Exploiting Value Prediction With Quasi-Unlimited Resources

Arthur Perais

To cite this version:

Arthur Perais. Exploiting Value Prediction With Quasi-Unlimited Resources. Hardware Architecture [cs.AR]. 2012. dumas-00725221

HAL Id: dumas-00725221
https://dumas.ccsd.cnrs.fr/dumas-00725221v1

Submitted on 24 Aug 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Exploiting Value Prediction With Quasi-Unlimited Resources

Internship Report
ISTIC - Research Master in Computer Science (MRI)

Author : Arthur Perais (arthur.perais@etudiant.univ-rennes1.fr)
Supervisor : André Seznec (seznec@irisa.fr)
Team : ALF - Amdahl's Law is Forever

Contents

1 State-of-the art **6**
1.1 Superscalar Processors and Value Prediction . . . 6
1.2 Value Predictors . . . 7
1.2.1 Computational Predictors . . . 8
1.2.2 Finite Context Method (FCM) Predictors . . . 9
1.2.3 Hybrid Predictors . . . 11
1.3 Geometric Length Branch Predictors . . . 13
1.3.1 TAGE and ITTAGE . . . 13

2 Validating the Choice of Geometric Length Value Predictors **16**
2.1 Experimental Framework . . . 16
2.2 Experimental Results . . . 16
2.2.1 Global Branch History . . . 16
2.2.2 Path History . . . 17
2.3 Conclusion . . . 17

3 The Value TAgged GEometric Predictor **19**
3.1 From ITTAGE to VTAGE . . . 19
3.1.1 Predictor Structure . . . 19
3.1.2 Confidence Estimation . . . 19
3.2 Building a VTAGE-Based Hybrid Value Predictor . . . 20
3.2.1 Computational Component . . . 20
3.2.2 Predictor Configurations . . . 21
3.2.3 Arbitrating Between the Two Components . . . 21

4 Performance Evaluation of the FS-VTAGE Predictor **23**
4.1 Objectives . . . 23
4.2 Performance Evaluation on Integer Workloads . . . 23
4.2.1 Experimental Framework . . . 23
4.2.2 Complementarity . . . 24
4.2.3 Accuracy and Coverage . . . 24
4.3 Performance Evaluation on Floating-Point and Vector Workloads . . . 29
4.3.1 Experimental framework . . . 29
4.3.2 Complementarity . . . 31
4.3.3 Accuracy and Coverage . . . 31
4.4 Improving the FS-VTAGE Performance . . . 34
4.4.1 Augmenting The VTAGE Component with *Per-Instruction* Confidence . . . 34
4.4.2 Using Confidence Estimation instead of *Filtering* in Stride . . . 37
4.4.3 Performance Evaluation of the Updated FS-VTAGE . . . 38

5 Future Work **41**
5.1 Predicting Memory Instruction Addresses and *Store* Values 41
5.2 Criticality 41
5.3 Microarchitectural Considerations 42
5.3.1 Misprediction Recovery 42
5.3.2 Prediction Resolution 42
5.3.3 Perspectives Regarding Branch Prediction 43
5.4 Vector Value Prediction 43

Introduction

Recent trends regarding general purpose microprocessors have focused on Thread-Level Parallelism (TLP), and in general, on parallel architectures such as multicores. Specifically, architects have been forced to turn themselves towards parallel architectures since adding transistors to enhance sequential performance does not yield as tremendous improvements as it used to.

However, due to Amdahl's law, the gain to be had from the parallelization of a program is limited since there will always be an incompressible sequential part in the program. The execution time of this part only depends on the sequential performance of the processor the program is executed on. Combined to the fact that a fair amount of software has not been written with parallelism in mind (and will probably never be rewritten or recompiled), there is still demand for improvement of modern microprocessors sequential performance.

One way to improve it is to leverage *Instruction Level Parallelism* (ILP), that is the presence in the code of independent instructions which can be executed in parallel. *Superscalar* processors were proposed to take advantage of ILP as they are able to execute several instructions per cycle. However, ILP has its limits and no processor with a superscalar width greater than six was ever released - Namely, the Intel Itanium - because empirically, there are seldom cases where more than a few instructions from the same control flow can be executed in parallel [9]. This observation can partly be explained by both control and data dependencies between instructions close in the code. Finding a way to break these dependencies would lead to direct sequential performance gains.

Control dependencies are handled via Branch Prediction, which consists in predicting the outcome of a branch before it is actually resolved. That way, the processor can continue fetching instructions instead of waiting for the branch resolution, hopefully on the correct path. True data dependencies (producer/consumer relations between two instructions), on the contrary, are always enforced.

Value Prediction proposes to speculate on the result of instructions in order to break true data dependencies. This effectively creates more ILP [16, 7] and allows for early speculative execution of dependent instructions. Value Prediction is based on *Value Locality*, that is the observation that instructions tend to produce results that were already produced by a previous instance of the instruction [13]. However, previous work in this field has provided us with moderately accurate predictors [14, 18, 26] when accuracy is critical due to misprediction penalties [27]. In this report, we aim to demonstrate that it is possible to design a very accurate hybrid predictor which does not trade accuracy for coverage. That is, a predictor which is able to accurately predict a reasonable amount of instructions. We also expect that many predictions will relate to instructions on the internal *critical-path* of the processor, where the potential gain of Value Prediction is the highest.

In the first chapter, we will focus on state-of-the-art by introducing superscalar processors and quickly reviewing existing value predictors. In the meantime, we will also describe the TAGE branch predictor and more specifically its sibling the ITTAGE indirect branch target predictor [21, 22, 24]. In the second chapter, we will validate the intuition that using a modified ITTAGE predictor for Value Prediction is worthwhile. Chapter 3 is dedicated to the description of a new hybrid value predictor composed of a Value TAGE predictor and a Stride-based value predictor. Chapter 4 focuses on the performance of this hybrid predictor on integer workloads and compares them to those of existing predictors. It also covers the perspective of

using Value Prediction in the presence of floating-point [1] arithmetic as well as vector (Single Instruction Multiple Data) instructions. Finally, chapter 5 gives directions for future work.

Chapter 1

State-of-the art

1.1 Superscalar Processors and Value Prediction

Superscalar processors differ from non-superscalar processors in the way that they can issue and execute several instructions per cycle, therefore having a potential *Instruction Per Clock* (IPC) greater than 1. For instance, consider the 2-wide superscalar pipeline in Fig. 1.1. In essence, it can be seen as two scalar pipelines placed side by side to allow the execution of two instructions per clock. Naturally, to execute two instructions per cycle, functional units are duplicated. That is, in a superscalar processor, the execution units are redundant to some extent.

Instructions can be executed *in-order* (static scheduling) or *out-of-order* (dynamic scheduling). *In-order* execution means that an instruction cannot begin (resp. finish) execution before an instruction that was located before it in the original code begins (resp. finishes) execution. Fig. 1.1 typically describes such a microprocessor pipeline.

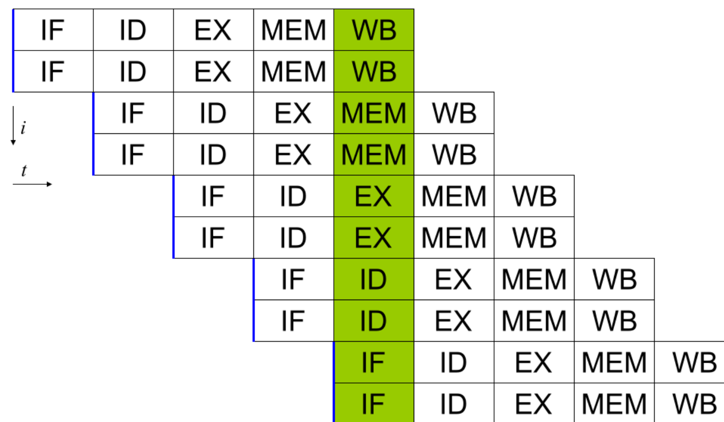


Figure 1.1: 2-wide superscalar 5-stage pipeline (Fetch, Decode, Execute, Memory, Write-back).

The major hurdle of static scheduling is that a long-latency instruction will forbid all subsequent instructions to proceed even if these subsequent instructions do not depend on the said instruction. To address this issue, *Out-of-order* execution was proposed. With this feature, any instruction whose operands are ready can begin execution, regardless of its original position in the code. Informally, the rough idea is that in superscalar processors, once decoded (in order), instructions are stored in a finite queue (*instruction queue*). Every cycle, n instructions whose operands are ready and for which an execution unit is available are selected

in the queue. In the case of *in-order* execution, the n instructions starting from the **head** of the queue can be selected, at best. In the case of *out-of-order* execution, **any** combination of n instructions can be selected in the queue, as long as they meet the requirements of having their operands ready and an execution unit available. That way, independent instructions can bypass an instruction which is blocked in the pipeline for some reason (structural hazard, data dependency, cache miss and so on). Note that to enforce the program semantics and guarantee precise interruptions, the instructions must be committed in order.

Because of execution units redundancy, several instructions can execute at the same time, but potential performance is lost when some execution units are idle. Even though *out-of-order* execution usually improves performance, an instruction will never issue if its operands are not available, and a single long-latency instruction can forbid all subsequent instructions to proceed if it is on the *critical path*. This limitation is referred as the *dataflow limit* by Lipasti and Shen in [12]. It stems from the simple fact that some instructions depend on the results produced by others (producer/consumer relation) and *cannot* execute before them because of the program semantics. A similar limit - the *control-flow* limit - applies to branches: theoretically, an instruction following a branch instruction cannot execute until the branch is resolved because it may or may not have to execute at all depending on the outcome.

However, the processor can speculatively execute instructions as long as it is able to rollback incorrect changes speculatively made to the architectural state. That is, as long as the sequential semantics of the program are enforced. The main example of speculation is *Branch Prediction* which makes use of small tables to store the speculative outcome of future branches. Predictions are usually generated by observing the behavior of past branches. This mitigates the *control-flow* limit. To mitigate the *dataflow* limit, *Value Prediction* aims to predict the result of an instruction before it is executed, artificially increasing ILP. With this predicted result, dependent instructions can execute earlier and the total execution time can be decreased. Value Prediction works because for a given static instruction, many instances of that instruction will either produce the same result, a result that was previously seen or a result that is a simple function of the previous result. This phenomenon is referred to as *Value Locality* [12].

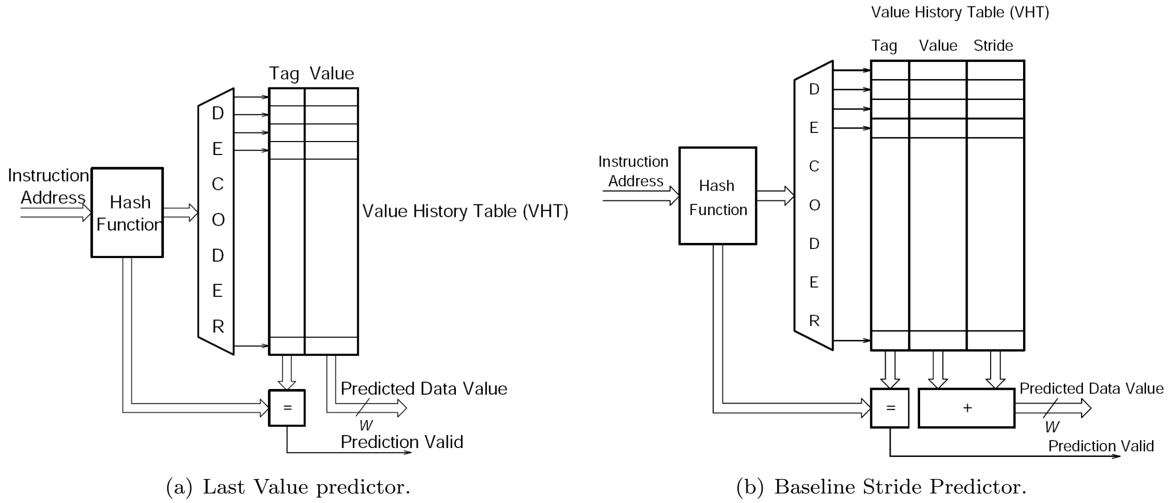
As stated in the introduction, the width of available superscalar processors is rather low and increasing it would not be worthwhile due to the limited ILP on most code. Yet, Value Prediction is able to create - even if speculatively - ILP. That is, as well as its potential for increasing sequential performance on modern microprocessors, Value Prediction might also reopen the door to very wide superscalar processors because of the increased potential ILP. Therefore, instead of keeping on adding cores and following the *Thread-Level Parallelism* paradigm, one can imagine dedicating more silicon area to fewer very wide superscalar cores to improve sequential performance and let the intrinsically parallel workloads be run on dedicated hardware (such as GPUs). This would be another step towards heterogeneous architectures.

1.2 Value Predictors

As demonstrated by Tullsen and Seng in [25], using predictors is not the only way to implement Value Prediction. However, it has advantages such as using structures similar to the well known branch predictors, not requiring any work from the compiler or software in general, and finally, being more accurate than what can be seen in [25]. Therefore, we focus only on value predictors in this report.

Sazeides and Smith define two types of value predictors in [18]. *Computational* predictors generate a prediction by applying a function to the value produced by the previous instance of the instruction. *Context-Based* predictors rely on patterns in the value history of a given static instruction to generate the predicted result of the current instance. The main representatives of both types of value predictors are described in this section. Finally, several hybrid predictors - featuring a computational component along a context-based component - are detailed.

In all cases, the evaluation of the predictors discussed in this section were made on different benchmark suites with different experimental frameworks. As a consequence, we reimplemented most of them and evaluated them using our own experimental framework. We will provide a detailed description of the setup in 4.2.1 for integer workloads and in 4.3.1 for floating-point and vectorized workloads. Results (accuracy



(a) Last Value predictor.

(b) Baseline Stride Predictor.

Figure 1.2: Baseline computational value predictors. From [26]

and coverage) will be given later in 4.2.3 (integer) and 4.3.3 (FP and vectorized code).

1.2.1 Computational Predictors

Last Value Predictor

The simplest computational predictor is the Last-Value predictor. As its name suggests, for a given dynamic instruction, it tries to predict the result produced by the last instance of the same static instruction. It is usually implemented as a simple table (potentially set-associative). An entry of the table contains a tag corresponding to the highest bits of the PC as well as the last value seen by the corresponding instruction. The table is accessed by the lowest bits of the PC, much like in a cache. While this scheme may appear very simplistic, it yields a small speedup. Specifically, Lipasti and Shen showed in [12] that a realistic Last-Value predictor (the predictor only stores the most recently seen value, as opposed to choosing from several previously seen values) yields a 4.5% speedup on programs taken from SPECINT92/95 as well as other UNIX utilities. A Last Value Predictor is shown in Fig. 1.2(a).

Stride and Variations

Stride predictors were proposed mostly to take advantage of loop induction variables. The result of instructions located in the body of a loop often directly depend on the value of the loop induction variable. For instance, when an array is read sequentially by a loop, the effective address of the *load* instruction is usually of the form : *Base + size_of_element * loop_index*. In this case, adding the corresponding stride to the last predicted value yields better results than simply returning the last value. The Stride predictor is essentially a Last-Value predictor in which the entries have been augmented with a *stride* field. In theory, the width of this field should be the width of a register value. However, for 32-bit registers, 8 bits appeared to be sufficient [12]. A similar behavior can be expected when register width is 64 bits. The baseline Stride Value Predictor is shown in Fig.1.2(b).

A possible improvement over this scheme - the *two-delta* stride predictor - is presented by Eickemeyer and Vassiliadis in [4] and uses two strides to add hysteresis: The stride used to compute the prediction is updated only when the second stride has been observed at least two consecutive times. The second stride is updated every time a prediction is made. In our implementation, this predictor behaves as a Last Value Predictor when a new entry is allocated (the stride used to make the prediction is 0 until it is updated).

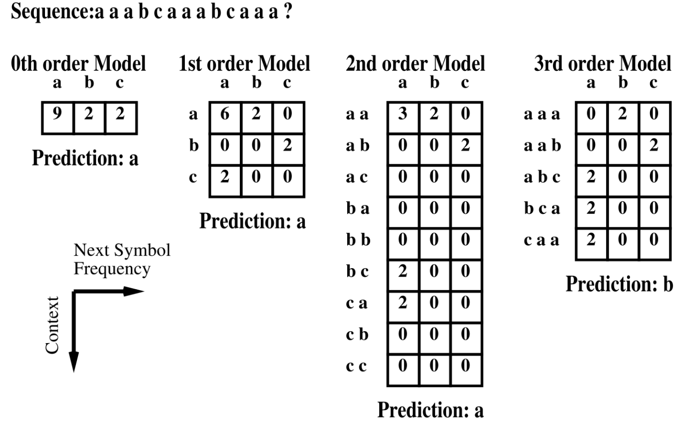


Figure 1.3: Theoretical FCM predictor. From [18]

Other variations include saturating confidence counters to only predict when the entry is confident enough (confidence is incremented by a certain amount when the prediction is correct and decremented by another amount otherwise).

Gabbay and Mendelson proposed a stride-based predictor [5] specifically tailored for predicting floating-point results [1]. The SEF predictor is more accurate than a Last Value or a Stride predictor because it tries to predict each component of a floating-point value separately. That is, a stride is kept for both the mantissa and the exponent. However, we chose not to use it because first, it cannot predict both single and double precision floating-point results out of the box, and second, in the event of a program without FP instructions, all this predictor storage would be wasted, rendering it highly inefficient. Nevertheless, it is surely possible to mix a "regular" Stride predictor with the SEF predictor to improve accuracy and coverage for FP workloads.

1.2.2 Finite Context Method (FCM) Predictors

Ideal FCM Predictor

A FCM predictor of order n uses the last n values in the value history of a given static instruction as its input to generate the speculative result of the current instance. For each possible pattern, the predictor stores saturating counters associated with each relevant value. A high value for a counter means that the associated prediction is very likely to be correct, and reciprocally. For instance, consider Fig. 1.3. The chosen prediction is the value associated with the highest counter matching the current context (history). The base FCM predictor is rather idealistic since for 64-bit registers, the potential number of counters the predictor has to store is 2^{64}, per pattern. Hence, realistic FCM predictors limit the number of counters stored per pattern (that is, the number of actual values that can be chosen by the predictor as the prediction for a given pattern) as well as the value history depth which is often limited to one digit [17, 18].

2-level Predictor

This predictor is a somewhat realistic FCM predictor. It was introduced by Wang and Franklin in [26] and is described in Fig. 1.4. It features two tables. The first one is the *Value History Table* (VHT) which can store up to four values per entry as well as a *Value History Pattern* (VHP) consisting of p two-bit fields. Each value stored in the VHT is assigned to the binary encoding {00,01,10,11} so that the VHP effectively records the order in which the last p results of the corresponding static instruction were seen during execution.

The second table is the *Pattern History Table* (PHT). It stores four counters per entry. The PHT is indexed by p-length bit-vectors from the VHP. When a PHT entry is selected, its four counters are compared

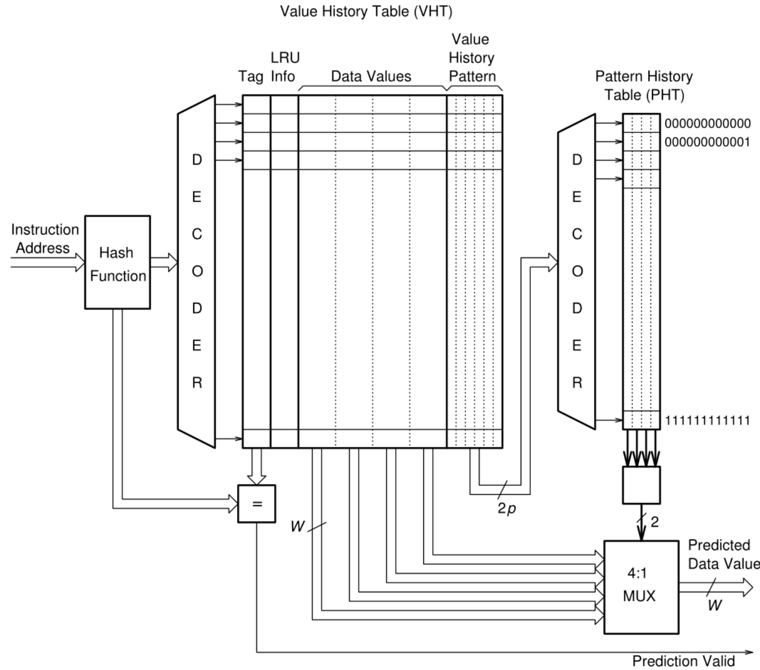


Figure 1.4: 2-level FCM predictor. From [26]

against each other and the one with the greatest value is selected. If its value is greater or equal to a given threshold, the value in the VHT whose position corresponds to the one of the counter in the PHT is selected as the prediction. In essence, this is a pth order FCM with a history only composed of four different values (but still p long).

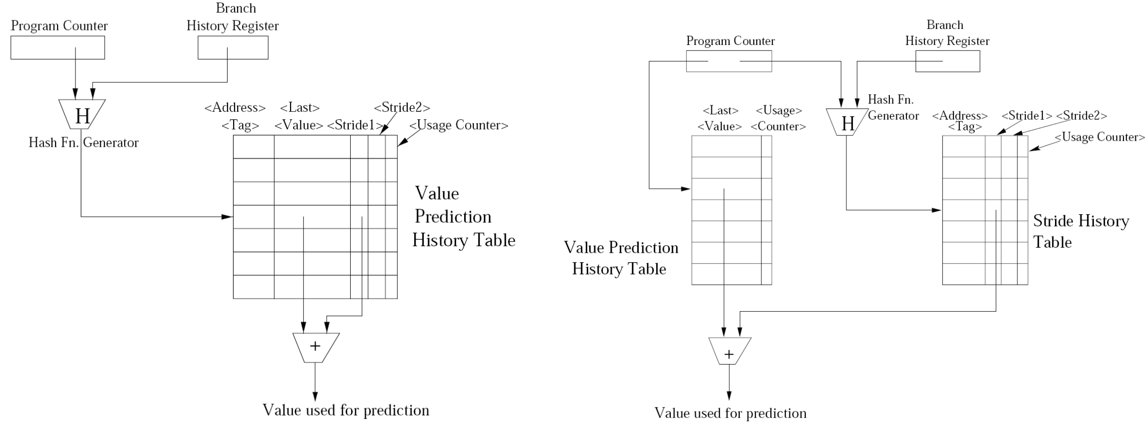
In further experiments presented in chapter 4, we will use this "order 6" (4 different values are tracked) 2-level predictor as well as an "order 8" version featuring a 24-bit history (recording the last 8 values seen because one outcome is now encoded on three bits) and thus a 2^{24}-entry PHT with 8 4-bit counters per entry. We will refer to it as the 2-level+ predictor. This version will only serve to study ideal performance since the PHT only is roughly 537 Mbits in size.

Note that using local value histories makes this predictor very hard to implement if we place ourselves in the context of a very wide and aggressively speculative processor. Indeed, keeping local histories up to date when the processor is speculating on both the branches and the values would be very difficult.

Per-path Stride predictor (PS)

In [14], Nakra, Gupta, and Soffa propose to use the branch history to better establish the current context in which a given dynamic instance is executed, in order to help predict its result. Even though the branch history is used (as opposed to the value history), it still classifies as a FCM predictor. In essence, they propose to modify a 2-delta Stride predictor by hashing the branch history and the PC to access the predictor instead of just using the PC. Actually, the predictor either selects both the stride and the value to which the stride should be added by hashing the PC with the branch history (Fig. 1.5(a)) or selects the stride with the result of the hash and the value with the PC (Fig. 1.5(b)).

The interest of only associating the stride to the context is that the last result of the instruction is stored only once for all the strides, at the cost of the additional tag in the table containing strides. However, associating both the stride and the value to the context should not impede accuracy as it will only consume more space. Therefore, we will only consider the second predictor (1.5) because storage is less of an issue in



(a) Branch history is used to access both the value and the stride.

(b) Branch history is only used to access the stride.

Figure 1.5: Per-path Stride Predictors. From [14].

the context of our work. One should note that the width of the global branch history considered in [14] is 2, meaning that predictions are only correlated with the two most recent branches. The hash function used by the PS predictor is the following: for i the address of the instruction and b the value of the Branch History Register (BHR), $f(i, b) = ((Addr(i) << Size(BHR)) + b)\ mod\ Size(VPHT)$.

1.2.3 Hybrid Predictors

As stated in [18], context-based predictors are able to capture results that computational predictors cannot, but the opposite is also true. Therefore, constructing hybrid predictors consisting of a computational and a context-based predictor seems to be a natural way of achieving high accuracy. Furthermore, since context-based predictors tend to require larger table to exceed the accuracy of computational predictors, building a hybrid may allow the size of the context-based predictor to be reduced because both sets of predictable instructions overlap [17].

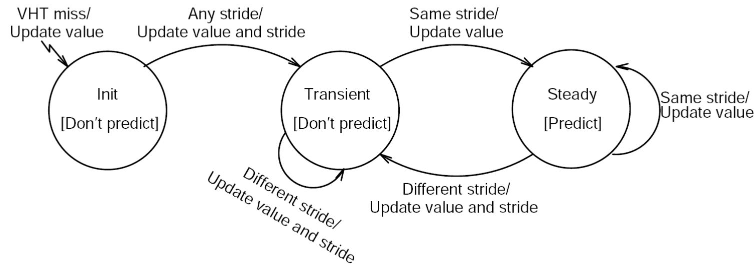
2-level + Stride

This hybrid is described in [26]. It consists of a 2-level predictor combined with a Stride predictor and it is shown to achieve higher accuracy than each predictor alone. The Stride predictor is a variation of the base Stride predictor in the sense that it uses 2 *State* bits which allow it to *not predict*. The automaton of this predictor is shown in Fig. 1.6(a).

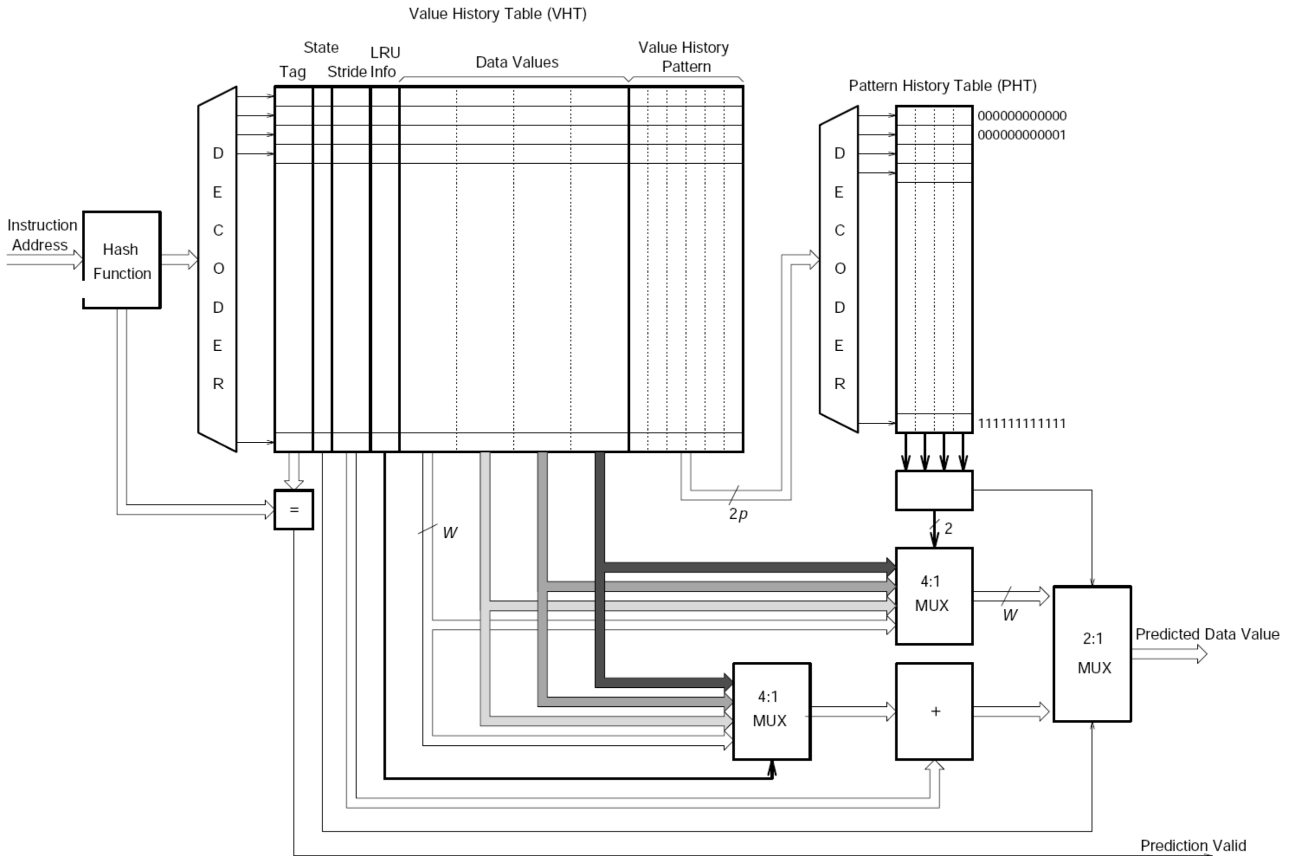
The prediction is selected as follow: if the 2-level predictor is confident enough (confidence is greater or equal to a given threshold), then use its prediction, if not and if Stride is in the *Steady* state, then use the prediction of the Stride predictor, otherwise or in case of a tag mismatch, do not predict. The 2-level + Stride predictor is shown in Fig. 1.6(b). It is also used by Calder, Reinman and Tullsen in [2] as well as Rychlik, Faistl, Krug and Shen in [15]. In the experiments reported in chapters 4, we used a threshold of 6 as it is the threshold used in [26]. Lastly, note that this hybrid suffers from the same implementation-related issues than the 2-level predictor (specifically, keeping local histories up to date).

Per-Path Stride Previous-Instruction (PS-PI)

[14] proposes to use the value produced by the last dynamic instruction executed in the program to help predict the result of the current instruction, and to combine a predictor using this method with the PS



(a) *Filtered* Stride predictor automaton.



(b) 2-level + Stride hybrid.

Figure 1.6: 2-level + Stride hybrid. The *State* field represents the state in which the automaton of the Stride component is in. From [26].

predictor. The resulting predictor is a hybrid consisting of a *Per-Path Stride* (PS) predictor and a *Previous-Instruction* (PI) predictor. Even though the PS predictor classifies as a FCM predictor, its core is still a Stride predictor, so in essence, the PS-PI predictor is a hybrid of a computational predictor (PS) and an order 1 "global" FCM (PI).

Selection is done by adding confidence counters in the Value Prediction History Table (VPHT) of the PS predictor so that the the prediction of PI is used when the counter in the PS predictor is low enough or simply when there is a miss in the PS predictor. Regardless of which component makes the prediction, confidence is

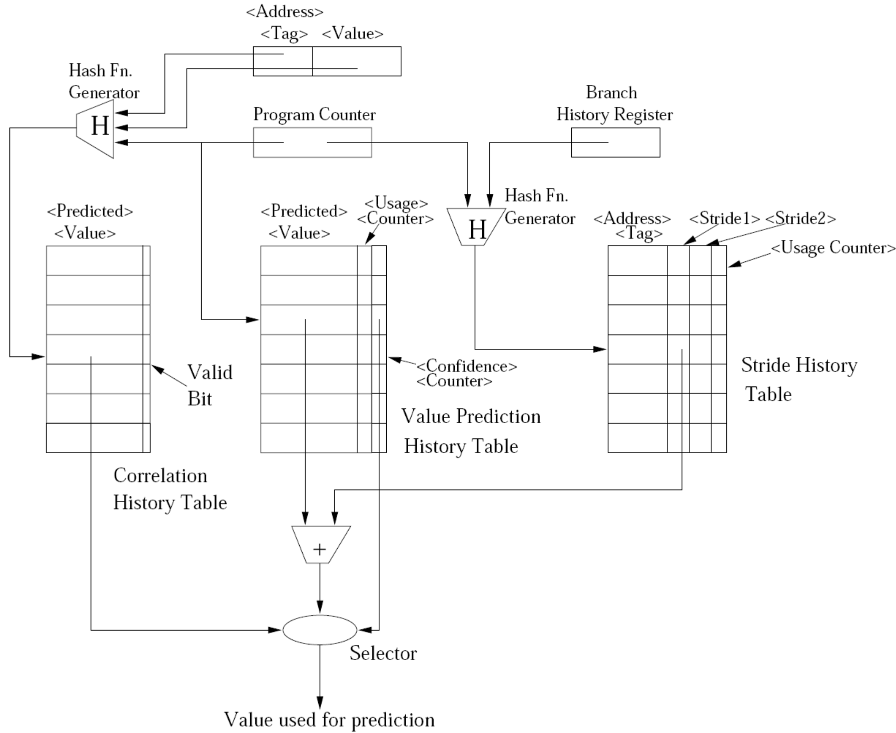


Figure 1.7: Per-Path Stride Previous-Instruction Hybrid Value Predictor. From [14].

incremented if PS is correct and decremented if PI is correct and PS hits. Since the Correlation History Table (CHT) of the PI predictor is not tagged, the PS-PI predictor will always make a prediction when the 2-level + Stride can *not* predict if it is not confident enough. This must be taken into account as the misprediction penalty might be high. The PS-PI predictor is described in Fig. 1.7. In the experiments reported in chapter 4, the threshold used to select the prediction was 3 (PS predicts if confidence is greater or equal to 3, otherwise PI predicts) and the hash function to access the PI is as follow: $index\ =\ PC\ xor\ last_PC\ xor\ last_value$.

1.3 Geometric Length Branch Predictors

1.3.1 TAGE and ITTAGE

Exploiting branch correlation - i.e. the correlation between the current branch and previous branches to guess the outcome of the current branch - has proven to be very efficient in the field of branch prediction. This is usually implemented by keeping a global (or local) branch history and hashing it in some manner with the current branch instruction address to form an index and access the predictor.

What can be observed is that most of the correlations are found between close branches (in terms of dynamic branch instructions executed), and few correlations are found between distant branches. However, to achieve high accuracy and coverage, capturing correlations between distant branches is necessary. Geometric Length predictors - in the form of the GEHL and TAGE predictors - were proposed by Seznec in [19, 20, 21] to fulfill this very requirement. The TAGE (**TA**gged **GE**ometric) predictor is described in Fig. 1.8(a).

The basic idea is to have several tables - *components* - storing predictions, each being indexed by a different number of bits of the global branch history hashed with the PC. The different lengths form a geometric series. This has the good property of dedicating most of the storage to short histories while still being able to capture correlations using very long histories (assuming the tables are similarly sized).

These tables are backed up by a base predictor which is accessed using only the PC. An entry of a tagged component consists of a partial tag, a usefulness counter u used by the replacement policy and a saturating signed counter *pred* giving the outcome of the prediction. An entry of the base predictor simply consists of a 2-bit saturating signed counter.

When a prediction is to be made, all components are accessed in parallel to check for a hit. The hitting component accessed with the longest history is called the *provider* component as it will provide the prediction. The component that would have provided the prediction if there had been a miss in the *provider* component is called the *alternate* component. In some cases, it is more interesting to have the *alternate* component make the prediction (e.g. when the *provider* entry was allocated recently). Such cases are monitored by a global counter so that if the entry in the *provider* is a new entry and the global counter is above 0, the *alternate* prediction is used. This global counter is decremented when *pred* is equal to *altpred*, and incremented otherwise. If no components hit, the base predictor provides the prediction.

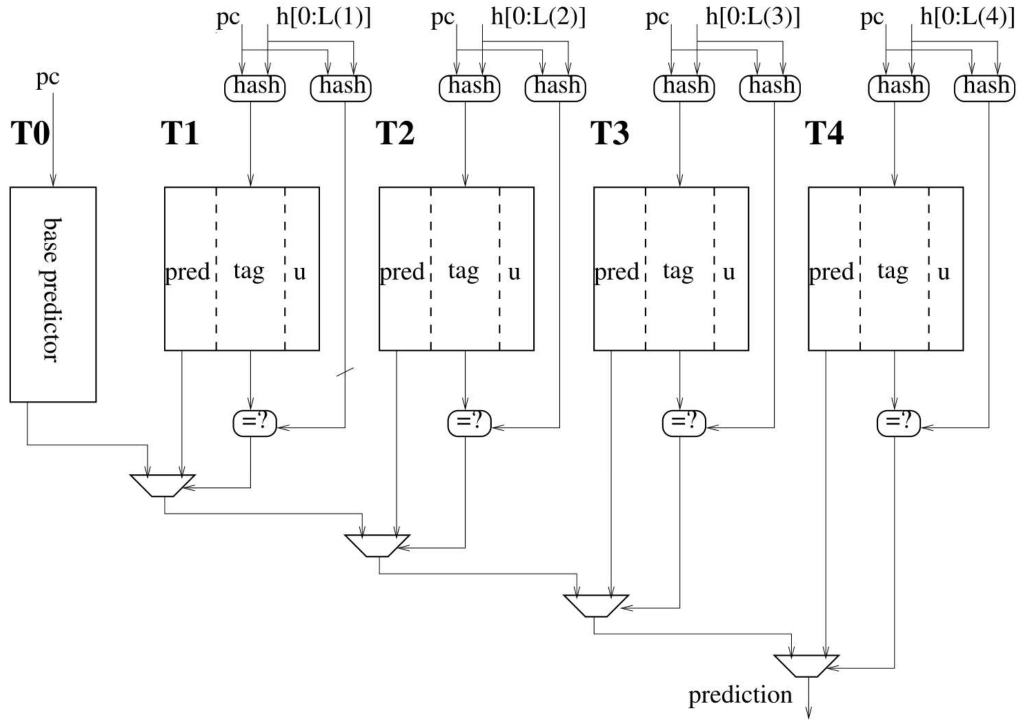
Once the prediction is made, only the *provider* is updated. On both a correct or an incorrect prediction, *pred*[1] and u[2] are updated. On a misprediction only, a new entry is allocated in a component using a longer history than the *provider*. First, all "upper" components are accessed to see whether one of them has an entry which is not useful (u is 0) or not. If not, the u counter of all matching entries are reset to 0 but no entry is allocated. Otherwise, a new entry is allocated in one of the components whose corresponding entry is not useful. The component is chosen randomly.

TAGE is well suited for branch prediction but also for indirect branch target prediction. The ITTAGE (**I**ndirect **T**arget TAGE) is a variation of the TAGE predictor which stores a full branch target *targ* as well as a hysteresis counter c in place of the counter used in TAGE to give the branch direction (*pred*). The entries are still partially tagged and the usefulness mechanism (deciding which entry to replace via the u field) is kept. It was introduced by Seznec and Michaud in [24] and further described in [22]. Fig. 1.8(b) shows an example of a 5-component ITTAGE predictor.

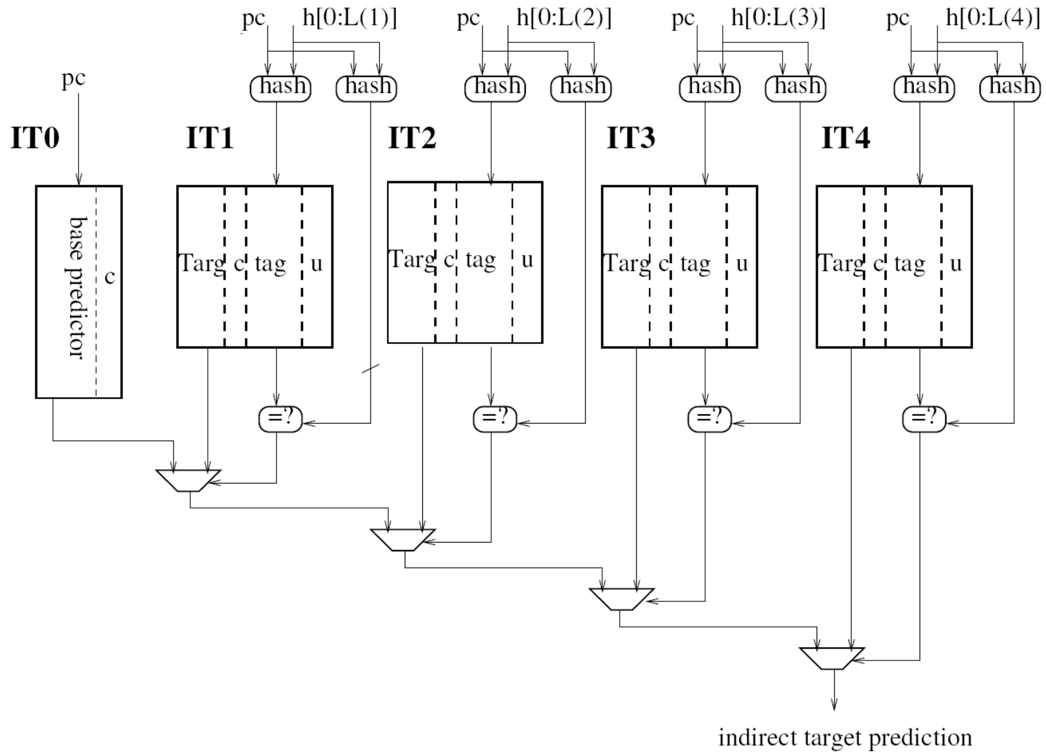
ITTAGE is essentially a "global" FCM branch predictor since it uses global information to select a prediction. This is a first similarity with context-based value predictors. The second similarity is the fact that ITTAGE is efficient to predict full addresses, that is, 32 or 64-bit values. Therefore, it is only natural to make the following hypothesis: If ITTAGE were to be adapted to predict values, it might be efficient and able to capture part of the predictions that a computational value predictor (such as Stride) cannot capture, as [18] states that both types of value predictors are - to some extent - complementary. Therefore, given a suitable selection mechanism, high accuracy could be obtained by combining a modified ITTAGE with a simple computational value predictor, providing us with an efficient and realistic way of predicting results.

[1] $pred = pred + 1$ if taken or $pred - 1$ if not taken

[2] $u = correct \,\&\&\, altpred \neq pred$ if the *provider* is not the base predictor.



(a) 5-component TAGE branch predictor. From [21].



(b) 5-component ITTAGE branch predictor. From [24].

Figure 1.8: TAGE and ITTAGE Geometric Length predictors.

Chapter 2

Validating the Choice of Geometric Length Value Predictors

Our first intuition is that ITTAGE can be modified to predict values and perform well, if not better than state-of-the-art. However, we need to assess that using a long branch history to correlate instruction results is efficient. To that extent, this chapter aims to study the similarities between the branch and path histories of two instances of the same static instruction producing the same result. If on average the histories show many similarities, modifying ITTAGE will be worthwhile.

2.1 Experimental Framework

What was studied is the context similarities between two instances of a single static instruction which produce the same result. That is, the similarities between the two global branch histories and the two path histories (consisting of the least significant bit of each jump target the control flow encountered).

We define the number of similar bits in two histories as the number of bits located at the same position in the two histories and having the same value. For instance, 10101010 and 10000110 have four similar bits (bits 0, 1, 6 and 7). We define the common subvector of two histories as the word or arbitrary length for which any bit i is equal to the ith bit in the first history and the ith bit in the second history. i goes from 0 (least significant bit) to j where j is the first bit in history_1 verifying history_1[j] != history_2[j]. For instance, the common subvector of 10101010 and 10000110 is 10 and its length is 2. For each benchmark, we give the average number of similar bits and the average length of the common subvector.

The benchmarks are fast forwarded until the $3B^{th}$ dynamic instruction is reached, then, the next million of register-producing instructions are studied. Even though one million dynamic instructions is a small number, the instrumented code is expected to be executed after the initialization phase of the benchmark and therefore to be representative of the computational part of the program. All SPECINT06 benchmarks [3] are used except *perlbench* and fed their respective reference workloads. Compilation is done with *gcc -O3 -m64* (Pointer width is 64 bits and dynamic linking is enabled). Instrumentation is done with PIN (x86_64 ISA) [10] on an Intel Nehalem.

2.2 Experimental Results

2.2.1 Global Branch History

Because of the accuracy results of the PS predictor [14], it comes as no surprise to see that the global branch histories are showing a common sub-vector in the hundred bits range, on average. In the same manner, the number of overall similar bits is quite high, as shown in Fig. 2.2.1.

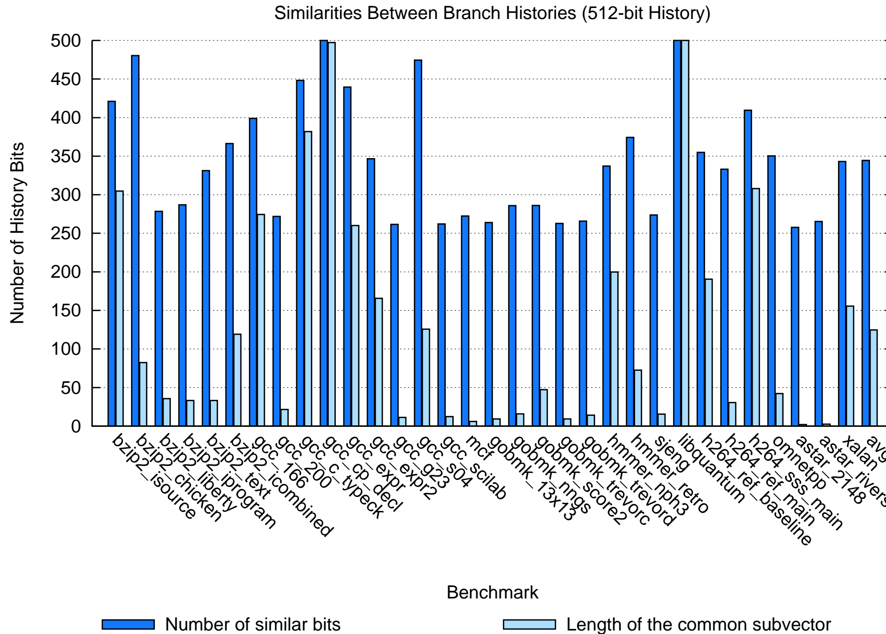


Figure 2.1: Similarities in the respective two 512-bit global branch histories of two instances of the same static instruction producing the same result.

2.2.2 Path History

Fig. 2.2.2 shows that path histories are also very similar: on average, with a 16-bit path history, three quarters of the two histories are similar and the length of the common sub-vector is two-third of the history length.

2.3 Conclusion

The main difference between path and global branch history is that for the former, the bars look the same for every benchmark whereas they are very different for the latter. This can be explained by the fact that only one bit per address is inserted in the path history, that is, all addresses whose LSB is 1 (resp. 0) are considered equal, leading to heavy aliasing.

The fact that global branch histories show a relatively good amount of similarities between instances producing the same result has one main consequence: if we limit ourselves to a global history length in the range of 100 to 150 bits, we can keep the indexing scheme of ITTAGE as is. The same can be said for the path history.

As a conclusion, we can say that the information used by the TAGE branch predictor (long branch history and a short path history) appear to be appropriate to correlate instructions results, even though a very large branch history will probably not be as efficient for Value Prediction as it is for Branch Prediction. Consequently, the maximum length of the branch history used in further experiments was limited to 128 bits (a value close to the average length of the common subvector observed for the studied benchmarks).

Figure 2.2: Similarities in the respective two 16-bit path histories of two instances of the same static instruction producing the same result.

Chapter 3

The Value TAgged GEometric Predictor

The previous chapter showed that the information used by the TAGE indexing scheme for Branch Prediction is also relevant for Value Prediction. This scheme is also used by the ITTAGE predictor whose role is to predict indirect jump addresses, which are not that different from register values. Therefore, the next step is to adapt the ITTAGE indirect branch target predictor to Value Prediction.

3.1 From ITTAGE to VTAGE

3.1.1 Predictor Structure

In fact, not much has to be changed to have ITTAGE predict values since the indexing mechanism can be kept as is. The *target* field has to be replaced by a *value* field which will contain a full (64 bits) register value instead of a branch target. The remaining variables are the number of tables, the partial tag width, the hysteresis counter width and the history update scheme. However, in the first version of our value predictor, all those variables were kept as in ITTAGE [22]. As a result, all configurations feature a single *useful* bit per entry in the tagged components and a 2-bit hysteresis counter *ctr* per entry in every component. The tag of tagged components is 7-bit long. Table 3.1 summarizes the different configurations we used. We refer to this predictor as the **V**alue TAGE (VTAGE) predictor.

Configuration	#Components	H_0	H_{max}	#Entries/component	Size
Small	1+7	1	64	2^{10}	$\simeq$ 598 Kbits
Quasi-Unlimited	1+7	1	64	$2^{pointer_size}$ (Base) | 2^{20} (Tagged)	-

Table 3.1: VTAGE Configurations Summary.

Because of physical memory limitations, we chose to limit the size of tagged components in the *Quasi-Unlimited* configuration to 2^{20} bits. Therefore, ideal performance will not be attained since two predictions may compete for an entry in a tagged component. However, the number of entries is such that we do not expect much aliasing.

3.1.2 Confidence Estimation

Prior to anything, we must determine how to decide if a prediction from VTAGE should proceed or not. In our case, we aim to maximize accuracy because mispredicting is costly [27]. As demonstrated in [23], it is possible to gauge the confidence of a prediction made by TAGE simply by observing the output, that

is, the value of the *pred* counter. In VTAGE, the *pred* counter is replaced by the *value* field containing the prediction. However, each VTAGE entry features a 2-bit hysteresis counter (*ctr*) whose behavior will strongly resemble that of the *pred* counter of TAGE. Intuitively, a high value for this counter means that the prediction was correct several times, so it has a high *confidence* and is essentially equivalent to having the *pred* counter saturated. Similarly, a low value for *ctr* means that the prediction was recently incorrect or that the entry has been newly allocated. In both cases, the prediction is not reliable, much like in the case of having the *pred* counter set to *weak taken/not taken*. Therefore, the hysteresis counters can be used to decide if VTAGE should predict or not.

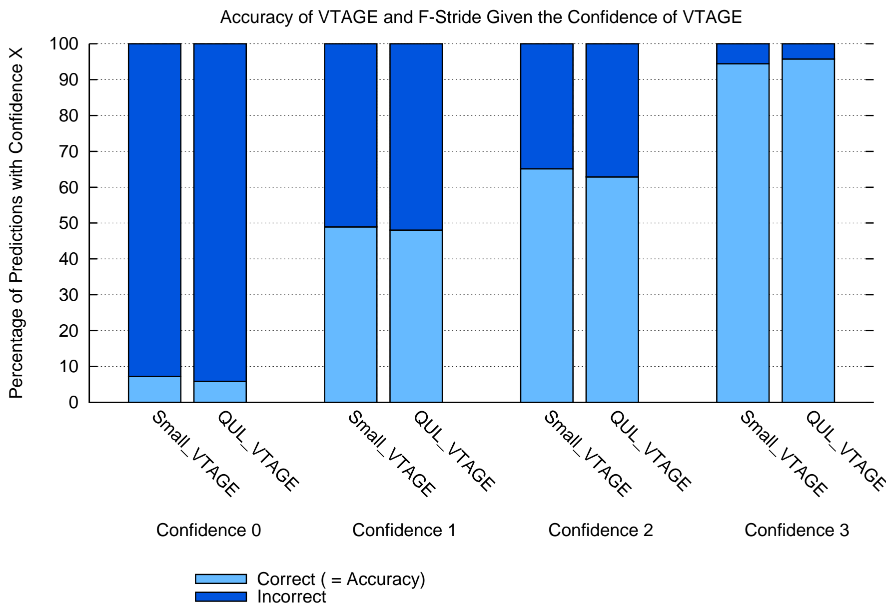


Figure 3.1: Average accuracy of VTAGE on its confidence. Results were gathered using the setup described in 4.2.1

Fig. 3.1 reports the accuracy of VTAGE for all possible confidence values. Results were collected on SPEC06INT benchmarks, and a detailed setup will be given later in 4.2.1. It appears that for all confidence values but 3, accuracy is not high enough to be interesting. Therefore, VTAGE should only be allowed to predict if the confidence associated with the prediction is 3.

3.2 Building a VTAGE-Based Hybrid Value Predictor

3.2.1 Computational Component

We chose the variation of the base Stride predictor used in the 2-level + Stride hybrid (see Fig. 1.6(a) in 1.2.3) as the computational predictor for our hybrid. We refer to it as the *Filtered* Stride predictor or F-Stride. This choice is mostly motivated by its reasonable coverage and its high accuracy (see Fig. 4.3 in 4.2.3 for detailed results) as well as its simplicity. The main difference is that we did not use tags for the *Filtered* Stride predictor as the filtering mechanism proved sufficient for our runs. We refer to the resulting hybrid predictor as the FS-VTAGE predictor.

Other candidates were the Last Value Predictor, which is in fact included in the Stride predictor, the 2-delta Stride Predictor, which is also a good candidate but is likely to be more complicated to update and lastly, the PS Predictor, even though it also qualifies as a FCM predictor. However, it appeared to us that PS would not bring much to VTAGE, because intuitively, the FCM behavior of PS is likely to be captured

by VTAGE while the computational behavior will be captured by Stride (or 2-delta). Therefore, we chose the simplest solution.

3.2.2 Predictor Configurations

A total of three configurations were used. For the first two configurations, VTAGE was combined with a similarly sized *Filtered* Stride predictor. In the third configuration (*ULS*), a *Small* TAGE predictor was combined to an "infinite" *Filtered* Stride predictor to check if even a small VTAGE can capture results that an "infinite" Stride predictor cannot. Table 3.2 summarizes the different characteristics of the predictors. Because we place ourselves in the context of future multicores featuring a lot of storage, the *Small* configuration is actually rather big by today's standards.

Hybrid	Entries (F-Stride + VTAGE)	Tag	Total Size (64-bit Stride)
Small	$2^{13} + (1024 \times 8\text{-components})$	7 bits	$\simeq 1,64$ Mbits
Quasi-Unlimited	$2^{pointer_size} + (2^{pointer_size} + 2^{20} \times 7\text{ T-components})$	7 bits	-
ULS	$2^{pointer_size} + (1024 \times 8\text{-components})$	7 bits	-

Table 3.2: FS-VTAGE Hybrid configurations summary.

On a related note and as stated in 1.2.1, the theoretical length of a stride is similar to that of a register, but in reality, an 8-bit stride is sufficient for 32-bit registers and a similar tradeoff can undoubtedly be made for 64-bit registers. However, further experiments with hybrid predictors use 64-bit strides so that complementarity between the two predictors is assured not be biased by having the *Filtered* Stride predictor mispredict due to the stride length.

3.2.3 Arbitrating Between the Two Components

To finalize the hybrid, we must decide which predictor should provide the result when both of them make a prediction, that is if the confidence associated with the prediction of VTAGE is 3 and the F-Stride predictor entry is in the *Steady* state. Fig. 3.2 suggests that in this case, it is slightly beneficial to favor VTAGE, whether in the finite configuration or the "infinite" one. As a consequence, our implementation always selects the value from VTAGE should the two predictors propose a prediction.

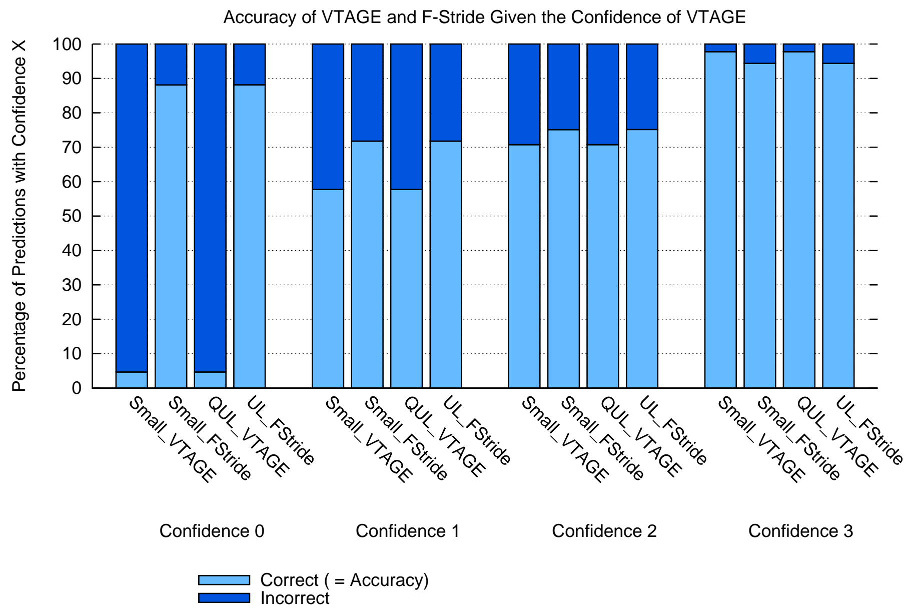


Figure 3.2: Average accuracy of VTAGE and F-Stride depending on VTAGE confidence when both predictors attempt to predict. Results were gathered using the setup described in 4.2.1

Chapter 4

Performance Evaluation of the FS-VTAGE Predictor

4.1 Objectives

In the last chapter, we presented the FS-VTAGE predictor. To demonstrate the qualities of this predictor, we will focus on two main objectives. First, we aim to assess the complementarity existing between the *Filtered* Stride and VTAGE predictors, that is the percentage of results that each predictor is able to capture and not the other. This metric will give us insight on the interest of putting a Stride-based predictor alongside a VTAGE predictor. Second, we will gauge coverage and accuracy. These quantities naturally depends on complementarity: if complementarity is non-existent, then the number of correct predictions will be at best that of the predictor with the highest coverage. However, if complementarity is non-negligible, these metrics will allow us to gauge the selection mechanism as well as simply compare this predictor with other hybrids. We will study the behavior of FS-VTAGE on integer and floating-point workloads separately.

Finally, since Value Prediction appeals for very high accuracy, our last goal will be to propose improvements to the FS-VTAGE predictor, should the baseline predictor not already be efficient enough.

4.2 Performance Evaluation on Integer Workloads

4.2.1 Experimental Framework

All benchmarks from SPECINT06 [3] were used and compiled with *gcc -O3 -m64*, except *perlbench* which was compiled with *gcc -O1 -m64*. Pointer width is 64 bits and dynamic linking is enabled. Instrumentation begins with the first instruction (no fast-forward) and lasts until two billion dynamic instructions have been instrumented. All benchmarks are fed their respective reference workloads. Instrumentation is done with PIN (x86_64 ISA) [10] on an Intel Nehalem. Fig. 4.1 details the percentage of executed instructions eligible for Value Prediction. VP-eligible instructions are all general-purpose integer register-producing instructions as well as non-store instructions whose destination is a memory location (e.g. *add mem1, mem2*). We leave memory operation address prediction and store value prediction for future work (see chapter 5).

Also, note that in modern x86 microprocessors, each instruction is broken down into one or more internal RISC-like μ-operations. For instance, instructions such as *add mem1, mem2* will in fact be broken down into two *load* instructions, one *add* instruction and one *store* instruction. Since we instrument at the instruction level, we will only try to predict the *add* instruction, and not the two *load* instructions. As a result, the numbers of Fig. 4.1 may be lower than what could be observed in related work using RISC ISAs. Finally, as only one thread is using the predictor, results are likely to be optimistic. In a real-world environment, context switching and *Simultaneous Multithreading* (SMT) may lower the numbers we obtained.

Percentage of Executed Instructions Used for Instrumentation

Percentage of Executed Instructions

Benchmark

Figure 4.1: Percentage of VP-eligible instructions per benchmark.

4.2.2 Complementarity

The two leftmost bars in Fig. 4.2 show that on average, 17% (resp. 22.5%) of the VP-eligible instructions are correctly predicted by *Small* VTAGE (resp. *Quasi-Unlimited* VTAGE) only, 20% (resp. 15%) of the VP-eligible instructions are correctly predicted by *Small* Stride (resp. *Unlimited* Stride) only, and another 27% (resp. 33%) of the VP-eligible instruction are correctly predicted by both. For *ULS*, the results are only slightly worse than for the *Small* configuration, further indicating that the two predictors indeed exhibit good complementarity.

These numbers demonstrate that complementarity exists and is non-negligible, even for very imbalanced predictor sizes. Therefore, assuming a reasonably good selection method, the FS-VTAGE hybrid will correctly predict more results than each predictor alone.

4.2.3 Accuracy and Coverage

To describe the following experiments, we define *Coverage*[1], which represents the total number of correctly predicted results out of the number of instrumented instructions, and *Accuracy*[2], which represents the chances for a prediction to be correct. For Value Prediction, accuracy is as important (if not more) as the global number of correct predictions because of the cost of a misprediction [27]. That is, depending on the misprediction penalty, it may be more interesting to have a predictor correctly predicting 99% (accuracy) of 50% (coverage) of the instrumented instructions than a predictor correctly predicting 75% of the instrumented instructions.

Distinct Predictors

As a first step, we evaluated the different predictors presented in section 1.2.1 and 1.2.2 on the same benchmarks as the ones we used to evaluate VTAGE. Table 4.1 and Table 4.2 respectively summarize the characteristics of the finite and "infinite" configurations used for all non-hybrid predictors.

[1] $(Correct/Instrumented) \times 100$

[2] $(Correct/(Correct + Incorrect)) \times 100$

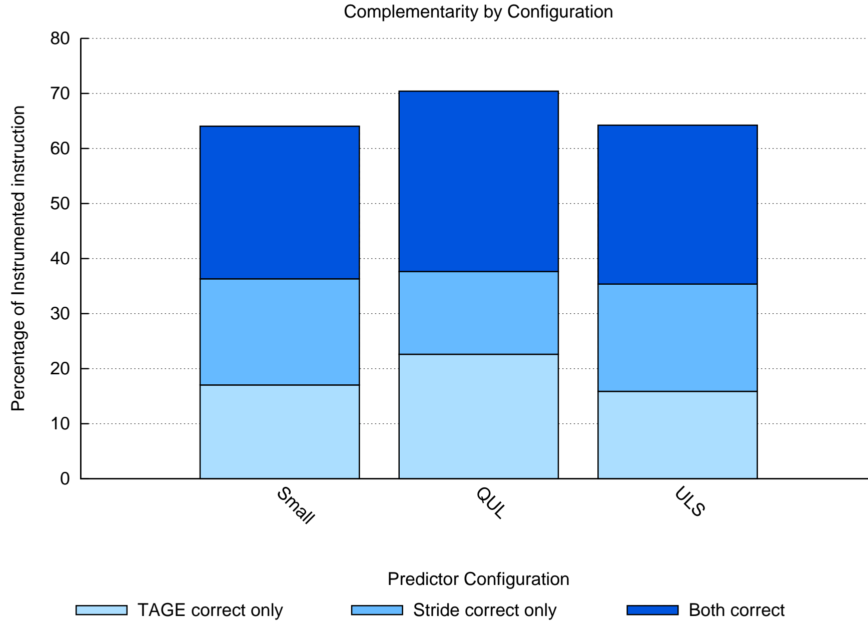


Figure 4.2: Complementarity by Predictor Configuration.

Predictor	#Entries	Tag	Size
Small F-Stride	2^{13}	None	$\simeq$ 1.05 Mbits
Small F-Stride w/ TAG	2^{13}	Full (51 bits)	$\simeq$ 1.47 Mbits
Small 2-delta	2^{13}	None	$\simeq$ 1.57 Mbits
Small 2-delta w/ TAG	2^{13}	Full (51 bits)	$\simeq$ 1.99 Mbits
Small PS_64b	$2^{10} \times$ 4-way	Full (54 bits)	$\simeq$ 1.02 Mbits
Big PS_64b	$2^{11} \times$ 4-way	Full (53 bits)	$\simeq$ 2.03 Mbits
Small PS_8b	$2^{10} \times$ 4-way	Full (54 bits)	$\simeq$ 538 Kbits
Small 2-level	2^{11} (VHT) + 2^{12} (PHT)	Full (53 bits)	$\simeq$ 722 Kbits
Small VTAGE	$2^{10} \times$ 8-components	7 bits (7 T-components)	$\simeq$ 532 Kbits

Table 4.1: Finitely-sized configurations summary. Top: computational predictors, bottom: context-based predictors. If not precised, the stride length is 64 bits.

Predictor	#Entries	Tag
UL F-Stride	$2^{pointer_size}$	None
UL 2-delta	$2^{pointer_size}$	None
UL PS	$2^{pointer_size}$	Branch history depth (2 bits)
UL 2-level	$2^{pointer_size}$ (VHT) $+2^{12}$ (PHT)	None
UL 2-level+	$2^{pointer_size}$ (VHT) $+2^{24}$ (PHT)	None
Quasi-UL VTAGE	$2^{pointer_size} + 2^{20} \times$ 7 T-components	7 bits (7 T-components)

Table 4.2: “Infinitely”-sized configurations summary. Top: computational predictors, bottom: context-based predictors. UL stands for **U**n**L**imited.

One could argue that the size of the respective finitely-sized predictors vary from 532Kbits to 1.99Mbits, rendering the comparison irrelevant. However, what interests us most is to compare predictors belonging to

the same category (computational and context-based). From this angle, the range of sizes is much narrower. Moreover, because the PS predictor can be viewed as either a computational or context-based predictor, we study three finite configurations. The two first (*Small_64b* and *Big_64b*) feature 64-bit strides so that they can be compared to computational predictors. The third one (*Small_8b*) features 8-bit strides to reduce its size in order to be compared to context-based predictors. Furthermore, by comparing results of the *Small_8b* and *Small_64b* configurations, we will be able to gauge the impact of the stride size on accuracy and coverage. Lastly, as the "infinite" configurations aim to gather optimal results, they all make use of 64-bit strides. Fig. 4.3 summarizes the number of correctly, incorrectly and not predicted instructions. Table 4.3 summarizes accuracy and coverage.

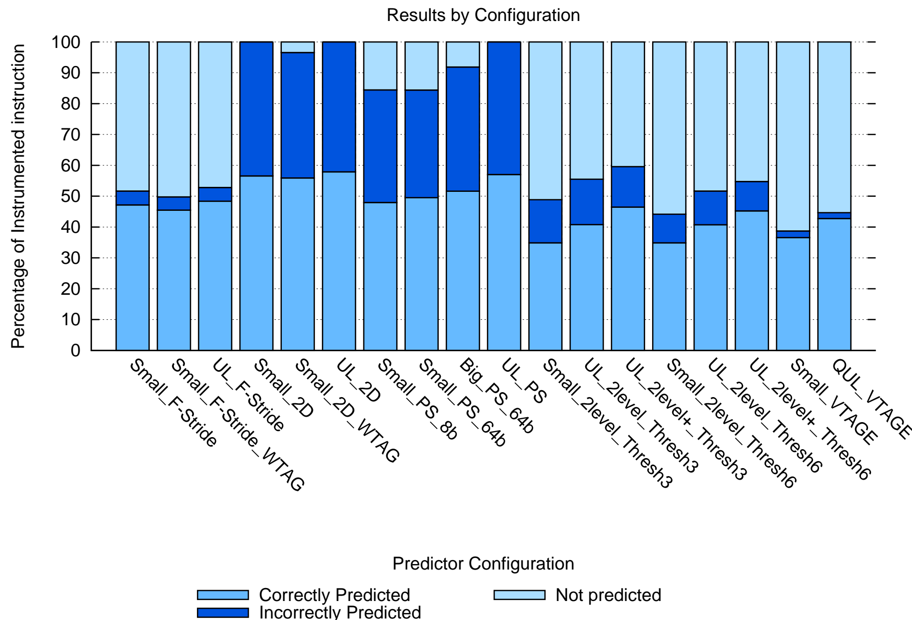


Figure 4.3: Results for different value predictors.

Predictor	Accuracy	Coverage
Small F-Stride	91.2%	47.1%
Small F-Stride w/ TAG	**91.3%**	45.4%
Small 2-delta	57.9%	**57.6%**
Small 2-delta w/ TAG	60.5%	55.2%
Small PS_64b	58.6%	49.5%
Big PS_64b	56.2%	51.6%
Small PS_8b	56.8%	**48.0%**
Small 2-level (Thresh 3)	71.3%	34.8%
Small 2-level (Thresh 6)	79.0%	34.9%
Small VTAGE	**94.4%**	36.8%

Predictor	Accuracy	Coverage
UL F-Stride	**91.5%**	48.3%
UL 2-delta	60.4%	**58.7%**
UL PS	57%	**57.1%**
UL 2-level (Thresh 3)	73.5%	40.8%
UL 2-level (Thresh 6)	78.8%	40.7%
UL 2-level+ (Thresh 3)	78.0%	46.5%
UL 2-level+ (Thresh 6)	82.6%	45.2%
Quasi-Unlimited VTAGE	**95.7%**	42.8%

Table 4.3: Accuracy and coverage summary for distinct value predictors. Left: finite configurations, right: "infinite" configurations. Top: computational predictors, bottom: context-based predictors.

Before analyzing the results, let us focus on the stride sizes: the difference between 8-bit strides (*Small* PS_8b) and 64-bit strides (*Small* PS_64b) is a coverage of 1.8% more of the instrumented instructions in favor of *Small* PS_64b and an accuracy which increases from 48.0% to 49.5%. Therefore, if not optimal,

using 8-bit strides is bearable, even for 64-bit registers.

For computational predictors, we can observe that the *Filtered* Stride predictor is the most interesting computational predictor of our set mostly due to its very high accuracy. 2-delta is not accurate enough although it could be augmented with confidence counters and behave similarly to *Filtered* Stride. Higher accuracy is however not guaranteed. The "infinite" configurations exhibit similar results, hinting that most static instructions eligible for Value Prediction can fit in an 8K-entry predictor for the majority of the benchmarks. The PS predictor is much less accurate and only covers slightly more instructions, rendering it uninteresting as a computational predictor. Interestingly, PS has less coverage than 2-delta, whereas the only difference between the two predictors is the indexing function. This essentially means that either branch history is not relevant enough to correlate results, which is in contradiction with 2.2.1 or that the hashing function used in [14] is not efficient enough.

Regarding context-based predictors, the 2-level predictor has the lowest coverage and a much lower accuracy than VTAGE. However, the 2-level+ shows better accuracy and coverage, hinting that a 4096-entry PHT is suffering from aliasing. However, the cost of expanding it seems prohibitive: as said in 1.2.2, increasing the value history from 4-deep to 8-deep results in a PHT size of roughly 537 Mbits (vs. 65 Kbits) if only the last 8 outcomes are recorded. Furthermore, even though PS has better coverage (around 10% of the instrumented instructions more, 15% in the "infinite" case), VTAGE is more interesting because it has a much higher accuracy. One could argue that adding confidence counters to the baseline PS predictor would yield a better accuracy, and this is most likely to be true. However, in the end, the strongest argument of VTAGE is that it is not a stride-based predictor, meaning that it can be combined with a stride-based predictor because both are complementary enough, as demonstrated in 4.2.2. On the contrary, combining PS with another stride-based predictor should not yield tremendous results, to say the least.

The results indeed advocate for a combination of a VTAGE predictor and a *Filtered* Stride predictor to attain high accuracy and moderate coverage. Considering the number of Table 4.3 and Fig. 4.2, we can expect an accuracy greater than 90% with a coverage greater than 50%. These numbers are quite interesting since [2] claims that the full potential of Value Prediction can be obtained by - correctly - predicting only 60% of the VP-eligible instructions. Those instructions are in fact the instructions on the internal *critical-path* of the processor.

Hybrid Predictors

In a second step, we evaluated the hybrid predictors presented in 1.2.3 on the same benchmarks as the ones we used to evaluate FS-VTAGE. Table 4.4 and Table 4.5 respectively summarize the characteristics of the finite and "infinite" configurations used for all hybrid predictors. Fig. 4.4 summarizes the number of correctly, incorrectly and not predicted instructions. Table 4.6 summarizes accuracy and coverage. Note that in this case, the *Small* configurations of 2-level + Stride, PS-PI and FS-VTAGE all feature the approximate same amount of storage and make use of 64-bit strides.

Hybrid	#Entries	Tag	Size
Small 2-level + Stride	2^{12} (VHT) + 2^{12} (PHT)	Full (52 bits)	$\simeq$ 1.68 Mbits
Small PS-PI	$2^{10} \times$ 4-way (PS) + 2^{12} (PI)	Full (54 bits, PS)	$\simeq$ 1.55 Mbits
Small FS-VTAGE	$2^{13} + (2^{10} \times$ 8-components)	7 bits (7 T-cpts, VTAGE)	$\simeq$ 1.64 Mbits

Table 4.4: Finitely-sized hybrid configurations summary. T-cpts stands for *tagged* components.

Hybrid	#Entries	Tag
UL 2-level + Stride	$2^{pointer_size}$ (VHT) + 2^{12} (PHT)	None
UL 2-level+ + Stride	$2^{pointer_size}$ (VHT) + 2^{24} (PHT)	None
UL PS-PI	$2^{pointer_size} \times 2$ (VHT+SHT) + $2^{pointer_size}$ (PI)	bhist depth (2 bits, PS)
QUL FS-VTAGE	$2^{pointer_size} + (2^{pointer_size} + 2^{20} \times 7$ T-compts)	7 bits (7 T-cpts, VTAGE)

Table 4.5: "Infinitely"-sized hybrid configurations summary.

The coverage of the FS-VTAGE hybrid is indeed greater than 50% at 56.9% (*Small*) and 61.2% (*Quasi-Unlimited*). It is actually slightly higher than that of the 2-level + Stride hybrid (55.6% and 59.8%) and equal to that of the PS-PI for the finite case (56.9%). However, it is lower than that of the PS-PI (67.7%) and 2-level+ + Stride (63.7%) for the "infinite" case.

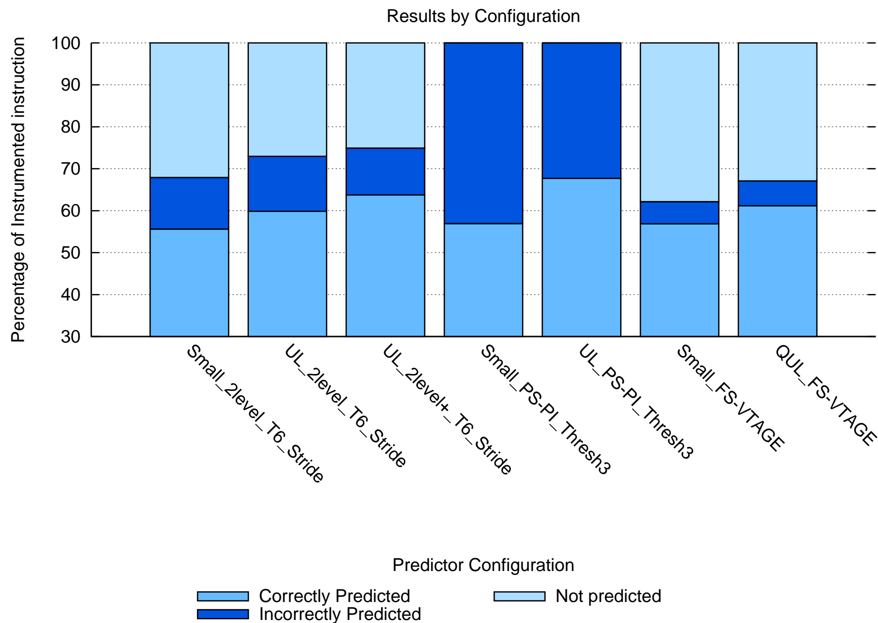


Figure 4.4: Results for different hybrid value predictors.

Predictor	Accuracy	Coverage
Small 2-level + Stride	81.9%	55.6%
Small PS-PI	56.9%	**56.9%**
Small FS-VTAGE	**91.5%**	56.9%

Predictor	Accuracy	Coverage
UL 2-level + Stride	82.0%	59.8%
UL 2-level+ + Stride	85.0%	63.7%
UL PS-PI	67.7%	**67.7%**
QUL FS-VTAGE	**91.2%**	61.2%

Table 4.6: Accuracy and coverage summary for hybrid value predictors. Left: finite configurations, right: "infinite" configurations.

Accuracy is high at 91.5% (*Small*) and 91.2% (*Unlimited*) which is much higher than that of other predictors: The second most accurate predictor accuracy score is 9.6 points lower in the finite case (2-level + Stride) and 6.2 points lower in the "infinite" case (2-level+ + Stride).

We can also observe that the accuracy difference between the *Small* and *Quasi-Unlimited* configurations is very small (around 0.3 point in favor of the smallest). However, the difference in coverage is non-negligible (around 4% of the instrumented instructions in favor of the biggest). Therefore, even though FS-VTAGE is

very accurate whatever its size, its coverage will benefit from increasing the components size.

To conclude, we indeed obtain an accuracy higher than 90%, which is better than that of any other hybrid considered, and a coverage higher than 50%, which is reasonable even though the PS-PI predictor covers slightly more results. Consequently, we claim that the FS-VTAGE predictor is the most interesting value predictor so far.

4.3 Performance Evaluation on Floating-Point and Vector Workloads

Floating point (FP) [1] instructions also exhibit *Value Locality* [12]. However, locality is somehow less prevalent than in integer workloads. Therefore, worse results than those observed for integer benchmarks are expected [5]. We chose to make the distinction between integer and FP results for the following reason: because of the way PIN instruments the code and the way the x87 ISA works, getting the result of x87 FP operations has proven very difficult and error-prone. Therefore, we chose to use the *Streaming SIMD Extensions* (SSE) ISA[3] as the FPU because first, it is easier to manipulate with PIN, and second, x87 is generally less efficient and SSE should be used when possible [11]. Interestingly, SSE features both scalar and vector instructions. Vector (Single Instruction Multiple Data or SIMD) instructions operate on several variables packed (concatenated) in a special purpose register (64-bit *mm* or 128-bit *xmm* for SSE). For instance, one 128-bit vector multiplication can multiply four single precision FP variables with four other FP variables, increasing throughput by the same amount.

Vectorized code is much more efficient than scalar code, and should always be used if possible. However, in the event of Value Prediction being only capable of predicting the value of general purpose registers, the programmer (or the compiler) will have to choose between the two features (VP or SIMD) if the code is vectorizable. Yet, Value Prediction may benefit to SIMD instructions.

In general, both SSE scalar and SSE vector instructions are present in the generated code. Therefore, we chose to use this opportunity as a way to verify the *Value Locality* of FP instructions as well as to assess the potential of Value Prediction for vector instructions.

4.3.1 Experimental framework

All floating-point benchmarks of the SPECFP06 suite [3] were used and compiled on an Intel Nehalem with *gcc -O3 -mfpmath=sse -msee4.1 -march=native -mno-mmx -m64* in order to force the use of SSE instead of x87 for FP instructions. Pointer width is 64 bits and dynamic linking is enabled. The benchmarks are fed their respective reference workloads. The first 10 billion instructions are skipped and the benchmarks are then run until 10 billion instructions have been instrumented as longer runs proved necessary to obtain relevant results for FP workloads. Instrumentation is done with PIN (x86_64 ISA) [10] on an Intel Nehalem. Fig. 4.5 summarizes the percentage of executed instructions that were instrumented for all benchmarks. VP-eligible instructions are all special-purpose *xmm* register-producing instructions.

In addition to using scalar SSE instructions for FP arithmetic, compiling with full optimizations (-*O3*) allows gcc to automatically vectorize code, that is factorize computations by replacing several scalar instructions by a single vector instruction when possible. Furthermore, SSE is able to operate on both integer and floating-point variables. Therefore, we define four categories of instructions among the instrumented instructions : *FP_Scalar*, *FP_Vect*, *INT_Scalar* and *INT_Vect*. However, since no instructions belonging to the *INT_Scalar* category were generated when compiling the considered benchmarks, the *INT_Scalar* category is ignored in the following sections. Fig. 4.6 summarizes the proportion of instrumented instructions each category represents. Note that the number of instructions belonging to the *INT_Vect* category is negligible for all benchmarks except *milc* and *soplex*.

[3] x86 features two FPU ISA: the older x87 and the newer SSE. While x87 is deprecated it is still used by default in GCC.

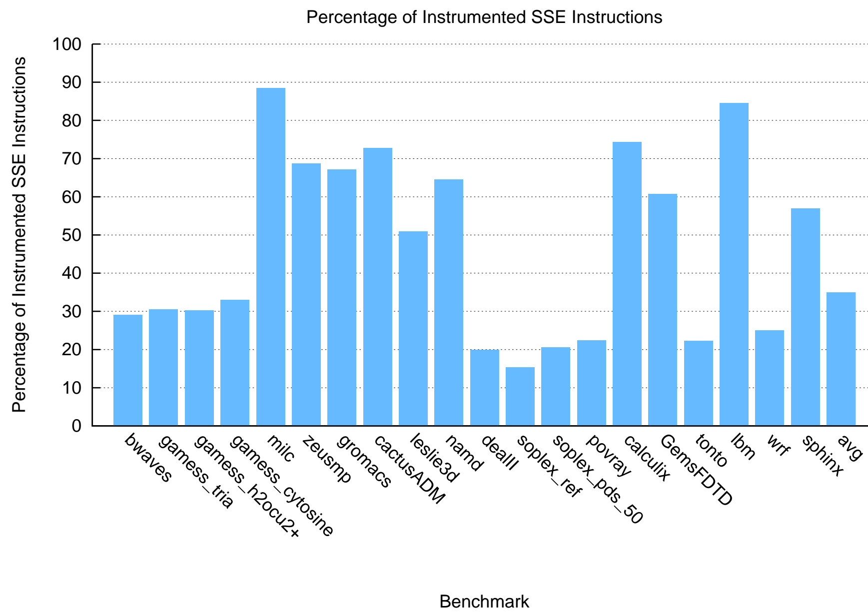


Figure 4.5: Percentage of VP-eligible SSE instructions per benchmark.

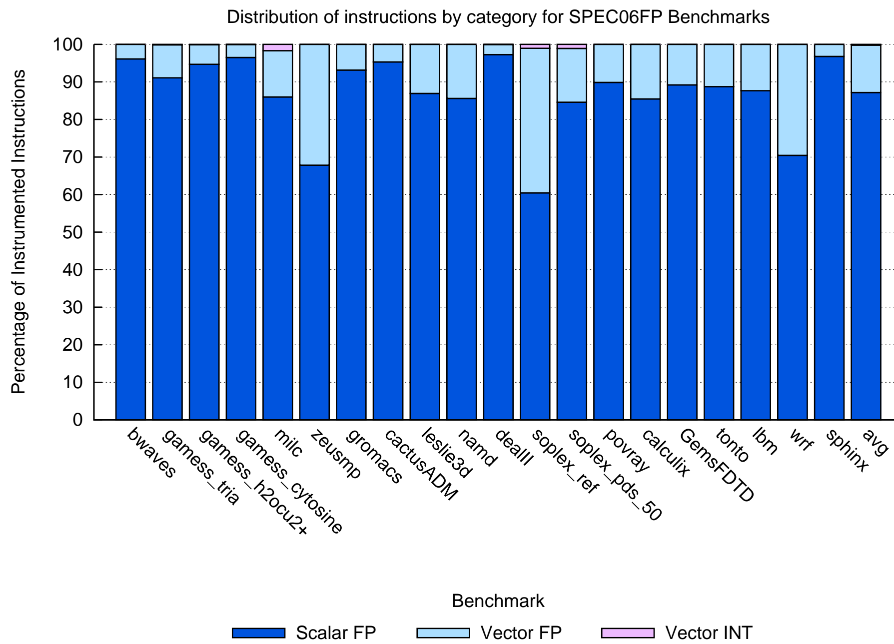


Figure 4.6: Proportion of *FP_Scalar*, *FP_Vect* and *INT_Vect* in different benchmarks.

In order to predict the results of up to 128-bit wide vector instructions, all predictors were augmented with 128-bit entries. For VTAGE and 2-level, this is straightforward since only the prediction needs to be stored. For stride-based predictors, each 128-bit entry actually contains 4 32-bit values and 4 32-bit strides. The tradeoff of using a smaller stride to save storage can still be made although several strides must be stored (one 8-bit stride per 32-bit variable of a 128-bit register for instance). To generate a prediction, each stride is added to the corresponding value independently from the others, meaning that overflowing is not propagated in any way. The four values are then concatenated to form the prediction. On an update, all strides and values are updated. In this section, we only consider hybrid predictors by lack of time.

Table 4.7 summarizes the different hybrid configurations. Due to the fact that much longer runs were needed to acquire relevant results, the "infinite" configuration of PI is less idealistic because of physical memory limitations. For predictors using PC-only indexing such as Stride, 2-level and PS (PC hashed to 2 bits), the *Unlimited* configuration is similar to that used for integer benchmarks. It should be noted that the *Small* FS-VTAGE predictor is 9,5% bigger than the *Small* 2-level + Stride. Even though this is not a tremendous difference, this is non-negligible and we will consider it when analyzing the results.

Hybrid	Entries	Tag	Size
Small 2-level + Str	4096 (VHT) + 4096 (PHT)	52 bits	$\simeq 2,95$ Mbits
UL 2-level + Str	2^{ptr_size} (VHT) + 4096 (PHT)	None	-
Small PS-PI	$1024 \times$ 4-way (PS) + 8192 (PI)	54 bits (PS)	$\simeq 3,12$ Mbits
Quasi-UL PS-PI	$2^{ptr_size} \times 2$ (VHT+SHT) + 2^{24} (PI)	2 bits (PS)	-
Small FS-VTAGE	2^{13} + ($1024 \times$ 8-components)	7 bits (VTAGE)	$\simeq 3,23$ Mbits
Quasi-UL FS-VTAGE	2^{ptr_size} + ($2^{ptr_size} + 2^{20} \times$ 7-components)	7 bits (VTAGE)	-

Table 4.7: Hybrid configurations summary.

4.3.2 Complementarity

Fig. 4.7 illustrates the complementarity of the components of F-Stride + VTAGE for floating-point and vector workloads.

What can be observed is that for this particular sample, the number of VTAGE-only predictable results (scalar or vector) is around 5 times bigger than the number of Stride-only predictable results, depending on the configuration. In fact, the VTAGE component is able to capture most of what the Stride component can predict since the number of results predictable by both components is quite high: between 40% and 45% of the instrumented instructions. The difference between these numbers and those found for integer workloads is probably due to the fact that adding a stride to a floating-point value as a whole does not make much sense since a FP number is made of several fields.

4.3.3 Accuracy and Coverage

Fig. 4.8 and 4.9 respectively summarize the number of correctly, incorrectly and not predicted instructions for all instrumented instructions and per-category. Table 4.6 summarizes accuracy and coverage. If we compare the results shown in Fig. 4.8 to integer results, accuracy is either higher for 2-level + Stride (89.3-89.5% vs. 81.8-82.0%), lower for PSPI (39.5% vs. 56.9%) and roughly similar for FS-VTAGE (90.2-89.9% vs. 87.1-90.0%). However, coverage is generally much lower at 41.1-42.2%, 39.5% and 40.7-41.9% versus 55.6-59.8%, 56.9% and 60.5-67.0% for 2-level + Stride, PSPI and FS-VTAGE respectively. In essence, all three hybrids globally cover roughly the same number of instructions, but FS-VTAGE is more accurate.

Since the number of integer SSE instructions is overall negligible, these results verify that even though predictability is indeed lower than for integer benchmarks, the predictability of floating-point instructions is still quite interesting.

Fig. 4.9 deals with coverage and accuracy for the different categories introduced in the beginning of the chapter. As seen in previous figures, the size of the hybrid does not influence much coverage and accuracy.

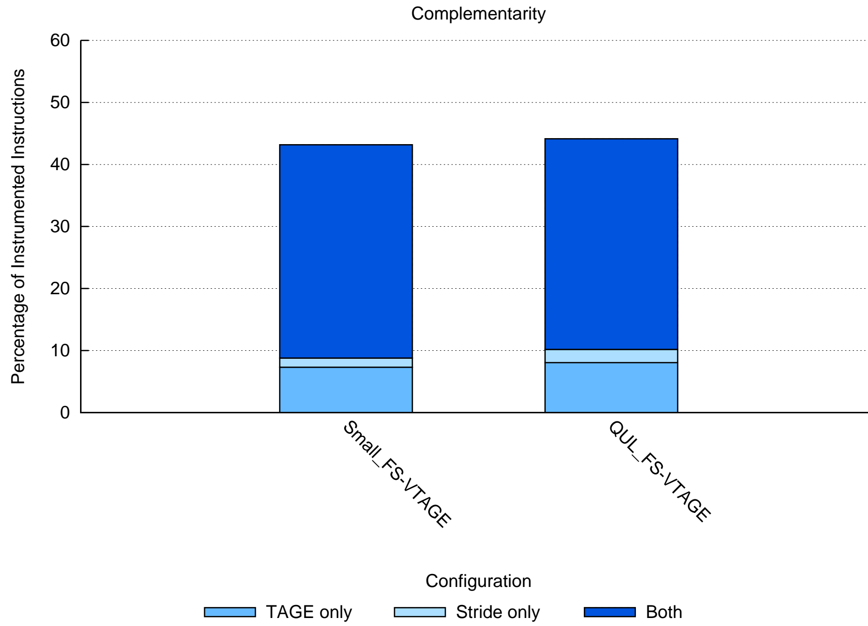


Figure 4.7: Complementarity of different FS-VTAGE configurations for SPEC06FP.

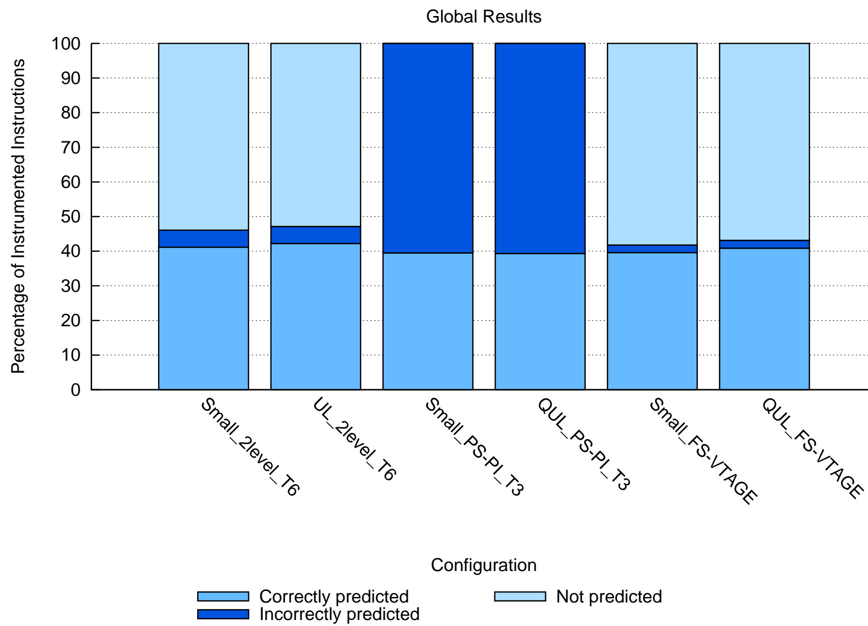


Figure 4.8: Proportion of overall correct/incorrect predictions.

An interesting fact, however, is that even if accuracy is essentially similar between scalar and vector instructions (slightly higher for vector instructions), coverage for scalar FP is around 40% while it is around 60% for vector FP for 2-level + Stride, around 55% for PS-PI and around 50% for FS-VTAGE. A similar behavior can be observed for integer instructions. These numbers hint that both integer and FP instructions exhibit more *value locality* than their scalar counterparts. However, note that the *Quasi-Unlimited* PS-PI predictor

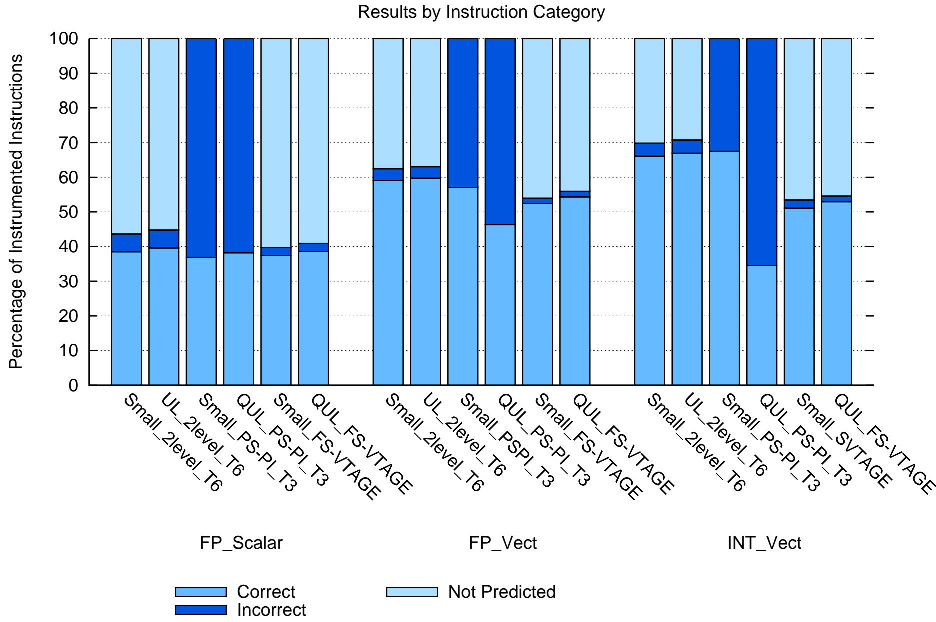


Figure 4.9: Proportion of correct/incorrect predictions depending on their category.

	Global		FP_Scalar		FP_Vect		INT_Vect	
Predictor	Acc.	Cov.	Acc.	Cov.	Acc.	Cov.	Acc.	Cov.
Small 2-level + Str	89.3%	**41.1%**	88.2%	**38.5%**	94.5%	**59.0%**	94.7%	66.1%
Small PS-PI	39.5%	39.5%	36.9%	36.9%	57.0%	57.0%	67.5%	**67.5%**
Small FS-VTAGE	**94.7%**	39.6%	**94.1%**	37.4%	**97.2%**	52.4%	**95.5%**	51.1%
UL 2-level + Str	89.5%	**42.2%**	88.4%	**39.6%**	94.7%	**59.7%**	94.6%	**67.0%**
Quasi-UL PS-PI	39.4%	39.4%	38.2%	38.2%	46.3%	46.3%	34.5%	34.5%
Quasi-UL FS-VTAGE	**94.7%**	40.9%	**94.2%**	38.6%	**97.2%**	54.3%	**97.0%**	52.4%

Table 4.8: Accuracy and coverage summary for hybrid value predictors. Top: finite configurations, bottom: "infinite" configurations.

performs rather poorly on vector instructions. No explanation immediately comes to mind to explain this behavior, except maybe an error in our implementation (even though there is no issue for integer and scalar FP).

As a conclusion, we can state that first, the predictability of floating-point instructions is real but it is lower than that of integer instructions, and second, vector instructions appear more predictable than their scalar equivalents, even though they often represent less instructions in total. Therefore, Value Prediction does not need to be limited to general-purpose integer workloads as it appears able to accelerate every type of computation. Also, the addition of a value predictor specialized in floating-point instructions such as the SEF predictor [5] (see 1.2.1) may yield better results, even if in our opinion, it requires some modifications to be worth the investment. As a remark, using another compiler (such as Intel *icc*) may generate executables containing more vectorized code. Consequently, one could demonstrate if the predictability of vector instructions is general to all of them or only to a subset.

4.4 Improving the FS-VTAGE Performance

In the previous section, we saw that the FS-VTAGE predictor is very accurate and covers a reasonable amount of VP-eligible instructions. However, Zhou, Fu, Rotenberg and Conte claim that the speedup obtainable with Value Prediction tremendously decreases if the accuracy of the predictor drops from 99% to 90% [27]. Since the accuracy of FS-VTAGE is around 90%, there is clearly room for improvement. To a lesser extent, we also consider increasing coverage.

4.4.1 Augmenting The VTAGE Component with *Per-Instruction* Confidence

Confidence Distribution for VTAGE Predictions

Regardless of the prediction selection mechanism, what can be observed in Fig. 4.10 is that for VTAGE, the majority of the wrong predictions are associated with confidence 0, and the majority of the correct predictions are associated with confidence 3 in all configurations. However, nearly 10% of correct predictions are made with a confidence strictly lesser than 3. If such results are also correctly predicted by F-Stride, then this is not an issue. Yet, it is not the case.

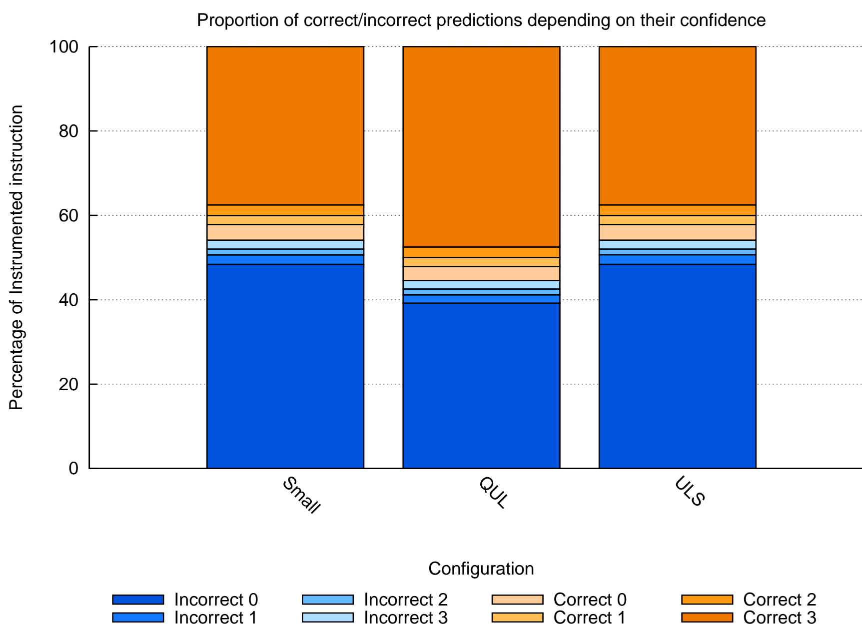


Figure 4.10: Proportion of correct/incorrect VTAGE predictions depending on their confidence.

As said in 4.2.2, 17 to 22% of the results are only predictable by VTAGE. If we consider the confidence distribution for such predictions (given by bars 2 and 5 in Fig. 4.11), we can observe that between 35% (*Unlimited*) and 40% (*Small*) of the said predictions are associated with a confidence value strictly lesser than 3. In other words, these predictions are not taken into account while they represent a non-negligible proportion of the correct predictions.

Similarly, if we look at bars 1 and 4, we can see that in the *Small* configuration, 1 to 2% of F-Stride-only correct predictions will effectively be provided by VTAGE as their confidence is 3. This number goes up to around 3% for the *Unlimited* configuration.

Therefore, we claim that even though using VTAGE hysteresis counters as confidence counters is simple and cheap, it is not optimal as some correct predictions that F-Stride is not capable to capture are discarded

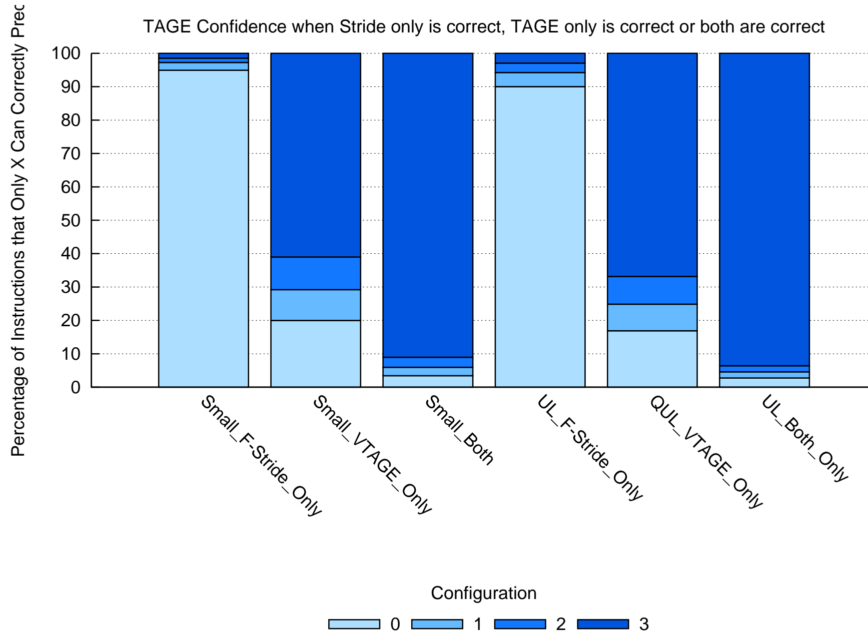


Figure 4.11: Confidence of TAGE predictions for results that only TAGE, *Filtered* Stride or both are able to predict.

and VTAGE is allowed to predict when only the Stride predictor is able to correctly predict. This directly impact coverage and accuracy.

Increasing Performance by Adding a Second Level of Confidence

There is potential for correctly predicting more results by improving the VTAGE confidence mechanism. On the one hand, one could try increase coverage by promoting some entries whose confidence is strictly lesser than 3 but happen to contain a correct prediction. On the other hand, one could try to increase accuracy by further filtering predictions whose confidence is 3. Both possibilities use the same principle of a second level of confidence. The base idea is to use *per-instruction* confidence counters (i.e. associated with static instructions) to filter the *per-prediction* confidence (hysteresis counters *ctr*, associated with dynamic instructions). One could even mix the two possibilities to obtain the best tradeoff between coverage and accuracy. This mechanism is based on the following assumption: if the current prediction comes from an entry whose confidence is low but the last few results of the corresponding static instruction have been correctly predicted, then the prediction is more likely to be correct.

Essentially, we added a 4K-entries, direct-mapped, fully-tagged table containing 3-bit saturating counters to the *Small* VTAGE predictor (resulting in a predictor size of around 831.5 KBits). The counter saturates at 7, the misprediction penalty[4] varies from 1 to 7 and the second level threshold used to decide if the prediction should proceed given the value of its *ctr* confidence counter also varies from 1 to 7.

Fig. 4.12 summarizes the number of correctly, incorrectly and not predicted instructions. Table 4.9 summarizes accuracy and coverage. In all cases, the *per-instruction* confidence counters saturate at 7. The labels should be understood in the following way: in all cases, the prediction is allowed to proceed if its *per-prediction* confidence is equal to 3 **and** (accuracy-oriented) / **or** (coverage-oriented) the *per-instruction* confidence is greater or equal to *Thresh*. *Pen* gives the misprediction penalty of the *per-instruction* confidence. In the last configuration (*Mixed*), the prediction is valid if either: (confidence is 3 and the *per-instruction* confidence is

[4] Amount subtracted to the confidence counter when the prediction is wrong

strictly greater than 3) or (confidence is strictly greater than 0 and the *per-instruction* confidence is 7). The misprediction penalty is 4. The *Mixed* configuration aims to obtain very high accuracy and a slightly higher coverage by allowing prediction whose confidence is 1 or 2 to proceed if their *per-instruction* confidence is high enough.

OR-ing *per-instruction* confidence with *per-prediction* confidence (bars 1 to 6) indeed yields a better coverage (37.2% to 42.4% vs. 36.6% for the reference), but always at a cost in accuracy (82.3% to 93.6% vs. 94.4% for the reference). Consequently, even though coverage is better, accuracy is too low to be interesting. Varying the threshold and the misprediction penalty only allows the modified hybrid to approach the accuracy of the reference in the 6th and 7th bar, but the gain in coverage is too low (0.7-0.8%). Therefore, leveraging *per-instruction* confidence to increase coverage only results in either a little increase in coverage at a non-negligible cost in accuracy or no gain at all.

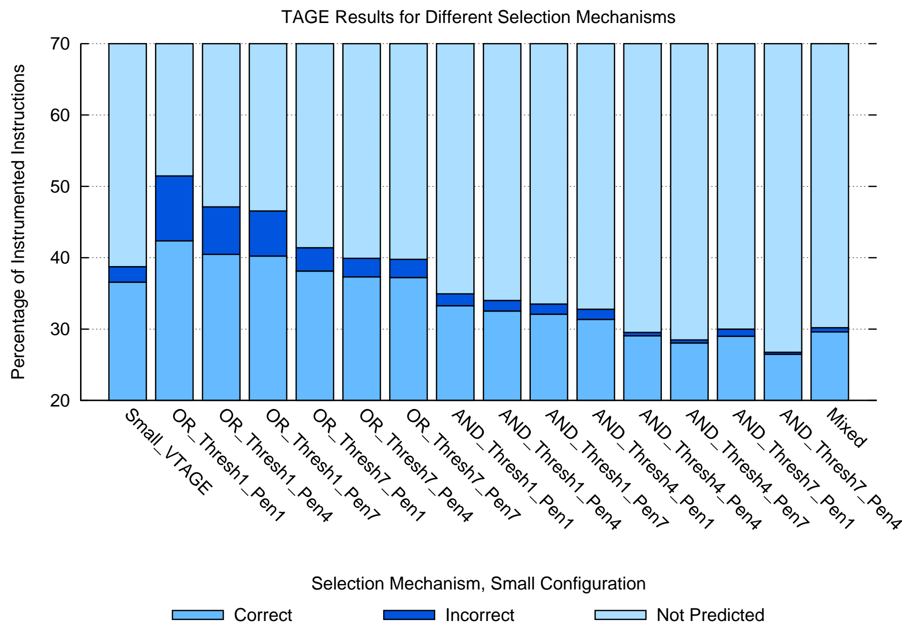


Figure 4.12: *Small* VTAGE predictor results with different confidence mechanisms.

Predictor	Accuracy	Coverage	Predictor	Accuracy	Coverage
Small VTAGE	94.4%	36.6%	AND_Thresh1_Pen4	95.6%	32.5%
OR_Thresh1_Pen1	82.3%	**42.4%**	AND_Thresh1_Pen7	95.7%	32.0%
OR_Thresh1_Pen4	85.9%	40.5%	AND_Thresh4_Pen1	95.6%	31.3%
OR_Thresh1_Pen7	86.4%	40.2%	AND_Thresh4_Pen4	98.4%	29.0%
OR_Thresh7_Pen1	92.1%	38.1%	AND_Thresh4_Pen7	98.5%	28.0%
OR_Thresh7_Pen4	93.5%	37.3%	AND_Thresh7_Pen1	96.7%	29.0%
OR_Thresh7_Pen7	93.6%	37.2%	AND_Thresh7_Pen4	**98.9%**	26.5%
AND_Thresh1_Pen1	95.2%	33.3%	Mixed VTAGE	**98.0%**	**29.6%**

Table 4.9: Accuracy and coverage summary for different modified *Small* VTAGE predictors. Left: coverage-oriented modifications, right: accuracy-oriented modifications.

When AND-ing *per-instruction* confidence with *per-prediction* confidence (bars 8 to 15), what can be observed in general is an increased accuracy (95.2% to 98.9% vs. 94.4% for the reference) at a cost in coverage

(26.5% to 33.3% vs. 36.6% for the reference). Quite naturally, the stricter the selecting policy is, the higher is accuracy and the lower is coverage. Accuracy higher than 98% is reached by the 12th, 13th and 15th bars. Bar 15 is even very close to 99% accuracy. These are very good scores. Finally, the last bar shows moderate coverage at 29.6% but its accuracy is very high at 98.0%.

To summarize our findings, we saw that if *per-instruction* confidence can be used to increase coverage, the gains are not tremendous and too costly from the point of view of accuracy. Yet, using it to maximize accuracy yields much better gains, even though coverage is reduced. As a result, we define only two candidates for our modified hybrid : *AND_Thresh7_Pen4* (we will refer to it as the *Accurate* VTAGE configuration) because of its very high accuracy and *Mixed* VTAGE because of its high accuracy and slightly higher coverage. If the Stride predictor is able to make up for the lost in coverage, the Stride + VTAGE hybrid will become that much more interesting for a microarchitecture where the misprediction penalty is high.

4.4.2 Using Confidence Estimation instead of *Filtering* in Stride

The *Filtered* Stride predictor we used already performs relatively well. However, it is possible that accuracy could be increased by using confidence counters instead of a filtering mechanism. In essence, the filtering mechanism can be viewed as a confidence counter with a threshold of 2 which only oscillates between 1 and 2 since the predictor is not tagged. If the predictor were tagged, it would be reset to 0 every time the entry is evicted.

We added one 2 or 3-bit saturating counter to each entry in order to estimate the likelihood of being correct for the corresponding prediction. Fig. 4.13 summarizes the number of correctly, incorrectly and not predicted instructions. Table 4.10 summarizes accuracy and coverage. The labels should be understood in the following way: *W/WO_TAG* informs on the presence of a full tag in each entry of the predictor, **T** stands for **T**hreshold and is the value confidence has to be greater or equal to for the prediction to proceed, **S** stands for the **S**aturated value of the confidence counter and finally, **P** stands for the **P**enalty applied to the counter on a misprediction.

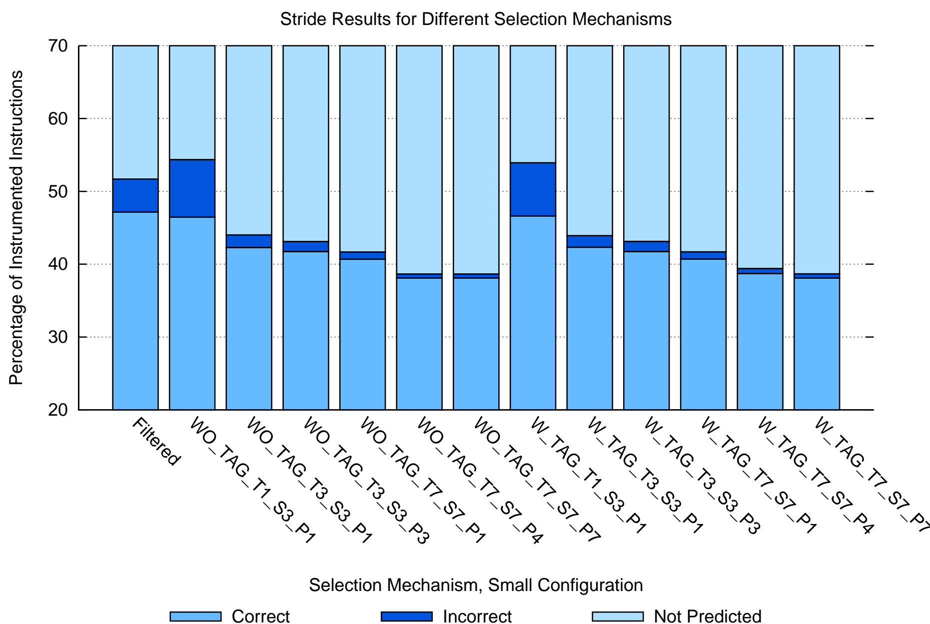


Figure 4.13: *Small* Stride predictor accuracy with different confidence mechanisms.

As suggested by the 2nd and 8th bars, using 2-bit saturating counters, a threshold of 1 and a penalty

of 1 is not strict enough. Indeed, accuracy is lower than that of the *Filtered* Stride (85.5% vs. 91.2%) and coverage is also worse (46.4% vs. 47.1%). Setting the threshold to 3 (3rd and 9th bars) allows to increase accuracy (from 91.2% to 96.1%)) at a cost in coverage (from 46.4% to 42.3%). Setting the misprediction penalty to 3 (4th and 10th bars) has little effect on coverage and accuracy.

If we go further and use 3-bit saturating counters with a threshold of 7 (bars 5 to 7 and 11 to 13), we can observe that accuracy can be as high as 98.5%, with coverage still around 38.0%, which is actually better than the *Small* VTAGE in the *Strict* configuration previously described in 4.4.1.

Fully Tagging the entries in the Stride predictor has little effect in our case, as shown by the right bars in Fig. 4.13. However, the small difference may be due to the limited number of static instructions encountered by the predictor during the runs.

Predictor	Accuracy	Coverage	Predictor	Accuracy	Coverage
Small F-Stride w/o TAG	91.2%	**47.1%**	Small F-Stride w/ TAG	91.3%	45.4%
WO_TAG_T1_S3_P1	85.5%	46.5%	W_TAG_T1_S3_P1	86.4%	46.6%
WO_TAG_T3_S3_P1	96.1%	42.3%	W_TAG_T3_S3_P1	96.4%	42.3%
WO_TAG_T3_S3_P3	96.8%	41.7%	W_TAG_T3_S3_P3	96.8%	41.7%
WO_TAG_T7_S7_P1	97.6%	40.7%	W_TAG_T7_S7_P1	97.6%	40.7%
WO_TAG_T7_S7_P4	98.5%	38.0%	W_TAG_T7_S7_P4	98.2%	38.7%
WO_TAG_T7_S7_P7	**98.5%**	38.0%	W_TAG_T7_S7_P7	**98.5%**	38.1%

Table 4.10: Accuracy and coverage summary for different modified *Small* Stride predictors. Left: untagged entries, right: fully tagged entries.

To conclude, using saturating counters to implement confidence actually appears more efficient than using filtering. Such counters would require none to little additional storage as the filtering mechanism already requires two bits per entry. As a matter of fact, a Stride predictor with 3-bit counters will consume only *#entries* more bits than the corresponding *Filtered* Stride predictor. They would also simplify the update policy since no comparison would have to be made between the observed stride and the stored stride. Furthermore, since coverage is moderate (38%) but accuracy is very high, we can expect the modified Stride + VTAGE hybrid to have very high accuracy and moderate coverage as both distinct predictor have non negligible complementarity, as shown in 4.2.2. As in the previous section, we select two candidates for our modified hybrid : *W_TAG_T7_S7_P4* and *WO_TAG_T7_S7_P4* (we will respectively refer to them as the *Accurate w/* and *Accurate w/o* Stride configurations) because of their very high accuracy.

4.4.3 Performance Evaluation of the Updated FS-VTAGE

The selected configurations of VTAGE (*Accurate* and *Mixed*) and Stride (*Accurate w/* and *Accurate w/o*) were combined to form four different configurations of the **C**onfidence **S**tride - **D**ual-**C**onfidence **V**alue **TAGE** hybrid (CS-DCVTAGE). Fig. 4.14 and 4.15 summarize the number of correctly, incorrectly and not predicted instructions. Table 4.11 and 4.12 respectively summarize accuracy and coverage for integer and FP/SIMD workloads. The experimental frameworks for integer and FP/SIMD were respectively described in 4.2.1 and 4.3.1

Results for integer workloads show that fully tagging the *Filtered* Stride predictor of the reference FS-VTAGE changes little since accuracy is slightly higher at 91.8% and coverage only decreases from 56.9% to 56.8%. Using longer tags for the tagged components of VTAGE yields a very modest increase in accuracy (from 91.5% to 91.6%) and a small increase in coverage (from 56.9% to 57.1%). The small difference between a tagged F-Stride and an untagged F-Stride is probably due to the fact that most static instructions fit in an 8K-entry predictor. Consequently, we only claim that modifying the length of the tag of tagged components in VTAGE does not seem interesting. We do not conclude on tagging the F-Stride predictor, although it appears to help it reach an even higher accuracy.

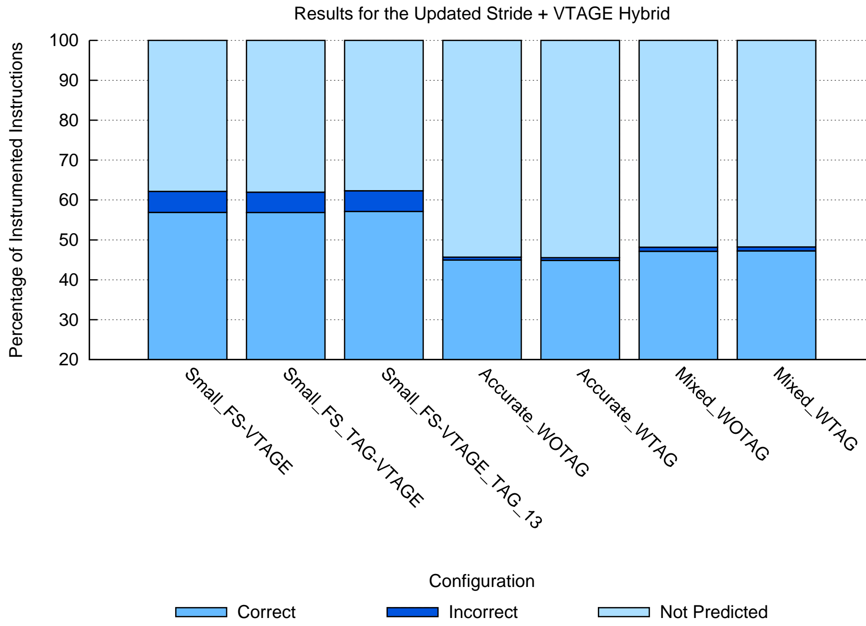


Figure 4.14: Results for the updated *Small* Stride + VTAGE Hybrid.

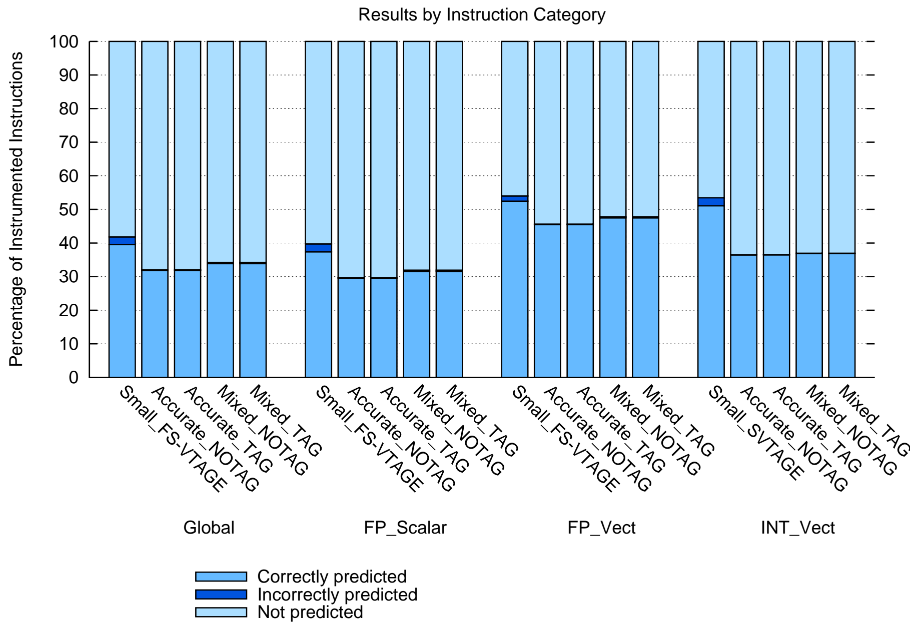


Figure 4.15: Results for the updated *Small* Stride + VTAGE Hybrid.

Continuing our analysis, the configurations based on the *Mixed* DCVTAGE predictor show very high accuracy at 97.9%, coverage being moderate at 47.2%. The configurations based on the *Accurate* VTAGE have the best accuracy at 98.5% and 98.4% depending on the Stride predictor being tagged or not. Coverage is slightly lower at 44.9%. The predictors behave similarly for FP and SIMD workloads, even though coverage is worse for scalar FP instructions and better for FP vector instructions, much like in the numbers provided

in 4.3.3.

Predictor	Accuracy	Coverage
Small FS-VTAGE	91.5%	56.9%
Small F-Stride w/ TAG + VTAGE	91.8%	56.8%
Small F-Stride + VTAGE TAG 13	91.6%	**57.1%**
Accurate w/o TAG	98.4%	44.9%
Accurate w/ TAG	**98.5%**	44.9%
Mixed w/ TAG	97.9%	47.2%
Mixed w/ TAG	97.9%	47.2%

Table 4.11: Accuracy and coverage summary for different modified *Small* Stride + VTAGE predictors. Integer workloads.

	Global		FP_Scalar		FP_Vect		INT_Vect	
Predictor	Acc.	Cov.	Acc.	Cov.	Acc.	Cov.	Acc.	Cov.
Small FS-VTAGE	94.7%	**39.6%**	94.1%	**37.4%**	97.2%	**52.4%**	95.5%	**51.1%**
Accurate w/o TAG	99.4%	31.8%	99.3%	29.5%	99.7%	45.5%	99.9%	36.5%
Accurate w/ TAG	**99.4%**	31.8%	**99.3%**	29.5	**99.7%**	45.5%	**99.9%**	36.5%
Mixed w/o TAG	98.9%	33.8%	98.8%	31.5%	99.3%	47.5%	99.7%	36.9%
Mixed w/ TAG	98.9%	33.8%	98.8%	31.5%	99.3%	47.5%	99.7%	36.9%

Table 4.12: Accuracy and coverage summary for different modified *Small* Stride + VTAGE predictors. FP/SIMD workloads.

To summarize, using global counters is not very effective if the goal is to further increase coverage. However, if only the number of mispredictions is considered, filtering the *per-prediction* confidence with the *per-instruction* counters drastically reduces the number of mispredictions in VTAGE, at a moderate cost in coverage (from 10 to 12% of the instrumented instructions). The *Mixed* configuration of DCVTAGE slightly reduces the loss in coverage, but it is not as precise as the *Accurate* configuration (98.0% vs. 98.9%). Ultimately, the choice between these two configurations depends on the requirements of the microarchitecture regarding accuracy.

Furthermore, the Stride predictor augmented with 3-bit saturating confidence counters has a very high accuracy and a higher coverage than DCVTAGE. That is, if we consider the CS-DCVTAGE hybrids, we obtain accuracy scores that were never attained by other predictors given the corresponding coverage scores. This clearly demonstrates the superiority of the CS-DCVTAGE hybrid when very high accuracy is critical to Value Prediction.

Chapter 5

Future Work

5.1 Predicting Memory Instruction Addresses and *Store* Values

It is not seldom for a value to be stored only to be reloaded a few instructions later because physical registers are too few. To that extent, predicting the value of a store instruction [7] may allow for early speculative *store*-to-*load* forwarding, among other things.

Furthermore, *load* and *store* addresses are usually more predictable than *load* values [6]. As a consequence, memory disambiguation can be improved and instead of predicting the value of a *load*, its address can be predicted and the value prefetched. Since our value predictor stores full register values, it can seamlessly adapt to store memory addresses.

5.2 Criticality

A realistic predictor cannot predict the result of all in-flight instructions. Predictions must therefore be prioritized. Several contributions make the observation that the speedup obtainable from Value Prediction is not strongly correlated with the accuracy or the coverage of the predictor, but rather with the number of *critical* results correctly predicted [2, 8, 15, 16]. A *critical* result is essentially a result belonging to an instruction which is on the current critical path of the pipeline. That is, if this result is known in advance, a bigger gain can be expected.

Using confidence mechanisms allows to have a misprediction probability in the range of 1-2%, as demonstrated in 4.4.3. This alone reduces pressure on the predictor in the sense that a value is transferred from the predictor to the execution units only if it has very high probability to be correct. The next step is to avoid such a transfer when the prediction is not *useful*. To that extent, a single *data-requested* bit can be used to only update the predictor if a depending instruction has read the prediction [15]. This mechanism reduces the number of updates required as well as the number of *useless* predictions transferred from the predictor. However, in the case where confidence is used to filter the predictions, a certain care must be taken to always update newly allocated entries so their confidence can grow. If such updates are not done, confidence of newly allocated entries will never grow and no prediction will ever be made.

Similarly, *update confidence* can be kept per entry in the predictor and serve the purpose of deciding to update an entry (or allocating a new one in VTAGE) or not [17]. An even stricter mechanism is proposed in [2]: the processor is assumed to be able to construct the *critical path* existing within its pipeline at any given time in order to only predict critical results. With this mechanism, it is shown that almost the full potential speedup of value prediction can be achieved while only predicting 60% of the register-producing instructions.

A less costly - and less efficient - mechanism is to prioritize instructions depending on their type (integer, memory, FP). For instance, [7] states that for an ideal processor (infinite buffers, instruction window, perfect

caches, perfect branch prediction and so on), it is more interesting to predict arithmetic instructions. However, it is generally more interesting to predict the result of *load* instructions than the result of arithmetic instructions for realistic implementations [2]. Such an approach is adopted in [13].

Future work regarding the criticality of predictions will consist in simulating a processor implementing Value Prediction and evaluating the speedups obtained given the different methods proposed to estimate the importance of a prediction.

5.3 Microarchitectural Considerations

In order to simulate a processor implementing Value Prediction, which is necessary to obtain speedup results, one must decide how to integrate Value Prediction in a given microarchitecture. More specifically, two issues must be addressed: how to handle mispredictions and when to resolve (verify) a prediction.

5.3.1 Misprediction Recovery

Misprediction recovery has been proven to be critical depending on the accuracy of the value predictor [27]. Three major mechanisms are available [25]:

- *Squashing* (or *Refetch*) removes the faulting instruction and all subsequent ones from the pipeline and refetches them from the instruction cache. This is identical to the branch misprediction recovery mechanism, the penalty is therefore similar (high).
- *Reissue* marks the faulting instruction and all subsequent ones (even **independent** ones) as re-eligible for issue. That way, these instructions do not need to go through the fetch, decode and dispatch stages another time. As a result, speculatively issued instructions must retain their entry in the instruction queue to be able to reissue as soon as possible.
- *Selective Reissue* is similar to *reissue* except only the faulting instruction and **dependent** ones are marked as re-eligible for issue. However, the processor must be aware of the dependencies exesting between all speculative instructions in flight.

Selective Reissue is generally the most efficient. However, assuming an architecture based on a small *instruction queue*, *Squashing* actually performs better than the *Reissue* mechanism because the latter puts too much pressure on the instruction queue, even though its misprediction penalty is lower [25]. This is because all instructions following a speculatively issued instruction must retain their entry in the instruction queue, so new instructions are not be able to enter it. This might however not be the case on a different architecture.

As stated in [27], *Squashing* is not suited to Value Prediction because when a value is misspeculated, the path of execution does not change, that is, the instruction must be re-executed but contrary to branch misprediction recovery, no “backtracking” has to be done. Since *Reissue* is not always efficient, *Selective Reissue* should be used if possible. However, several microarchitectural issues arise, specifically, how should the processor track the *critical path* inside its pipeline?

5.3.2 Prediction Resolution

In essence, prediction resolution can be *speculative* or *non-speculative* [27]. To illustrate these choices, consider an instruction whose result is predicted and for which one operand is also predicted. In the case of *speculative resolution*, once the instruction leaves the execution stage, its predicted result is compared against the computed result and if both are matching, the result is considered to be *correct* when it is still speculative because one of its operand also is. In the case of *non-speculative* resolution, the comparison is delayed until the speculative operand is resolved.

Non-speculative resolution is the simplest scheme. Its main advantage is that a predicted instruction can re-execute at most once. However, it implies that each operand of an instruction must be marked as speculative or non-speculative in order for an instruction to tell if it can resolve its predicted result or not. As a result, it may be possible for resolution to become a bottleneck since a long *resolution chain* might build itself up in the processor and require several cycles to shorten.

On the contrary, *speculative* resolution does not need to mark operands since it is assumed that previous predictions are always correct. Hence, the resolution hardware is likely to be better utilized. However, with this method, a predicted instruction can be re-executed several times and suffering the misprediction penalty more than once for only one instruction will impede performance. Therefore, *speculative* resolution appears as a legitimate mechanism only if the value predictor is highly accurate. Fortunately, this is the case of the CS-DCVTAGE described in 4.4.3. Therefore, future work must study the interest of *speculative* resolution over *non-speculative* resolution in the presence of such a Value Predictor. Furthermore, we need to assess the limits of the *non-speculative* resolution method in the presence of a very wide superscalar processor where many instructions may need validation at any given time.

5.3.3 Perspectives Regarding Branch Prediction

Interestingly, Zhou, Fu, Rotenberg and Conte claimed that the speedup obtained with perfect Value Prediction and perfect Branch Prediction is greater than the sum of both distinct speedups [27]. This holds for a 4-wide superscalar processor based on the MIPS R10000 implementing Value Prediction with *Selective Re-issue* and *non-speculative resolution* (see 5.3.1). Therefore, Branch and Value Prediction seem to be more entangled than expected and very high branch prediction accuracy will make the processor able to tolerate more value mispredictions for the same speedup. This is mostly due to the fact that less value-speculated instructions which also happened to be on the wrong path will need to be reissued.

Furthermore, if we look at the matter from the other angle, very high value prediction accuracy will benefit to the branch predictor. Indeed, if the value predictor is able to deliver correct values well in advance, the branch predictor can speculate on the control flow earlier than it usually does. Given a highly accurate branch predictor, this could for instance better hide the latency of a miss in the instruction cache.

5.4 Vector Value Prediction

Predicting the result of SIMD instructions is not that different from predicting the result of scalar instructions in concept. The main difference is that SIMD registers are several times larger than general purpose registers. Therefore, predicting their content implies either having larger entries in the predictor or accessing several entries of the predictor to fill the entire SIMD register.

The first solution is unacceptable: huge entries may be required (AVX *ymm* registers are 256-bit wide as of now), and most of the storage will be wasted in the event of the entry storing the value of a general purpose register. A variation is to dedicate most of the storage to normal entries and specialize some entries for vector instructions. However, as the length of the vector register grows, the amount of wasted storage when there are no vector instructions in the code also grows.

The second solution is more interesting but it implies several accesses to the predictor. Those accesses may or may not be serialized, but in the latter case, the predictor response time is likely to be impacted. As VTAGE features several components, one could imagine distributing a large result among several components in order to limit the number of accesses to one per component. The only prerequisite is to have at least as many components as the maximum width of a vector register divided by the width of a scalar register (e.g. 4 components for 256-bit vector registers and 64-bit scalar registers). When a prediction is requested, the predictor should be able to tell if it is required to produce a scalar or a vector. To that extent, each request sent to the predictor should be delayed to after the opcode of the instruction is known. To form a vector, the value of all n highest hitting component are concatenated. If the number of hitting components is less than n, no prediction is made (the base predictor always hits). Interestingly, no additional information concerning the type of value stored in an entry is required, even though having this information might be interesting to

help the replacement policy. For instance, if an entry is to be evicted but is part of a set of entries predicting the result of a vector instruction, it may not be interesting to replace it.

Conclusion

We began with a brief summary of existing limitations to sequential performance by introducing the *dataflow limit* in superscalar processors. Then, different value predictors were described as a way to implement *Value Prediction*, allowing the *dataflow limit* to be speculatively ignored. We also recalled that value predictor can be classified in two categories: computational and context-based. In the meantime, we described the ITTAGE Geometric Length indirect branch target predictor and highlighted the similarities of Branch Target Prediction and Value Prediction.

Based on these similarities, we evaluated the potential for using a Geometric Length value predictor as a value predictor. We found that instances of a given static instruction producing the same result shared many similar bits in their respective global branch histories and path histories. From there, we naturally chose to adapt the ITTAGE branch target predictor to Value Prediction and described the **V**alue TAGE predictor. Because hybrid value predictors have proven more efficient than single component predictors, we added a simple computational predictor to the VTAGE predictor to form the **F**iltered **S**tride + VTAGE (FS-VTAGE) hybrid.

We evaluated the interest of FS-VTAGE itself on integer workloads by gauging the complementarity of its two components and showed that a non-negligible percentage of register-producing instructions could only be predicted by a single component. Furthermore, we found that FS-VTAGE is able to achieve the highest accuracy of all considered hybrid predictors and is second to the PS-PI predictor for coverage, although not by much.

Yet, *Value Locality* is not limited to integer instructions since it has been shown that floating-point instructions are also well predictable, though the predictability of SIMD instructions was, to our knowledge, never considered. Consequently, we decided to use the SSE instruction set (extending the x86_64 ISA) to evaluate both the behavior of FS-VTAGE on floating point workloads and the predictability of SIMD instructions. Our results, concur with previous work regarding the predictability of floating-point instructions. Interestingly enough, they also show that SIMD instructions are somehow more predictable than their scalar counterparts, since the FS-VTAGE hybrid coverage is around 40% for scalar instructions and 50 to 55% for FP SIMD instructions, with similar accuracy. This suggests that code portions using SIMD instructions could be further accelerated by Value Prediction, granted a compatible predictor.

Nonetheless, Value Prediction appeals for very high accuracy since mispredicting is likely to induce severe penalties depending on the microarchitecture. Therefore, we proposed to add *per-instruction* confidence to the VTAGE predictor to further reduce the number of mispredictions. Following the same approach, we compared the *Filtered* Stride predictor to a simple Stride predictor augmented with a confidence estimation mechanism and found that the latter was more accurate. The two modified predictors were then combined to form an updated Stride + VTAGE (CS-DCVTAGE) predictor able to reach an accuracy score of 98.5% (resp. 99.4% for FP) while still covering nearly 45% of the general purpose register-producing instructions (resp. 30 to 35% of the special purpose register-producing SSE for both scalar and vector instructions).

We conclude this report by suggesting some directions for future work. First, an essential direction to explore is the usefulness of predictions, that is the fact that a correct prediction may or may not reduce the overall execution time. Even though gains in IPC have been shown in previous work using a simple Stride predictor, one could also want to further study the following observation: VTAGE can predict some results that Stride cannot. It is very possible that these predicted results are more relevant in the sense that they

may be able to release more subsequent instructions that are on the *critical path*, especially when instruction exhibiting a stride behavior are not usually on the critical path [8]. Even so, bandwidth limitations will cap the number of predictions the hybrid is able to make every cycle. To that extent, prioritizing which result to predict prior to even knowing which component will make the prediction is also necessary. Second, we described some microarchitectural choices that one needs to make when implementing Value Prediction and highlighted the relation between Value and Branch Prediction. Lastly, an interesting direction to further explore is the design of an efficient value predictor able to predict values of arbitrary widths. Such a predictor could seamlessly adapt to a workload containing vectorized code. We gave some insights as to what must be considered in the last section.

To summarize in a few words, we proposed the FS-VTAGE value predictor from which the context-based component is derived from previous work in branch prediction. This hybrid predictor achieves the highest accuracy so far and has moderate coverage. For a realistic - in the near future - storage budget (1.6 Mbits) FS-VTAGE is leading on accuracy by a fair margin (around 10 points) on both integer, floating-point and vector workloads. It is second to the PS-PI on coverage by a small margin: 0.1 point for integer, 2 points for FP and SIMD. The same behavior can be said for the idealistic configuration we considered. As Value Prediction necessitates accuracy in the 95-99% range to be beneficial, we proposed to modify the F-Stride and VTAGE components to create the CS-DCVTAGE value predictor and reach an even higher accuracy of 98.5% (99.4% for FP/SIMD). The accuracy of CS-DCVTAGE is actually the highest of all considered hybrid predictors, and its coverage is still reasonable. Therefore, we claim that CS-DCVTAGE is the most efficient hybrid value predictor of the considered predictors. Yet, simulation to assess the real gain in execution time is still required to validate this contribution.

Bibliography

[1] IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, 1985.

[2] B. Calder, G. Reinman, and D.M. Tullsen. Selective value prediction. In *ACM SIGARCH Computer Architecture News*, volume 27, pages 64–74. IEEE Computer Society, 1999.

[3] Standard Performance Evaluation Corporation. CPU2006. http://www.spec.org/cpu2006/.

[4] R.J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37(4):547–564, 1993.

[5] F. Gabbay and A. Mendelson. *Speculative execution based on value prediction*. Citeseer, 1996.

[6] J. González and A. González. Speculative execution via address prediction and data prefetching. In *Proceedings of the 11th international conference on Supercomputing*, pages 196–203. ACM, 1997.

[7] J. González and A. González. Limits of instruction level parallelism with data speculation. In *Proc. of the VECPAR Conf*, pages 585–598. Citeseer, 1998.

[8] J. González and A. González. The potential of data value speculation to boost ilp. In *Proceedings of the 12th international conference on Supercomputing*, pages 21–28. ACM, 1998.

[9] J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Pub, 2011.

[10] Intel. PIN Instrumentation Tool. http://www.pintool.org/.

[11] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A, 2B, and 2C: Instruction Set Reference, A-Z*. Intel Corporation, May 2012.

[12] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 226–237. IEEE Computer Society, 1996.

[13] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. *ACM SIGOPS Operating Systems Review*, 30(5):138–147, 1996.

[14] T. Nakra, R. Gupta, and M.L. Soffa. Global context-based value prediction. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 4–12. IEEE, 1999.

[15] B. Rychlik, J. Faistl, B. Krug, and J.P. Shen. Efficacy and performance impact of value prediction. In *pact*, page 148. Published by the IEEE Computer Society, 1998.

[16] R. Sathe and M. Franklin. Available parallelism with data value prediction. In *High Performance Computing, 1998. HIPC'98. 5th International Conference On*, pages 194–201. IEEE, 1998.

[17] Y. Sazeides and J.E. Smith. Implementations of context based value predictors. Technical report, Citeseer, 1997.

[18] Y. Sazeides and J.E. Smith. The predictability of data values. In *micro*, page 248. Published by the IEEE Computer Society, 1997.

[19] A. Seznec. Analysis of the O-GEometric History Length Branch Predictor. *International Symposium on Computer Architecture*, 0:394–405, 2005.

[20] A. Seznec. Genesis of the O-GEHL branch predictor. *Journal of Instruction-Level Parallelism*, 7, 2005.

[21] A. Seznec. The L-TAGE Branch Predictor. *Journal of Instruction-Level Parallelism*, 2007.

[22] A. Seznec. A 64-Kbytes ITTAGE indirect branch predictor. *Journal of Instruction-Level Parallelism*, 2011.

[23] A. Seznec. Storage free confidence estimation for the tage branch predictor. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 443–454. IEEE, 2011.

[24] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length branch prediction. *Journal of Instruction Level Parallelism*, 8:1–23, 2006.

[25] D.M. Tullsen and J.S. Seng. Storageless value prediction using prior register values. In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pages 270–279. IEEE, 1999.

[26] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 281–290. IEEE Computer Society, 1997.

[27] H. Zhou, C. ying Fu, E. Rotenberg, and T. Conte. A study of value speculative execution and misspeculation recovery in superscalar microprocessors. *Department of Electric & Computer Engineering, North Carolina State University, pp.–23*, 2000.