline is full.

It is to be noted that the clock duration in the pipelined architecture can be made shorter by increasing the number of stages in the pipeline. However, the dynamic power consumption of the microprocessor increases with the clock frequency. Due to this reason, current microprocessors have clocks that are at 4-5 GHz, and it is impractical to increase the frequency of the clock further.

2.2 MEMORY HIERARCHY

Load and store instructions access the memory for reading or writing data. As memory accesses on average take longer than computation, in an in-order pipeline as described in the previous section, the memory stage is typically the slowest stage and determines the clock cycle. Therefore, to achieve high performance, memory accesses need to be low-latency operations.

A slow memory technology such as DRAM is cheap and dense, while fast memory such as SRAM consumes more chip area and is expensive. To achieve a good balance between storage capacity and access latency in a modern processor, the memory

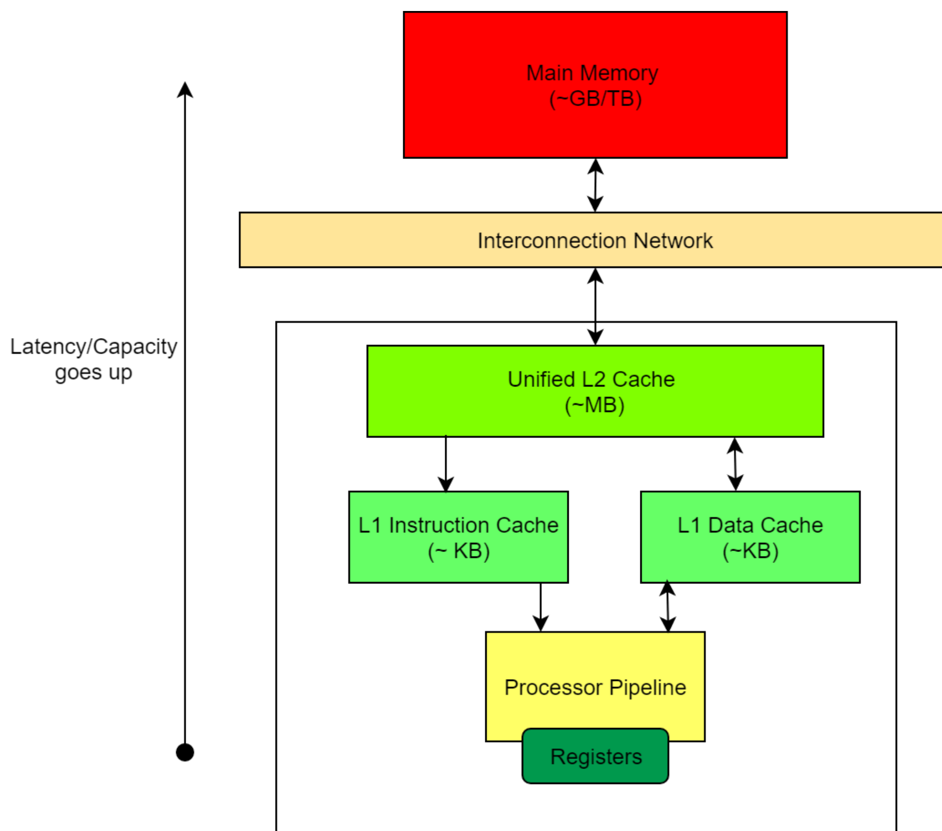

Figure 5: Memory System in a Processor

subsystem consists of a hierarchy of storage in which the faster, higher levels store a subset of the slower, global memory, as depicted in Figure 5.

In a memory hierarchy, typically the lower level of memory is the main memory or DRAM, with the higher levels or *caches* being SRAM. Apart from being built with slower memory technology (DRAM), the main memory is often located physically *off-chip* from the processor. It is thus useful to note that while an access to the smallest level of the cache hierarchy takes a few processor cycles, an access to the main memory may take several hundred cycles, stalling the processor in the process. When a load/store instruction requests to read/write an address in memory, data is brought into the fast storage if absent. This is termed a *cache miss*. As the cache has low capacity, the old

data is evicted to the lower level storage once fresh data is brought in. An algorithm called the *cache replacement policy* decides which data to evict from the caches. For example, the *Least Recently Used* (LRU) policy evicts the cache line which has been used the furthest in the past by the processor.

Programs access large amounts of memory relative to the size of the caches. For high performance, it is necessary that in the average case, most of the memory accesses are made to the higher-level caches rather than to the slow main memory. The primary reason caches provide performance benefit despite having very low storage capacity is that typical programs exhibit a property known as *locality*. *Temporal* locality is seen when programs access the same memory address repeatedly over time, allowing the cache to serve the memory request after the data is loaded from the main memory. Another form of locality exhibited is *spatial* locality, in that programs access data that is *close by* the data currently accessed. In conclusion, the exploitation of locality by caches reduces some of the impact of the slow nature of memory accesses on performance.

2.3 Dependence Handling in Pipelining

2.3.1 Control Dependences

Branch instructions can modify the flow of the program such that the next PC is not the same as the instruction immediately after the current PC. Branch addresses are typically determined only in the Execute stage of the pipeline. This presents a problem for the pipelined microarchitecture as the Fetch stage needs to access the memory with the address of the next instruction (next PC), which may not be ready as this address is yet to be generated by the Execute stage. This scenario where the address of the next instruction is unknown at Fetch is called a *control dependence.* As a result, one may naturally expect to stall the Fetch stage by the *fetch-to-execute* delay for branch

instructions, losing performance. In deeply pipelined modern processors, this delay may be as long as 10-15 cycles and control dependences become a key performance bottleneck.

To resolve control dependences, modern processors use branch predictors. These predictors learn the direction and target of branch instructions dynamically and predict the address of the next instruction. The Fetch stage can then proceed even while the branch executes. However, as the prediction is not guaranteed to be correct, the pipeline must be reset upon detecting a mismatch between the predicted instruction address and the result of the branch instruction. Modern branch predictors typically achieve high accuracies in the range of 90-99% [9], [10].

2.3.2 Data Dependences

An instruction in a program may be dependent on the output of a previous instruction. For example, consider the instruction sequence below, with R* representing the register id.

I1: R1 = R0 + R2

I2: R3 = R1 * R2

I3: R1 = R4 – R0

As the value of register R1 produced by instruction I1 is required by I2, the instruction I2 is said to have a *true data dependence* or a *Read-After-Write* (RAW) dependence on instruction I1. These dependences convey the semantics of any sequential program, and hence are to be obeyed. In an in-order pipeline, such dependences cause the dependent instruction and all subsequent instructions to stall until the producer instruction completes execution.

Further, as instruction I3 writes the value of R1 after the instruction I2 has read it, I3 is said to have a *Write-After-Read* (WAR) or an *anti-dependence* on I2. If I3 were to write R1 before I2 read it, the program would execute incorrectly. Similarly, the instruction I3 is said to have a *Write-After-Write (WAW)* dependence on I1, as they write to the same register R1. If instruction I3 were to write to R1 before I1, the result of I3 would be lost.

It is to be noted that WAR and WAW dependences only exist because of insufficient architectural registers. For instance, if the instruction I3 could write its result into another register id, there would be no WAR dependence with I2. As a result, these dependences are often called *fake* dependences. As an in-order pipeline only executes instructions in program order, fake data dependences are not an issue.

2.4 OUT-OF-ORDER EXECUTION

In the previous sections, an in-order pipeline was described as an efficient way to parallelize instruction processing while still maintaining sequential program semantics. Even though WAR and WAW data dependences are not an issue, true data dependences cause the pipeline to stall. This is particularly a bottleneck when the dependent instruction is waiting for the result of a load instruction that misses all the levels of the cache hierarchy, potentially stalling the pipeline for hundreds of processor cycles. The problem with the in-order pipeline is that it stalls even though there may be several instructions in the program *downstream* that are independent of the stalled instruction. Such independent instruction streams should ideally be executed while the stalled instruction waits for its operands. The presence of such independent streams of instructions in a program is called *Instruction Level Parallelism* or ILP.

To exploit ILP and reduce the impact of RAW dependences on performance, instructions can be executed out of program order. Such a processor is called an *Out-of-Order* processor. During program execution, the processor uncovers instructions that are independent of one another and executes them in parallel. This approach results in the processor not stalling entirely when a RAW dependence is observed. To achieve this, the processor fills a window of instructions in a hardware buffer called the *Instruction Queue* (IQ), and schedules the instructions whose operands are ready, while those instructions stalled by RAW dependences wait in the IQ. This results in instructions executing in parallel and out of program order, while still maintaining the true data dependences between dependent instructions. However, to maintain sequential program semantics, the processor still updates the architectural state in program order. It does so using a structure called the reorder buffer (ROB).

In summary, an Out-of-Order processor pipeline hides the latency caused by RAW dependences by exploiting ILP. It uses an in-order front-end that fetches instructions in program order and feeds them into the out-of-order execution unit. The instructions then update the architectural state in an in-order manner.

While out-of-order execution hides the latency of true data dependences through executing independent instructions in the instruction stream, it strictly enforces RAW dependences and stalls the dependent instructions in the pipeline until their producers finish execution. Hence, the single-thread performance of an out-of-order processor is limited by true data dependences. This work revisits the technique of *Value Prediction* as a way to improve performance by breaking the RAW dependences. At its core, value prediction aims to predict the result of the producer instruction in a RAW dependence so that the consumer can be speculatively executed. Upon a correct prediction the

dependence is broken, allowing the dependent instruction to execute concurrently with the producer, and higher sequential performance is achieved.

Chapter 3: Value Prediction

3.1 VALUE PREDICTION FOR SINGLE-THREAD PERFORMANCE

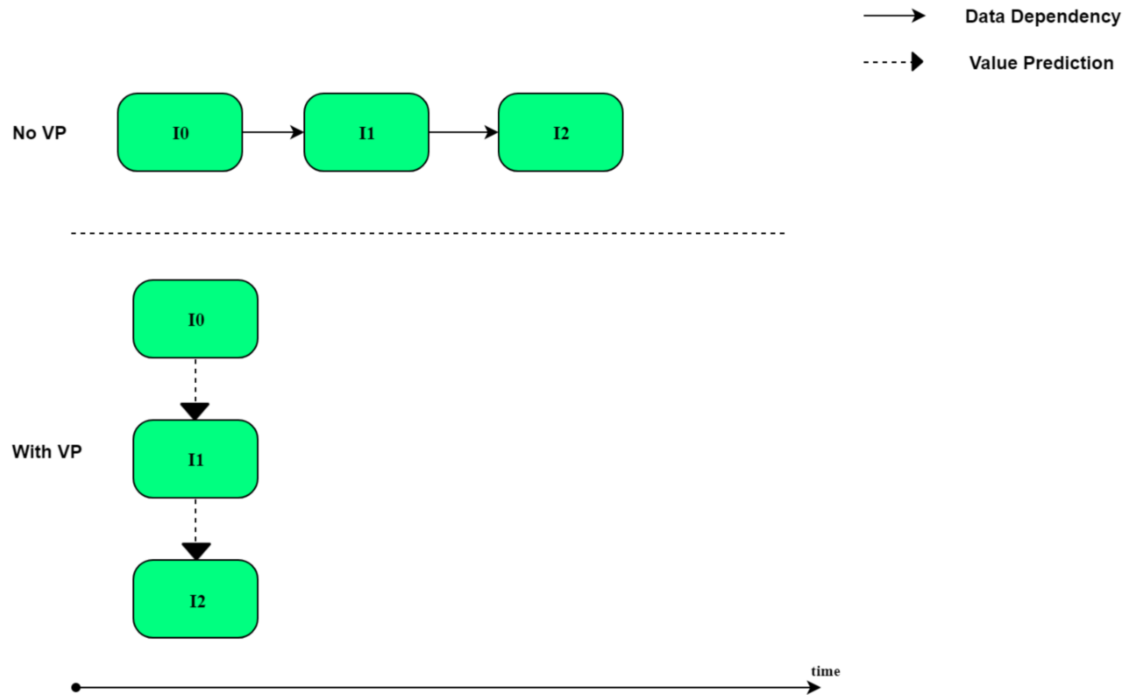

Figure 6: Value prediction to improve performance

As introduced in the previous chapter, value prediction is a speculative execution technique to resolve RAW dependences. For example, consider the sequential execution of three dependent instructions as illustrated in Figure 6. Without value prediction, each instruction would have to wait for its producers result. However, if we can accurately predict the result of an instruction, say I0, then instruction I1 can proceed, thus improving the ILP of the program and hence performance.

Predicting values only improves performance if the producer instruction's result is not ready when the dependent instruction arrives. For that to occur, the average

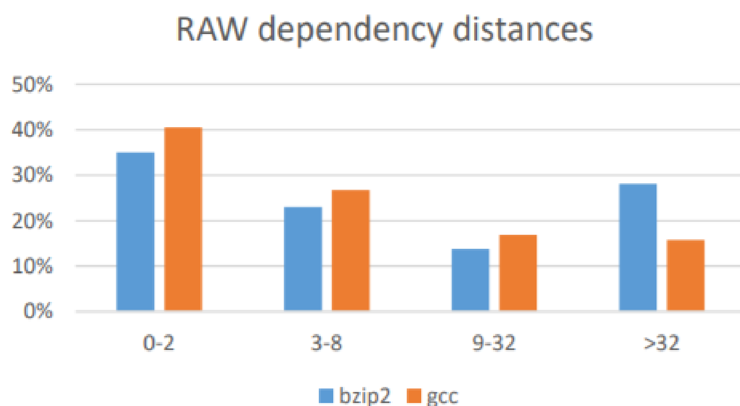

Figure 7: RAW dependence distances

RAW dependence distance between the producer and the consumer needs to be smaller than the pipeline depth from fetch to execute; otherwise the producer's result can be forwarded to the dependent instruction. We motivate the use of value prediction by measuring the RAW dependence distances in two SPEC2006 [11] benchmarks, *bzip2* and *gcc*, using an Intel PIN [12] simulator. As is evident from Figure 7, nearly 70% of dependent instructions occur within 8 instructions of the producer, which is likely before the producer has executed.

Further, evaluation of perfect value prediction on the Championship Value Prediction (CVP) framework [3] using a set of 130 Qualcomm-provided traces provides an average speedup of 245% over a baseline that performs no value prediction. Although absolute numbers are dependent on the simulation environment, this result motivates the use of value prediction for improving single-threaded performance.

Just as the technique of caching data in a processor benefits from the spatial and temporal locality of memory addresses, the predictability of data values is made possible due to the *locality of values*. This observation was first described by Lipasti et al. [13] and Gabbay and Mendelson [14]. The key insight that enables value prediction is that

even though data in registers can span a large space of 2^{64} values, dynamic instructions tend to produce values that remain constant, exhibit regular strided patterns, or exhibit repeated irregular patterns.

For example:

Constants: 100, 100, 100, 100, 100 ...

Strided: 2, 4, 6, 8, 10 ...

Correlated: 17, 41, 8, 140, 17, 41, 8, 140 ...

To predict these patterns, Sazeides and Smith [6] define two types of value predictors: *computation-based* predictors and *context-based* predictors, which we describe in the next section.

3.2 VALUE PREDICTION MECHANISMS

Several algorithms have been proposed to predict the value produced by an instruction. The efficacy of a value predictor is measured using three metrics:

1. **Accuracy** – the ratio of the number of correct predictions made to the total number of predictions made.
2. **Coverage** – the ratio of the number of predictions made to the total number of values eligible for prediction in a given program.
3. **Speedup** – the percentage increase in IPC of the processor over a baseline that performs no value prediction. Speedup is a function of the accuracy and the coverage achieved by the predictor, and the class of instructions predicted.

3.2.1 Computation-based Predictors

A computation-based value predictor applies a computation or a function to the result of the previous instance(s) of the instruction to generate the prediction for

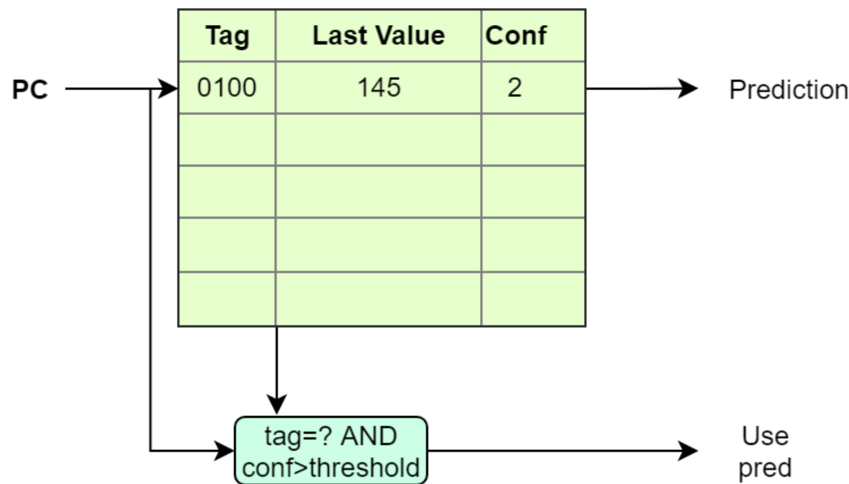

Figure 8: Last value predictor

the current instance. The last value predictor [13], the stride predictor [14] and the value estimator [1] are examples of computation-based predictors.

3.2.1.1 Last Value Predictor

As proposed by Lipasti et al. [13], a last value predictor predicts the value of the current instance of an instruction to be the same as the value produced by the previous instance. This predictor applies the identity function on the previously observed value and uses it as the next prediction. This prediction strategy, albeit very simple, is useful when the instruction produces constant values. For example, a significant portion of instructions in a program tend to repeatedly access variables and memory locations that are not modified once set, and hence are runtime constants.

The predictor is depicted in Figure 8. It is a structure that contains the lower bits of the instruction PC as a tag, along with the last seen value. Typically, the predictor entry also consists of a saturating counter used as a confidence mechanism. The predictor is accessed using the lower bits of the instruction PC as index, and if the tag matches, the

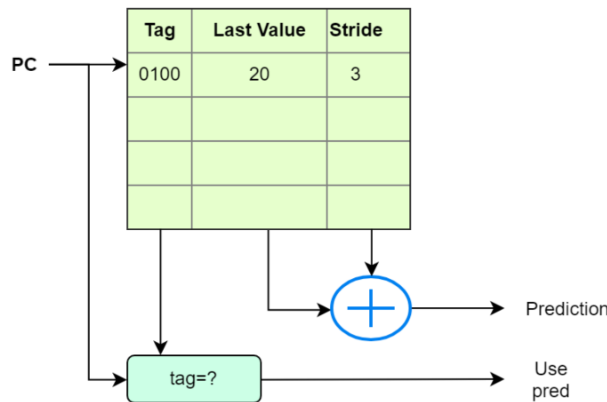

Figure 9: Stride predictor

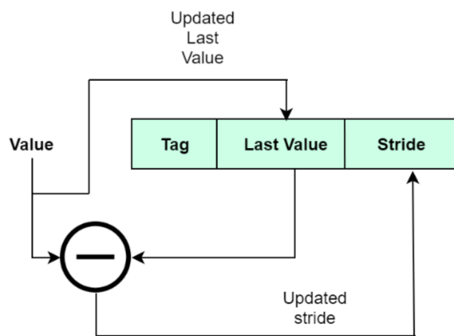

Figure 10: Updating the stride predictor

stored value is used as a prediction, given it has high enough confidence. If the tag does not match or the confidence is not high enough, no prediction is made. When an instruction

retires, it updates its result into the table at the appropriate location based on its PC. The counter is incremented on a correct prediction and reset otherwise.

3.2.1.2 Stride Predictor

Apart from constants, instructions in programs also tend to produce values that exhibit a regular strided pattern. For instance, when traversing an array, the addresses accessed by a program are separated by a constant stride. Similarly, the loop index variable of a *for-loop* typically exhibits a regular strided pattern. To predict strided patterns, Gabbay and Mendelson [14] introduced a stride predictor, as shown in Figure 9.

A stride is computed by the difference in the values between consecutive instances of an instruction. To make a prediction for the current instance, the result of the previous instance is added to the stride. To update the predictor, when the instruction retires, its value is stored as the *last value*, while the difference in the value of the current instance and the previous instance is stored as the new stride. The update mechanism is depicted in Figure 10.

In contrast to the last value predictor described in the previous section, the stride predictor can predict values that it has never seen before by simply computing them through addition.

3.2.2 Context-based Predictors

Contrarily to computation-based predictors that compute new values based on previous ones, context-based predictors learn the values produced by a certain program context in the past and apply it in the future when the same context is observed. Program contexts can include values produced by the same instruction, value produced by other instructions, global branch outcome history, etc. These form a key class of predictors as not all instructions tend to produce values that are constant or exhibit strided patterns. For instance, programs commonly traverse linked lists and graphs, and the values produced by the pointer accesses in such programs tend to be irregular and not amenable

to computation-based predictors. However, if the programs traverse the same data structures repeatedly, a context-based predictor would be able to learn the values produced on each instance of some context and apply it when the same context repeats. We describe several context-based predictors in the sections below.

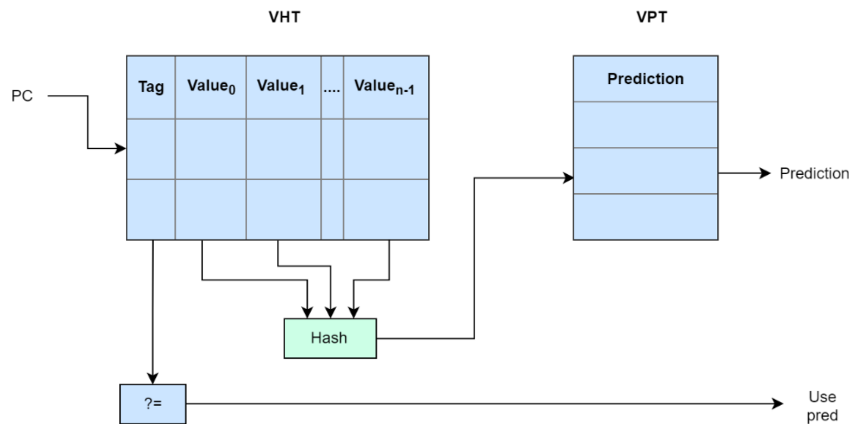

Figure 11: Finite Context Method

3.2.2.1 Finite Context Method (FCM) Predictor

Introduced by Sazeides and Smith [6], FCM predictors use a two-level strategy to predict values. The first-level table, called the *Value History Table* (VHT) stores the last n-values produced by a static instruction. It is indexed by the instruction PC and is tagged. The second-level table, called the *Value Prediction Table* (VPT) is indexed by a hash of the last-n values stored in the VHT. This stores the actual value to be used for prediction and a confidence counter mechanism. As described by Sazeides and Smith, an *order-n* FCM tracks the last-n values in the VHT and is depicted in Figure 11. The update mechanism involves updating the value history in the VHT and the actual value produced by the instruction in the VPT.

While being versatile to predict constants and strided value patterns, the FCM predictor is particularly useful at predicting correlated value patterns. For example,

consider traversing a four-node linked list repeatedly. For accessing the value of a node in a linked list, a pointer access is involved. It initiates a load of the pointer to the node which generates the node address. This is followed by a load of the address which generates the value of the node. The node addresses would show a repeating correlated pattern A_1-A_2-A_3-A_4 in each iteration, while the node values would present the sequence V_1-V_2-V_3-V_4. As the addresses and values can take arbitrary values, a stride predictor would not be able to predict the address and value sequences. However, using a history of the value produced by the load in an order-1 FCM, one can predict the next value. In this case, learning that A_3 is followed by A_2 is sufficient to predict A_3 every time we observe A_2.

3.2.2.2 Differential FCM

Proposed by Goeman et al. [7] differential FCM is a modification on the original FCM predictor that tracks the differences in the local values of an instruction rather than the values themselves. The VPT is indexed using the hash of these deltas, and a delta is predicted to be used. The final value prediction is simply a sum of the predicted delta and the last value of the instruction stored in a separate table. This makes the predictor a lot more space efficient as constant patterns which used to take up several distinct VPT entries in the original FCM design now would take up exactly one entry, as all constant patterns have a delta history of zero. Further, DFCM trains faster than the FCM predictor in case of constant patterns. This is because the predicted delta of zero holds true for numerous PCs that exhibit constant values, and it is sufficient to learn the zero-delta for one PC and apply the same to other PCs.

The update mechanism is similar to the FCM predictor, but deltas are computed upon instruction retirement and appropriately stored in the VHT and VPT, instead of values.

3.2.2.3 DFCM++

For prediction mechanisms such as FCM and DFCM that use a PC-local history of values, it is imperative to make predictions using a value history that is not stale. In programs that have tight for-loops, it may often occur that the previous instance of an instruction may be in-flight when the prediction for the current instance is required. This would mean that the Value History Table would not be updated with the correct value history and using a stale history would likely cause a misprediction. Unless a specialized mechanism is used to address this issue, the predictor would simply have to give up predicting instructions that have several in-flight instances.

Deshmukh et al. [1] propose the use of speculative delta histories that are updated using the predictions made by the DFCM predictor. In the first-level table of the DFCM predictor they maintain a *commit-time* history which is always correct, but also augment it with a *predict-time* history which is updated speculatively. During prediction, the predict-time delta history is used. Upon instruction retirement, the commit time history is updated with the correct deltas.

3.2.2.4 VTAGE and EVES

Perais and Seznec [15] address the problem of predicting inflight instructions differently than DFCM++ in that they do not use local value histories in their predictor. They use global branch history information along with the program counter of an instruction to index into a prediction table, and hence convert the problem of predicting

correlated values into constant values per branch context. This allows them to not store speculative histories and yet make predictions on constant, strided and correlated patterns of values. Intuitively, the predictor would require branch history lengths proportional to the length of the correlated value streams to remove aliasing. This is described in detail in Section 4.1.1.

While long branch histories provide the capability of predicting long streams of correlated values, invariably using long histories even for constant PCs and short value streams would slow down the training time of the predictor. The length of the branch history used hence presents a tradeoff between accuracy and training time of the predictor. To exploit this tradeoff, VTAGE uses an array of branch history lengths in a geometric progression. This allows VTAGE to use the appropriate branch history length to predict different classes of instructions.

Seznec further enhances VTAGE by making it more space efficient and augments it with a stride predictor in a hybrid predictor design called EVES [2].

3.2.3 Store-Load Value Predictors

As discussed in the previous section, loads form an important class of instructions for value prediction due to their potentially higher latency than arithmetic instructions. An important problem in load-value prediction is that of conflicting stores. For example, if an instruction sequence looks like the following:

Load X – Store X – Load X

This exhibits with two dynamic instances of the same load with an interleaving store to the same memory location *X*, then the store would modify the learnt value of the load. However, as the store modifies a memory location in the data cache, it is possible to predict the value of the subsequent load using the value in the data cache. Sheikh et al

[16] propose the use of address prediction of the subsequent load to query the data cache. If the data is found, they then use the value to predict the second load.

3.3 Prediction Validation and Recovery

With several algorithms described to predict the value of an instruction, we now discuss some microarchitectural design tradeoffs in incorporating value prediction in a microprocessor.

Validation and misprediction recovery mechanisms are key design choices to be made in implementing value prediction. Validation can be done either at execution time, when the result of the operation is ready, or at commit time, when the instruction that produced the value becomes the oldest instruction in the reorder buffer (ROB). The trade-offs associated with this choice are clear: commit time validation necessarily has a higher misprediction penalty as there can be a multi-cycle latency between the value being ready at execute and the instruction reaching the top of the ROB. Hence, from a purely performance standpoint validation at execution is the more attractive option. However, validation at execution requires checkpointing the architectural state, additional ports to the physical register file and comparators at the output of functional units to compare the result with the predicted value, and hence has much higher hardware complexity.

Upon validation, a misprediction recovery mechanism is to be initiated for every incorrect prediction that was consumed. There are two alternatives for recovery: pipeline squashing and selective re-issue. The former method involves flushing the pipeline state and re-executing from the instruction which consumed the incorrect value. However, it is not necessary to squash instructions that are independent of the producer instruction. Therefore, alternatively a selective re-issue mechanism can search through the instruction queue for dependent instructions which consumed the incorrect value, and selectively re-

issues them. Undoubtedly, the latter method is immensely complex in hardware. Despite that, large performance improvements over pipeline squashing are not guaranteed as the re-issue is on the critical path of the recovery mechanism. Hence, the two methods can be expected to perform comparably in the average case.

3.4 Challenges in Value Prediction

As discussed in Chapter 1, control and data dependences impact the performance of a pipelined processor. While branch prediction has been widely adopted by modern microprocessors to resolve control dependences, there is no known implementation of value prediction in any commercial processor. This is because value prediction presents several significant challenges that either do not exist for conventional branch prediction or are less drastic. Firstly, instead of a binary taken or not taken decision, data can span a much larger range of values, making the state required for accurate predictions prohibitive. As the value predictor is typically stored in the processor core, the amount of predictor metadata that can be allocated on the chip is very limited (few kilobytes).

Secondly, virtually every instruction depends on the result of some preceding instruction. This makes misprediction detection techniques such as validation at execute implausible due to the amount of checkpoint state required. Value prediction must instead rely on validation at commit, incurring a drastically higher misprediction penalty. As a result, there is a large asymmetry between the small average benefit of correctly predicting a value and the large misprediction penalty, requiring that value predictors be very accurate, typically over 99%.

By observing computation and context-based predictors, two key points emerge:

1. Although context-based predictors can predict the values of a wider class of instructions, such predictors are more complex than simple predictors such as last

value predictors or stride predictors as they must store previously observed contexts. Hence, they inevitably incur a larger area overhead. Typically, the longer the context to be stored, the larger is the area overhead.

2. The quality of the context is crucial for the predictor to learn the patterns and apply it. While longer contexts intuitively can make more accurate predictions, they slow down the training of the predictor as the predictor observes the same context less frequently to learn from it. This negatively affects the coverage of the predictor.

In the following Chapters, we address some of these issues of managing metadata and learning patterns accurately without sacrificing training speed.

Chapter 4: Managing Predictor Metadata

As described in the previous chapter, solutions that use the Finite Context Method (FCM) [1], [6], [7], and EVES [2] have been proposed to learn and predict streams of values that exhibit constant, strided and correlated patterns. However, the metadata storage overhead required by today's value predictors is a major challenge in their implementation. As shown in prior work [7], FCM-based methods in particular require hundreds of kilobytes of storage to be competitive. Inspired by research in prefetchers [17], [8] that store their prediction metadata in off-chip memory, we evaluate an irregular prefetcher called the Irregular Stream Buffer (ISB) [8], that is capable of learning correlated streams of arbitrary length in the context of value prediction. We qualitatively argue that ISB's management of metadata is a key insight that future value predictors should employ to achieve a practical implementation.

Previous work [17], [8] in the irregular prefetching community has demonstrated success in storing the metadata in the off-chip memory as a way to balance the amount of on-chip state required and the impact on memory performance. Similar to the idea used by prefetchers, we propose that one way of managing the metadata of PC-indexed value predictors is by tracking the I-TLB misses and evicting the cached metadata for the PCs whose pages are not TLB-resident. However, current value predictors store their values in large monolithic tables that are ill-suited to be off-loaded to the memory hierarchy. For example, in an FCM-based predictor, it is only possible to evict entries that correspond to a particular PC from the level-1 value history table. Evicting level-2 prediction table entries that correspond to a complete stream of values is impossible, as these values are scattered throughout the level-2 table due to the hash of the value history, and no information about their location is stored.

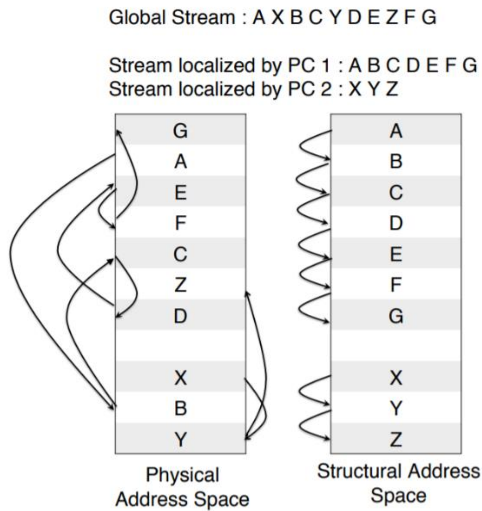

Figure 12: Structural address space [8]

On the contrary, the Irregular Stream Buffer (ISB) prefetcher intelligently stores its metadata such that large contiguous streams of values per PC are in consecutive locations in the predictor structure, and hence can be efficiently offloaded to the off-chip memory. Therefore, we evaluate the ISB prefetcher in the context of value prediction and discuss mechanisms for reducing the on-chip area overhead of future value predictors. This solution allows for a larger effective distributed predictor structure, potentially improving value prediction performance. While it is evident that such a solution trades off increased pressure on the memory hierarchy for the benefit of value prediction, it is to be noted that no prior work in value prediction allows for making this tradeoff, completely leaving a design space unexplored.

4.1 Irregular Stream Buffer (ISB) for Value Prediction

This section describes the Irregular Stream Buffer (ISB) [8] by first briefly summarizing the concept of structural address space as illustrated in Figure 12, and the prefetcher design. We then describe the several enhancements made to adapt the

prefetcher to value prediction. We leave out the detailed description of the address mapping mechanisms used by ISB for brevity.

4.1.1 Irregular Stream Buffer

Unlike prior methods where the correlated streams of values associated with a particular PC can be scattered throughout the predictor structure, ISB uses a layer of indirection to map a stream of correlated values to a special address space known as the structural address space, where consecutive entries are at consecutive locations of the address space. Figure 12 shows that the stream X, Y, Z due to PC2, which is initially in the physical address space, is stored such that the complete stream is mapped to consecutive locations in the structural address space. This enables ISB to efficiently fetch streams of values associated with a PC from the memory hierarchy and evict complete streams out to memory when needed.

The training strategy employed by ISB is similar to a first-order FCM: it determines pairs of values per PC that have high correlation, say A and B, and maps the two values to consecutive structural address locations, say S and S+1, in the predictor data structure. It maintains a confidence counter for the entry S+1, which is incremented if B is seen to follow A repeatedly, and decremented if a different value C is seen after A. While making a prediction for a trigger value A, the structural mapping S for A is determined, which naturally provides S+1 as the next value in the stream. Then, a reverse mapping from structure address S+1 to physical address B is used upon determining which the value B is predicted. The reader should note that any arbitrarily long stream of values can be mapped to consecutive locations in the structural address space. B.

4.1.2 Enhancements to ISB

Evaluating vanilla-ISB in the context of value prediction by training ISB on the stream of values produced made it evident that the implementation of ISB as described by Jain and Lin [8] exhibited aliasing between PCs. This was due to the common data structure used for storing all structural addresses, which caused severe destructive interference between different program counters. To resolve this issue, we implemented a stricter PC-localized version of ISB named local-ISB which completely eliminated any such interference by separating per-PC streams in the structural address space. Though destructive interference should be limited, constructive interference from different PCs should ideally be allowed to exist as it helps the predictor train faster. In the next enhancements attempted, we only separated PCs into different spaces in the predictor structure if they happened to show destructive aliasing. We name this version lazy-ISB, as it lazily separates PCs that interfere. Intuitively, this method is expected to tradeoff some accuracy to provide superior coverage.

The problem of value prediction is considerably more challenging than prefetching due to several reasons. Most importantly, the penalties of a misprediction in a prefetching environment, cache pollution and bandwidth overhead, typically impact the performance far less than a recovery from a value misprediction. Thus, prefetching can afford to be less conservative than value prediction. To improve the accuracy of ISB in the context of value prediction, we employed strict confidence thresholds for predicting correlated pairs of values (127 consecutive appearances) and drastically reduced confidence upon a misprediction by dividing the confidence by 4. Finally, we observe that the main drawback of value context-based predictors such as ISB [8] and FCM [6] as compared to solutions such as VTAGE [15] is that the prediction depends on the value history being up to date. For instance, to make a prediction in a stream of values A, B,

C..., the value B must be updated in the predictor to make the prediction C. To extract higher performance from our predictor, we speculatively update the value history with the predicted value before the instruction retires if we are confident of the prediction, enabling us to make back-to-back predictions.

4.2 EVALUATION

We evaluate our ideas in an out-of-order pipeline using the Championship Value Prediction (CVP) [3] infrastructure, which is a simulation infrastructure provided by Qualcomm as part of a year-round value prediction competition. Qualcomm has provided 135 traces, each with 30M dynamic instructions, to be used for evaluating a value predictor submission. The traces are separated into Integer, Server and Floating Point (FP) traces. Table 1 describes the baseline microarchitectural details used for our experiments. We also implement an Intel PIN [12] -based simulator for testing our predictor implementation and use microbenchmarks alongside SPEC2006 benchmarks.

Table 1: Baseline microarchitecture for simulation

Instruction Window Size	256
Fetch Width	16
Branch Prediction	2-level predictor (2-bit PHT entry, 16-bit global history)
Memory Disambiguation	Perfect
L1 cache	32 KB, 4-way, 64B, 2c
L2 cache	1 MB, 8-way, 64B, 12c
L3 cache	8 MB, 16-way, 64B, 60c
Main memory	150c fixed latency

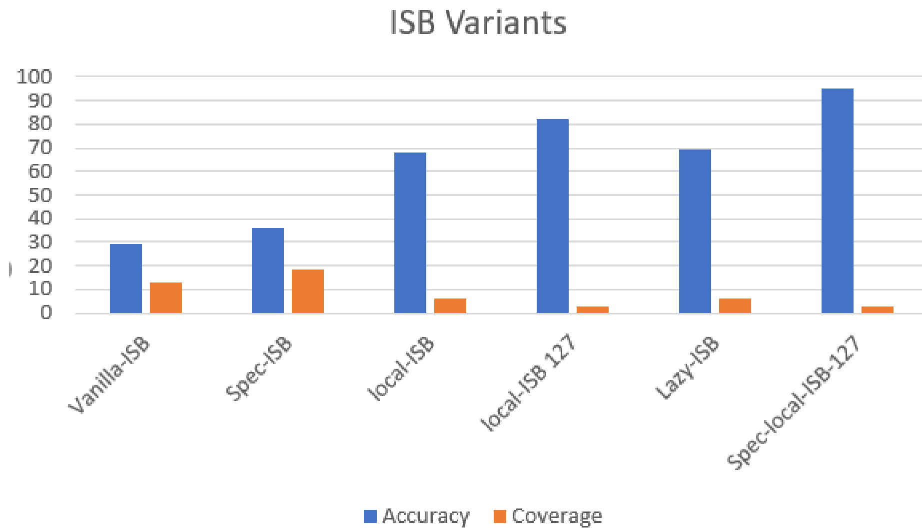

Figure 13: ISB variants, accuracy and coverage

4.3 Results

We implement the vanilla-ISB prefetcher for value prediction in the CVP [3] simulator and an Intel PIN-based simulator, and thoroughly test the implementation against several microbenchmarks that involve array, linked list and tree traversals. We observe that while ISB performs well on the array and linked list microbenchmarks, it does not handle diverging value streams that occur in tree traversal well. This is expected as diverging streams effectively invalidate the previous correlations learnt by ISB.

On the Qualcomm traces, we observe that the accuracy of the predictor, shown in Figure 13, is not sufficient for obtaining any performance benefits. In fact, we see a 55% slowdown as compared to the baseline due to the high number of mispredictions. We identify that most mispredictions can be attributed to destructive aliasing between PCs. As a result, our local-ISB enhancement more than doubles the accuracy of the predictor. To further improve the accuracy, we incorporate confidence mechanisms that require pairs of values to appear for several times greater than a confidence threshold to be

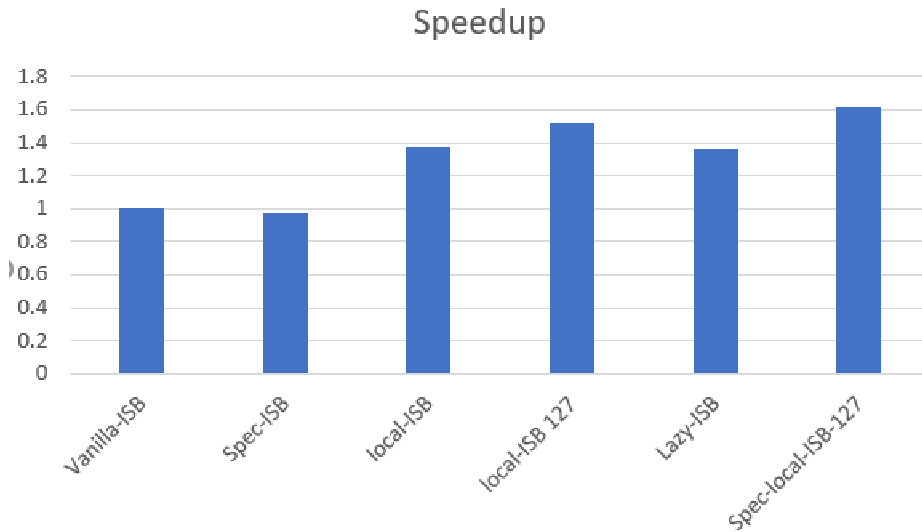

Figure 14: ISB variants, speedup

considered for prediction. Through sensitivity studies, we find that a threshold of 127 is optimal for our implementation.

The lazy-ISB variant which was designed to selectively remove destructive aliasing and maintain constructive aliasing, provides improved coverage. However, the accuracy is reduced. As a result, the lazy-ISB variant does not see any improvement in speedup. We then evaluate the speculative update enhancement in the local-ISB variant and observe that the accuracy of the predictor improves to 95.32%, while providing an average speedup of 5.32% across benchmarks, which indicates that early update solves a key problem associated with context-based predictors. This variant of local-ISB, with speculative update and confidence thresholds improves the performance of vanilla-ISB by more than 60% as shown in Figure 14 and improves the performance of the system by 5.3% over the baseline of no value prediction.

Finally, we conduct an oracle value prediction study for the final local-ISB variant and observe 15.57% speedup by predicting all the values that would have been

predicted by local-ISB correctly, indicating that the performance achieved is still limited by accuracy. On our PIN-based simulator we analyze the mispredictions made by local-ISB on SPEC 2006 benchmarks, and as expected, we observe that divergence in the stream of values due to branch instructions causes most of the remaining mispredictions. We discuss potential future directions for improving accuracy in the following section.

4.4 DISCUSSION:

As is evident from the results of our experiments on using ISB for value prediction, the accuracy requirements for value prediction are extremely high, and our current version of ISB (Speculative local-ISB with confidence mechanisms) falls short due to poor handling of divergence of value streams. We observe that divergence is dependent on the control flow of the program and can be predicted by taking the branch direction history into account. We would expect the accuracy to improve by identifying and storing divergent values in streams and predicting them based on control flow. While FCM handles divergence by using a long value history, it incurs a severe area overhead and loses the ability to offload the metadata. This motivates us to design a predictor that uses branch history for handling divergence, but a short value history for tackling correlated value streams while keeping area overheads manageable.

Chapter 5: Handling Divergence

5.1 Understanding Control Flow in Programs

Branch instructions change the flow of control in programs. As discussed in the previous chapter, control flow affects the predictability of data values and hence should be an important consideration when designing a value predictor. Using simple examples, we discuss how branch instructions affect the value produced by a given static instruction. We then qualitatively analyze how existing predictors counter the effects of control flow *divergence,* where in a learnt pattern of values is disturbed through the dynamic changes in branch direction.

Consider the example shown in Figure 15 below. It depicts a traversal of a linked list, which is a common occurrence in programs. The *n* values produced by the instruction at program counter *pc1* are illustrated as A_0, A_1 … A_{n-1}. As the value produced by instruction *pc1* potentially changes in every iteration of the loop, we denote this form of value variability as *local* divergence.

Now consider a more complex example as depicted in Figure 16. We observe that this scenario commonly occurs during traversal of tree and graph data structures, where the value of a static instruction may diverge from the observed correlated stream A_0, A_1, … A_{n-1} depending on the path of the graph chosen. As the values generated by the instruction depends on not just the "loop iteration" but also on the control flow path chosen to reach a particular node, we denote this form of value variability *global* divergence. After abstractly defining the two forms of divergence observed in programs, we now discuss how existing value predictors handle them.

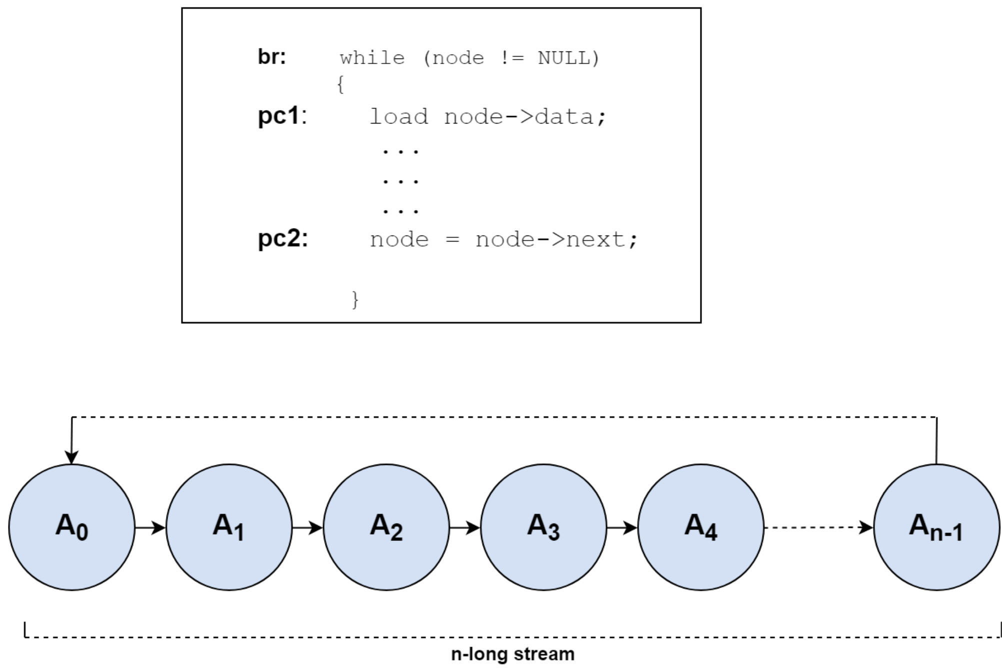

Figure 15: Linked list traversal: correlated value stream

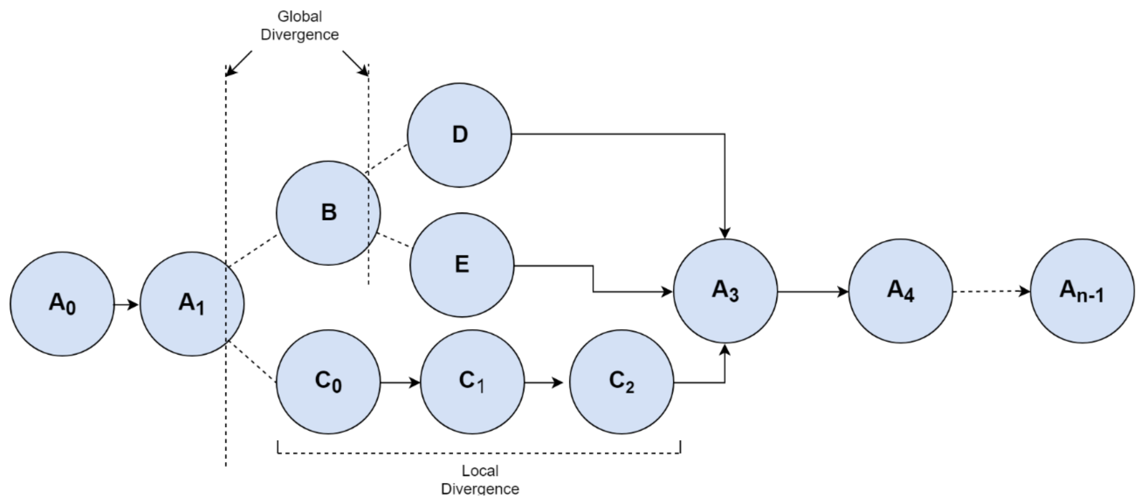

Figure 16: Graph traversal, local and global divergence

Branch Context

Branch History (4 bit)	Value
0000	A_0
0001	A_1
0011	A_2
0111	A_3
1110	A_0
1101	A_1
1011	A_2

Value Context

Value History	Value
A_0	A_1
A_1	A_2
A_2	A_3
A_3	A_0

Figure 17: Prediction using branch and value contexts

5.1.1 Divergence Handling using Branch Contexts

Similar to Figure 15 above, consider traversing a four-node linked list repeatedly. To correctly learn the values generated by the instruction *pc1* using just branch information, a method such as VTAGE [15] looks at the outcome history of the branch *br* in the program. Given a branch history register of (at least) four bits, we can isolate the values produced by *pc1* such that no two values have the same branch history information, as elucidated in Figure 17 on the left. The reader may convince themselves that this indeed is the case.

While this allows a predictor that uses branch contexts to accurately predict streams of values, the stored length of the branch history limits the maximum predictable length of the value stream before aliasing of histories starts affecting accuracy. In particular, for a *n-long* value stream as shown in Figure 15, a minimum branch history length of *n* is required. As value streams can be of arbitrary length, using branch contexts presents a problem in handling local divergence.

On the other hand, as global divergence occurs solely due to the path of control flow chosen, it is natural to use branch history information to resolve this form of divergence.

5.1.2 Divergence Handling using Value Contexts

Let us consider the linked-list example in Figure 15 again. An intuitive way of predicting the next value in the stream is through learning pairwise correlations. For instance, an FCM [6] predictor would learn that A_{i+1} always follows A_i except at the last value of the steam. This is shown in Figure 17.

While this scheme of learning using value contexts works very well in handling local divergence, it is ill-suited to handle global divergence. With a single value history, multiple values may follow any given value. For instance, while traversing a graph such as in Figure 16, values B or C_0 may follow A_1. As no control flow information is available, an FCM-like predictor substitutes it with longer value history, which acts a proxy for the control flow path taken.

We argue that this is inherently an inefficient way of dealing with global divergence as potentially a very long value history is required to substitute branch history. Consider the graph traversal example in Figure 16 again. If the two values B and C_0 are on equally likely paths of a branch, a value history of twice the length of the entire stream is required to accurately predict which value follows A_1. However, a single bit of relevant branch history would suffice to predict the global divergence.

While the reader may validly argue that using appropriately long branch or value histories would resolve all the issues of handling divergence, a predictor that employs long context information is bound to be limited by the training time, and hence have poor coverage. Moreover, the area overheads of using long contexts makes it a poor design

decision. This dichotomy between value history being efficient at handing local divergence and branch history being efficient at handling global divergence necessitates a design that uses the *relevant context information* to make predictions.

5.2 COMBINING EVES AND DFCM++:

As described in the previous section, we hypothesize that combining branch and value contexts may provide benefit in accuracy and training time. As existing predictors such as EVES [2] employ only branch histories and DFCM++ [1] employs only value histories, we conduct an experiment that combines the two predictors in an oracle manner. We make the following observations:

1. Perfectly combining the two predictors gives us better coverage than DFCM++ which implies that branch history helps add additional predictions that are not learnt by value history alone. Figure 18 below shows that coverage increases 4% w.r.t DFCM++. On the other hand, value history helps increase coverage of EVES by 20%, implying that value contexts handle local divergence better.
2. The combined predictor gives better accuracy than DFCM++, possibly correcting the inaccuracies due to global divergence in DFCM++. This is depicted in Figure 19.
3. As seen in Figure 20, the combined predictor provides a speedup of 3.7% over EVES, while improving DFCM++ by over 13%.
4. On average, out of all the correct predictions made, both predictors predict correctly 63% of the time, while DFCM++ predicts correctly 25% of the time when EVES does not predict/mispredicts. EVES predicts correctly

5% of the time when DFCM++ misses to predict. This is illustrated in Figure 21.

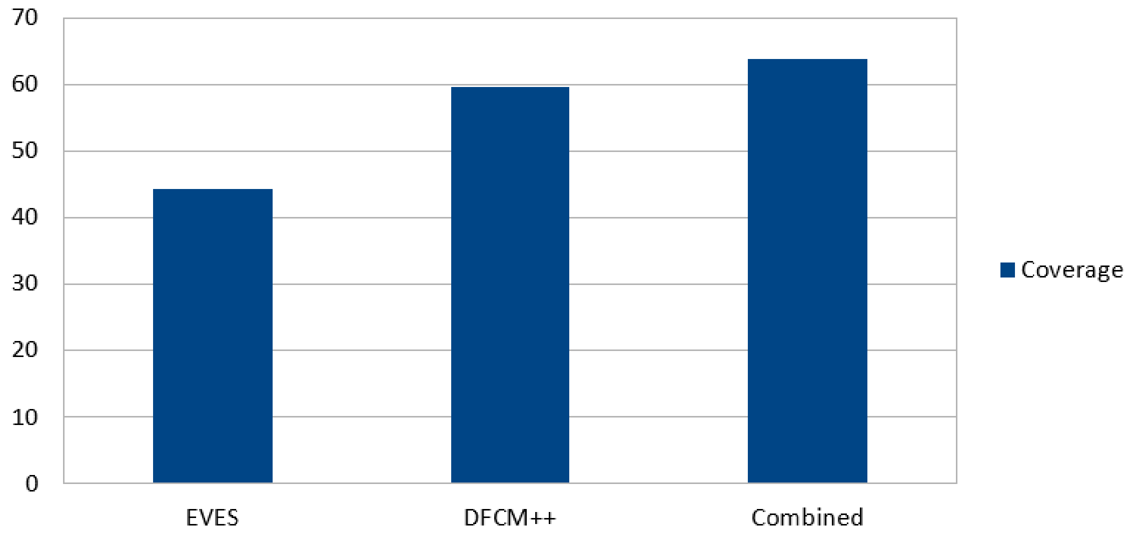

Figure 18: Coverage on combining predictors

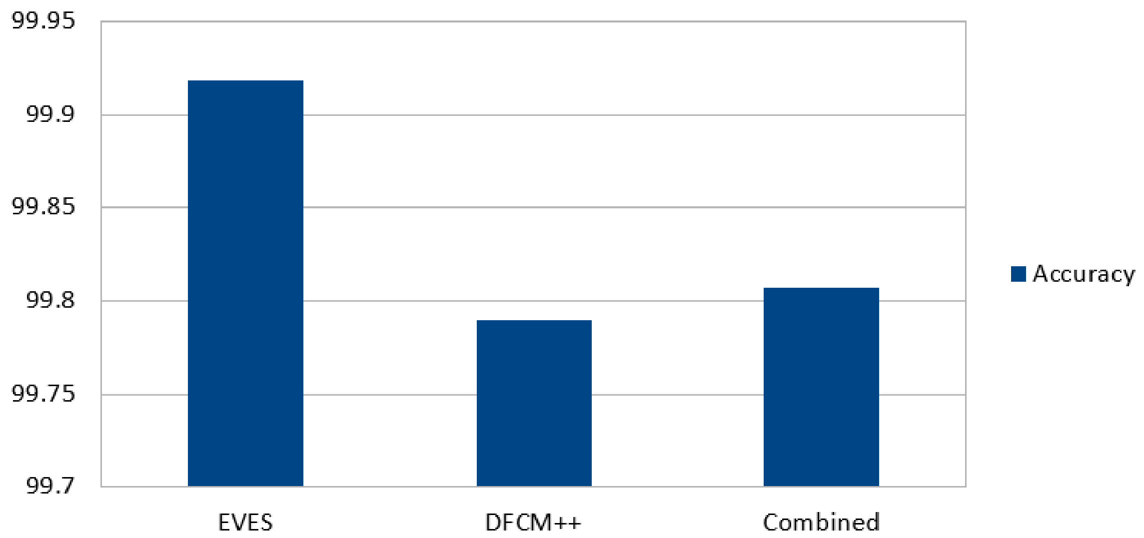

Figure 19: Accuracy on combining predictors

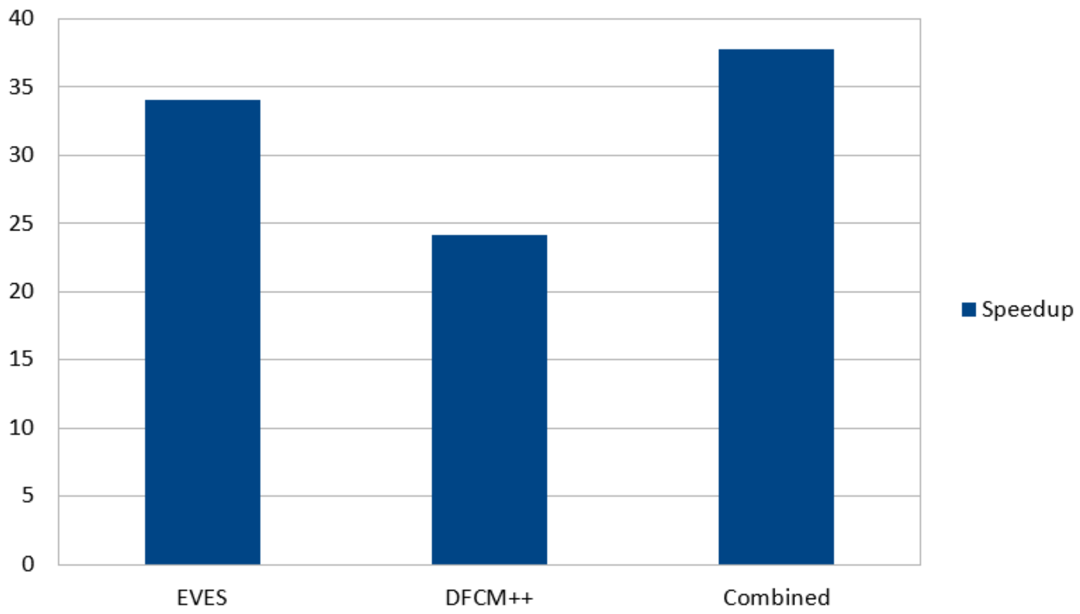

Figure 20: Speedup on combining predictors

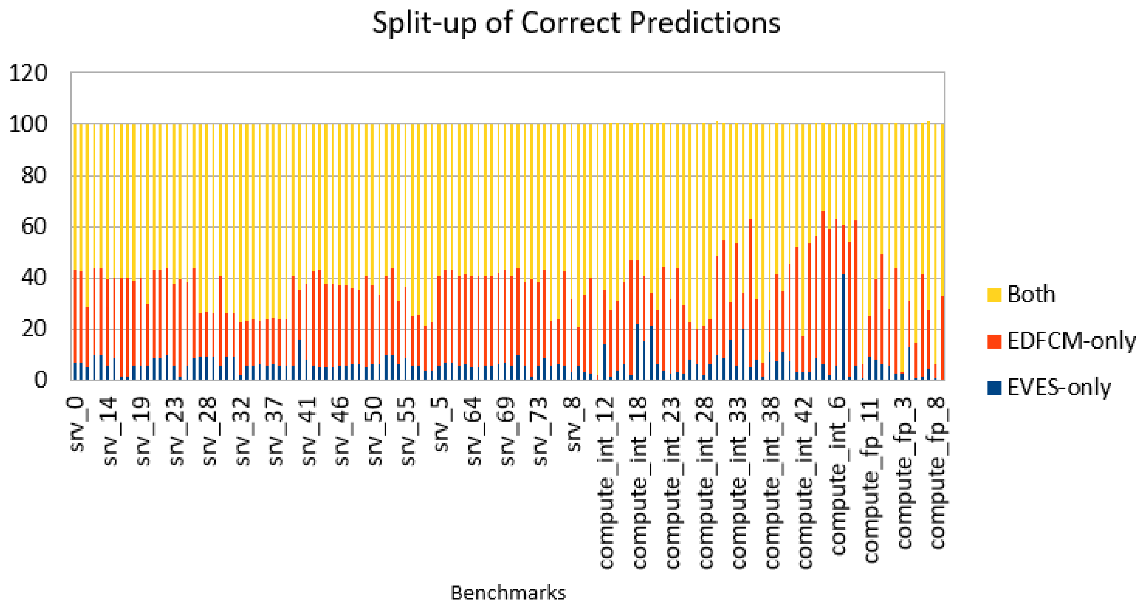

Figure 21: Correct prediction by each component of hybrid

5.3 Employing Relevant Context Information: The Relevant Context-based Predictor (RCP)

As performance of a value predictor is contingent on the accuracy of predictions and coverage, appropriate context information is key for a high-performance value predictor. A context may include PC information, branch history and/or value history. To choose the best combination of value and branch histories as part of the context, we define two metrics that quantify the quality of a context. First, we measure the *variance* of a particular context, defined as the number of unique data values seen by a context. Intuitively, predicting a context that has lower variance is expected to be more accurate. Variance is a measure of the localization capability of the context. It goes down with richer context information and is equal to 1 for a context whose value does not vary.

We further measure the *predictability* of data values observed with a context, which is defined as the percentage of all contexts that have a variance = 1 and repeat at least a set number of times. We then employ this information to choose the best combination of value and branch contexts.

As seen in Figure 22 and Figure 23 below, we observe that average variance across contexts reduces as we increase branch history and value history length. However, while at 1024 bits of branch history the variance is nine values, it is significantly higher at 250 values for a value history of 16 64-bit values. This provides evidence that branch history achieves better localization. When we combine branch and value histories, we observe that the localization achieved is almost double that of the using branch history alone. This is shown in Figure 24.

As contexts may have low variance but may never repeat, rendering them unpredictable, we measure the percentage of contexts that show variance = 1 and repeat a set number of times. Setting this threshold to 4, we observe that the predictability of

value contexts is higher than branch contexts. This is expected as long branch contexts do not repeat as often, and hence incur a training time penalty. We observe that combining branch and value histories yields the best results even at short history lengths. This is illustrated in Figure 25. It is to be noted that we combine contexts by using equal number of bits of branch history and value history, where values are 64-bit each.

In conjunction with the qualitative analysis in the Section 4.1, we see that the most predictable values are obtained in contexts using a short value history (1 value) that handles local divergence augmented with branch history (64 bits) to handle global divergence. We design a PC-localized predictor that employs 64-bit branch history and a single value history to make predictions and term it the Relevant Context-based Predictor (RCP). Figure 26 shows the speedup obtained by employing RCP that combines 64 bits of branch history and a single value history, by varying the confidence threshold. We observe a geomean speedup of 21% over a baseline that performs no value prediction across the 135 benchmarks.

After combining a simple stride predictor with RCP, we compare it against the state-of-the-art value prediction mechanisms. We perform better than the schemes that solely employ branch histories (EVES) and value histories (DFCM++) owing to our better divergence handling capabilities, achieving a geomean speedup of 38% over no value prediction. Table 2 shows the comparison between the three predictors. We observe that we lose some coverage as compared to the EVES-DFCM++ hybrid at the expense of better accuracy.

Table 2: Comparing our predictor against EVES, DFCM++

	EVES	DFCM++	RCP
Accuracy (%)	99.91	99.78	99.88
Coverage (%)	44.26	59.57	50.66
Speedup (%)	34	24	38

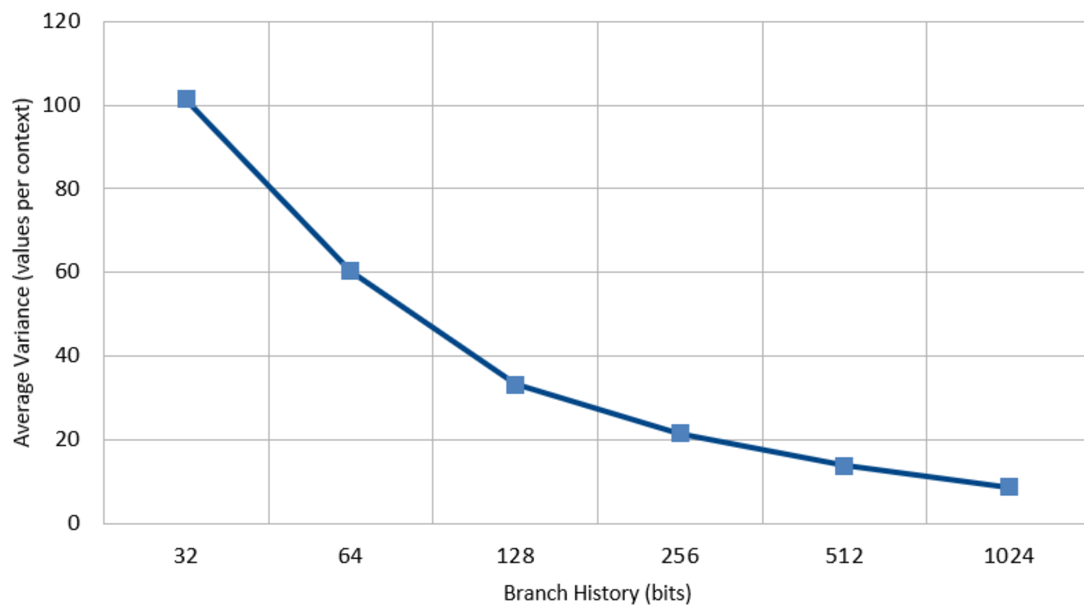

Figure 22: Variance of branch contexts

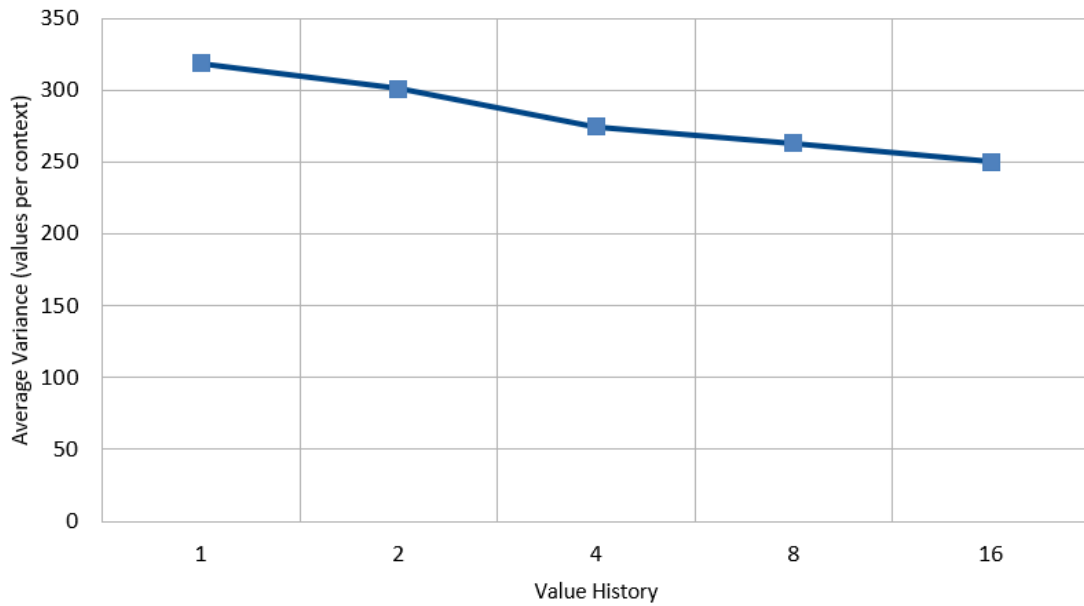

Figure 23: Variance of value contexts

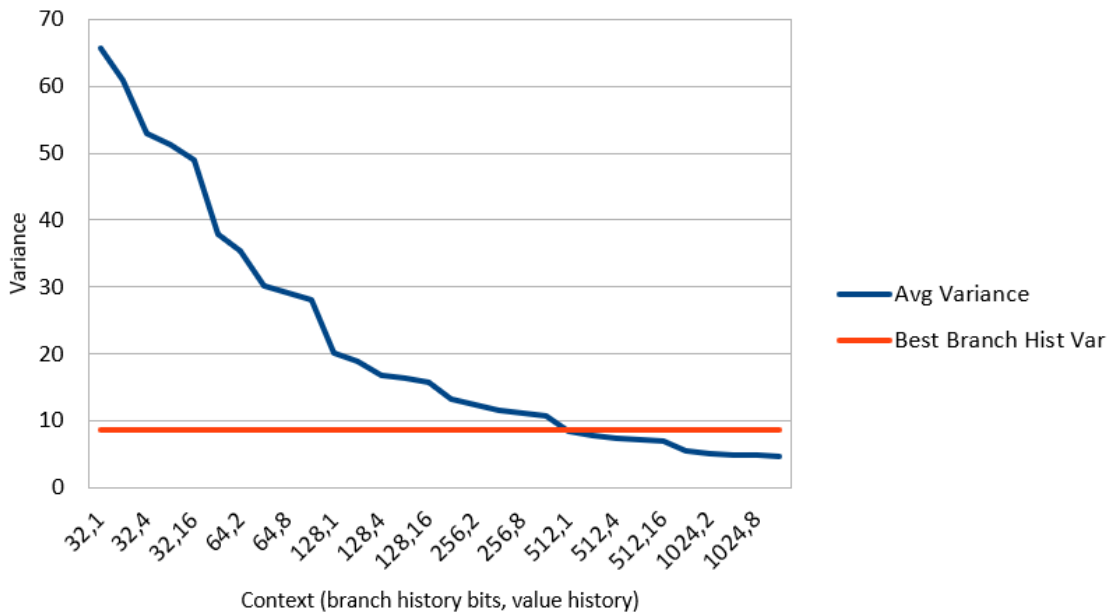

Figure 24: Variance upon combining contexts

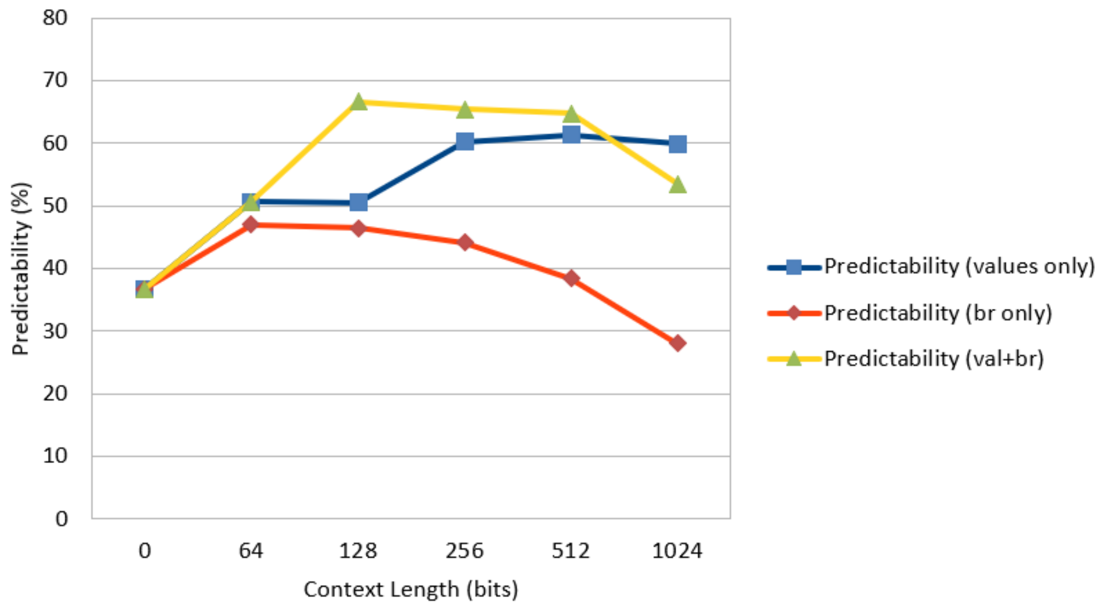

Figure 25: Predictability of values for different contexts

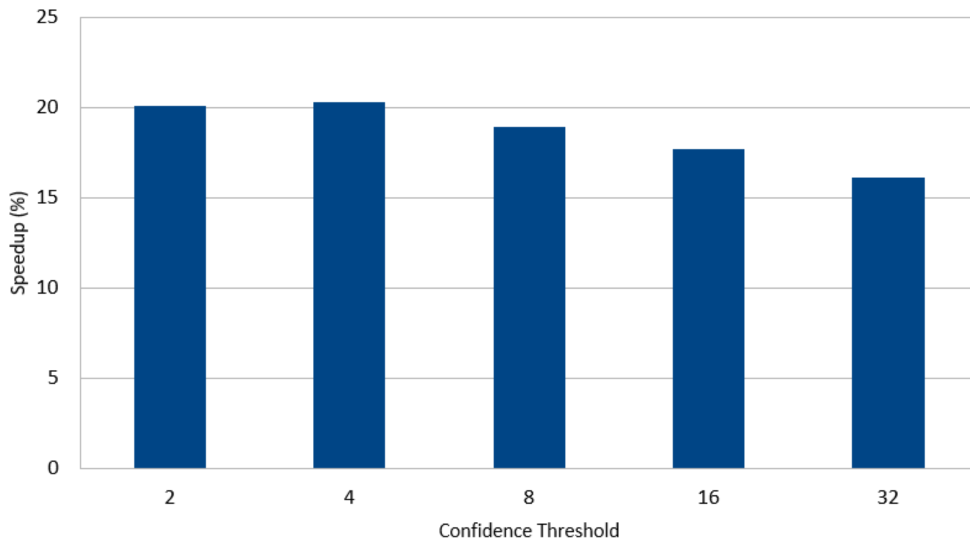

Figure 26: Speedup over no VP, obtained using {64-bit branch hist, 1 value} context

Chapter 6: Conclusion

6.1 Limitations and Future Work

We identify that some aspects of our work need a more thorough investigation. In this work, we do not model the impact of limited predictor structure sizes or the impact of mechanisms that off-load metadata to the memory hierarchy. We anticipate the performance to be impacted by limited predictor size and a negative interaction with the caches, although an intelligent management policy that involves the value predictor, I-TLB and the cache replacement policies should be able to make the appropriate tradeoffs. Such policies have been implemented for irregular prefetchers [8].

Further, we do not model memory subsystem optimizations such as prefetchers, which help hide the latency of memory accesses to some extent. We anticipate that some of the performance gained by value prediction would be lost due to prefetching, but our initial experiments show that by perfectly predicting all the L1 cache hits and the ALU operations, we can achieve up to 200% speedup over a baseline that performs no value prediction.

A key insight provided by EVES [2] and DFCM++ [1] is that the use of an array of context lengths helps training time and hence coverage. While we improve coverage in our predictor through the use of a single combination of branch and value contexts, a more sophisticated design would involve a combination of different history lengths of branch and value histories in the same predictor. We anticipate further improvements in predictor performance through such a mechanism.

Finally, there exist several limitations due to the choice of our simulation infrastructure. The CVP [9] infrastructure is limited to using only the Qualcomm-provided traces. The speedups obtained by the predictors on other benchmark suites were

not evaluated in this work. Further, the infrastructure assumes fixed microarchitectural parameters related to caches, branch prediction, and memory disambiguation, which limits analysis. An implementation in a cycle-accurate environment such as *gem5* can provide much more flexibility in design and analysis and provide insights on overall system performance.

6.2 Conclusion of the Thesis

General-purpose computing has seen a slowdown in improvements as Dennard scaling and Moore's law are fading or almost gone. While multicore computing provides an attractive alternative to achieve improved performance for some categories of workloads, it is limited by the sequential portion of the workloads, as stated by Amdahl's Law [11]. Moreover, some kinds of workloads cannot be parallelized. As a result, single-core performance is still a performance bottleneck, and value prediction is targeted at a wide category of users who run sequential and partially parallelizable workloads.

However, implementing value prediction is bound to have high area and energy overheads unless intelligent methods to manage the predictor state are introduced. To that end, we introduce an enhanced version of an irregular prefetcher ISB, which is capable of off-loading the predictor metadata to the memory hierarchy. This allows a small fraction of the predictor state to be cached in the processor core.

Existing value predictors either employ branch history contexts [2] or value history contexts [1] to make predictions. We demonstrate that control flow divergence in programs necessitates the use of very long histories to achieve high accuracy. As such, existing approaches slow down the training time of the predictor and hence achieve low coverage. We identify that branch and value histories provide mutualistic advantages to a value predictor in terms of handling control flow divergence, and therefore we combine

them in a novel predictor design called the Relevant Context-based Predictor (RCP). Our predictor maintains high accuracy while improving training time, achieving an average of 38% speedup over a baseline that performs no value prediction on the Qualcomm-provided traces.

BIBLIOGRAPHY

[1] N. Deshmukh, S. Verma, P. Agrawal, P. Biswabandan and M. Cahudhuri, "DFCM++: Augmenting DFCM with Early Update and Data Dependence-driven Value Estimation," in *CVP-1 1st Championship Value Prediction*, Los Angeles, USA, 2018.

[2] A. Seznec, "Exploring Value Prediction with the EVES predictor," in *CVP-1 1st Championship Value Prediction*, Los Angeles, USA, 2018.

[3] "CVP," 2018. [Online]. Available: https://www.microarch.org/cvp1/.

[4] G. Moore, "Cramming more components onto integrated circuits," in *Proceeding of the IEEE*, Jan 1998.

[5] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the Spring Joint Computer Conference*, 1967.

[6] Y. Sazeides and J. Smith, "The predictability of data values," in *In Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1997.

[7] B. Goeman, H. Vandierendonck and K. Bosschere, "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency," in *Proc. Seventh Int'l Symp. High Performance Computer Architecture*, 2001.

[8] A. Jain and C. Lin, "Linearizing Irregular Memory Accesses for Improves Correlated Prefetching," in *MICRO*, 2013.

[9] A. Seznec and P. Michaud, "A case for (partially) TAgged GEometric history length branch prediction," in *Journal of Instuction Level Parallelism*, Feb. 2006.

[10] "Championship Branch Prediction," Held in conjunction with the Internation Symposium in Computer Architecture, 2014. [Online]. Available: http://www.jilp.org/cbp2014/program.html.

[11] *Standard Performance Evaluation Corporation. CPU2006.*.

[12] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2005.

[13] M. Lipasti and J. Shen, " Exceeding the dataflow limit via value prediction," in *In Proceedings of the annual International Symposium on Microarchitecture (MICRO)*, 1996.

[14] F. Gabbay and A. Mendelson, "Speculative Execution Based on Value Prediction," in *Technion TR-1080*, 1996.

[15] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *In Proceedings of the International Symposium on High Performance Computer Architecture*, 2014.

[16] R. Sheikh, H. Cain and R. Damodaran, "Load Value Prediction via Path-based Address Prediction: Avoiding Mispredictions Due to Conflicting Stores.," in *In Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.

[17] K. Nesbit and J. Smith, "Data Cache Prefetching Using a Global History Buffer," in *HPCA*, 2004.