Evaluation and Benefit of Imprecise Value Prediction for Certain Types of Instructions

Uroš Radenković*, Marko Mićović and Zaharije Radivojević

School of Electrical Engineering, University of Belgrade, Bulevar kralja Aleksandra 73, 11000 Belgrade, Serbia; micko@etf.bg.ac.rs (M.M.); zaki@etf.bg.ac.rs (Z.R.)
* Correspondence: uki@etf.bg.ac.rs

Abstract: Based on branch prediction, value prediction has emerged as a solution for problems caused by true data dependencies in pipelined processors. While branch predictors have binary outcomes (taken/not taken), value predictors face a challenging task as their outcomes can take any value. Because of that, coverage is reduced to enhance high accuracy and minimise costly recovery from misprediction. This paper evaluates value prediction, focusing on instruction execution with imprecisely predicted operands whose result can still be correct. Two analytical models are introduced to represent instruction execution with value prediction. One model focuses on correctly predicted operands, while the other allows for imprecisely predicted operands as long as the instruction results remain correct. A trace-driven simulator was developed for simulation purposes, implementing well-known predictors and some of the predictors presented at the latest Championship Value Prediction. The gem5 simulator was upgraded to generate program traces of SPEC and EEMBC benchmarks that were used in simulations. Based on the simulation result, proposed analytical models were compared to reveal the conditions under which a model with imprecisely predicted operands, but still correct results, achieved better execution time than a model with correctly predicted operands. Analysis revealed that the accuracy of the correct instruction result based on the predicted operand, even when the predicted operand is imprecise, is higher than the accuracy of the correctly predicted operand. The accuracy improvement ranges from 0.8% to 44%, depending on the specific predictor used.

Keywords: value prediction; speculative execution; computer architecture; performance evaluation

1. Introduction

Speculative execution is a vital technique that modern processors use to improve performance. This technique involves executing instructions in advance without knowing whether those instructions will actually be executed. While this technique brings significant improvements, it also presents complexity [1,2] and security [3,4] challenges that must be carefully approached when designing processors. Handling mispredictions and ensuring the correct execution state is maintained requires additional processor time and resources, including complex processor organisation and more energy consumption. Security vulnerabilities exploited the speculative execution process to gain unauthorised access to sensitive data. Some speculative execution techniques include branch prediction and value prediction.

For many decades, pipelined processors have used branch predictors (BPs) to avoid control dependencies and achieve better throughput of instructions. The main task of a BP is to predict which instruction should be executed next after the branch instruction. The prediction’s outcome can be either an instruction immediately after the branch instruction or an instruction from the branch target address. Therefore, there are only two possible outcomes: a branch taken or a branch not taken.

Based on this very idea of using BPs, computer architects have proposed many mechanisms for value prediction, including value predictors (VPs). The primary purpose of a VP
is to avoid true data dependencies when an instruction cannot continue execution because it must wait for data that some other instruction will produce. Unlike BP, which has only two possible outcomes, the outcome for a VP can be any value that fits in the data length bits. Therefore, it is challenging for computer architects to evolve a highly accurate VP.

BPs found in the literature [5] have very high accuracy that can be above 99% and cover all conditional branches. On the other side, VPs do not have accuracy at that level if they cover all types of instructions. In order to achieve higher accuracy, VPs have confidence mechanisms that determine whether they will even predict data for a particular instruction [1,6–8]. Also, some VPs are specialised in certain types of instructions and make predictions only for them [9–12]. In that way, VP achieves higher accuracy but at the expense of instruction coverage. For VP, it is essential to have high accuracy because every time they mispredict value, it is necessary to restore the architectural state before the start of speculative execution. A misprediction recovery penalty was identified as one of the most complicated factors for implementing VPs [1].

In this paper, our aim is to evaluate how VPs behave if they predict operands only for certain types of instructions whose results can be correct even though some operands do not have correct values. Generally, VPs can directly predict the results of instructions, usually the value of the instruction’s destination register. Instead, we use VPs to predict instruction operands, and then the instruction results are calculated based on the predicted operands.

The introduction of a memory hierarchy in the processor’s design, primarily cache memory, aims to hide the memory delay and mitigate the problem that the memory is significantly slower than the processor (memory wall). If the data requested by the processor is in cache memory, access to the main memory is avoided, thus speeding up execution. When a cache miss occurs, the processor accesses memory for missing data, halting instruction execution until the access completes. A VP can be included in that situation to predict memory operands, and the processor can continue with speculative execution. Therefore, operands to predict are those originating from memory because their values may not be immediately available as register operands. Also, [9] reported that VPs for load instructions (one operand originates from memory) are more efficient than for other instruction types. Our research focused on showing the phenomenon that it is possible to obtain a correct result even based on an imprecisely predicted operand. We wanted to see the maximum potential of execution with an imprecisely predicted operand. Therefore, reads from the memory of instruction operands that are suitable for imprecise prediction are treated as cache misses. In each such case, a VP is used to predict the operand value coming from memory.

We observe the accuracy of correctly predicted operands and the accuracy of correct instruction results calculated based on predicted operands. VPs have the complicated and challenging task of predicting the entirely correct value. We want to explore in which situations the result of an instruction can be correct, although its predicted operands do not have the correct value. Suppose the result of an instruction based on an imprecisely predicted operand is correct and no subsequent instructions depend on the predicted operand. In that case, the execution can proceed correctly.

Also, in this paper, we present two analytical models that describe the time needed to execute some number of instructions if VP predicts operands, which come from memory, for the instructions we discussed above. The first model represents a situation when only the correct value of the predicted operand is acceptable. The second model represents a situation when the imprecise value of the predicted operand is acceptable because the result of the instruction is correct with that value. We will compare these two models based on the accuracy of the existing VPs. The assumption is that the second model will achieve better time because we suppose that the accuracy of the correct instruction result, based on the predicted memory operand (perhaps imprecisely predicted), is greater than the accuracy of the correctly predicted memory operand.

To the best of our knowledge, this is a novel and innovative consideration of speculative execution that may lead to the correct result of certain types of instructions, even if the
operand is imprecisely predicted. In this way, it is possible to obtain a correct instruction result because, for specific types of instructions, the result will not depend on all the bits of the operand for which the value is predicted. Some subsequent instruction that uses the result of the instruction for which the operand is predicted will obtain the correct result, besides the potentially imprecisely predicted operand. This further provides the possibility that recovery must be performed only in the case of an incorrect result and not always when the operand is imprecisely predicted. Summarising the challenges mentioned earlier, the main contributions of this paper to this field of computer architecture, specifically speculative execution using imprecise value prediction, are listed below:

- Speculative execution of specific instructions is proposed, where it is possible to obtain a correct result based on an imprecisely predicted operand. The idea of the proposed solution is that predictors can be used to predict the operands of certain types of instructions, the result of which does not entirely depend on the predicted operand. The characteristics of these instructions are described in Section 3;
- Also, the proposed solution for executing specific instructions based on an imprecisely predicted operand includes the situation when it is possible to obtain the correct part of the result that some instructions will use. The part of the result that some instructions will use is called the useful result, which is described in Section 3;
- Two analytical models have been proposed that describe the speculative execution of instructions when predicting the value of operands. One covers the situation where only a correctly predicted operand is acceptable, and the other covers a proposed solution where an imprecisely predicted operand is also acceptable. The models are described in Section 4;
- An evaluation of the benchmarks used was done in order to determine the number of operand bits that need to be correctly predicted and which affect the result in the case of an imprecise prediction of the operand for specific instructions. This analysis is given in Section 5;
- An evaluation of the proposed solution and a comparison of two analytical models were made in order to describe the conditions under which a model with an imprecise predicted operand is profitable to use. This is presented in Section 6.

The rest of the paper is organised as follows. Section 2 provides an in-depth description of the existing value predictors and highlights the key challenges associated with their implementation. Section 3 delves into the motivation behind the value prediction for certain types of instructions. Section 4 introduces instruction execution models incorporating value prediction, represented as analytical models. Section 5 outlines the methodology employed in this paper and provides a detailed description of the simulations conducted. Section 6 presents the results obtained from the simulations, followed by a thorough discussion. Finally, Section 7 encapsulates the conclusion of this paper.

2. Value Prediction

This section provides an overview of various well-known value predictors. It includes a concise description of how the value prediction mechanism is utilised. Additionally, this section briefly describes the challenges and problems of value prediction.

2.1. Usage of VPs

As mentioned above, VPs were invented to break true data dependencies at the hardware level. VPs enable speculative instruction execution with predicted value before the instructions that have to produce value complete their execution. The idea of value prediction is presented for the first time in [12,13]. Also, value locality, the third facet of locality besides spatial and temporal locality, was observed in [12]. The notion of value locality says that a previously seen value within a storage location can repeat itself in the future.

Value prediction can be employed in other scenarios besides those previously described. In [8,14], VP is used to predict the data address, which would be used for data
prefetching. The authors of [15] presented the BP in combination with VP, where VP predicts the data necessary to determine branch condition. Paper [16] described speculative strength reduction, a mechanism that replaces the instruction with an equivalent but less costly instruction if the operand has a value of zero or one. The VP is used to predict if the source operands have these values.

2.2. Recovery from Mispredictions

At some point in execution, the processor must check whether the predicted value is correct, and if not, the architectural state must be restored. Two approaches are described in [1]: pipeline squashing and selective reissue. Pipeline squashing is a mechanism used to recover from branch misprediction, where all instructions after branch instructions are flushed. Applying pipeline squashing to the value prediction means that all instructions will be flushed when a misprediction occurs. On the other hand, the selective reissue mechanism does not flush all instructions from the pipeline but only replays the instruction with the incorrect operand and all dependent instructions.

VPs must employ a confidence mechanism to minimise mispredictions and avoid wasting time on recovery [1,6–8]. The confidence mechanism is usually implemented as counters whose values represent confidence in predicting [7]. Specialising in certain types of instructions also helps VPs obtain higher accuracy. For example, several predictors are presented for load instruction [10,12,17]. With confidence mechanisms and specialisation for certain types of instruction, predictors try to reach higher accuracy and avoid a misprediction recovery penalty at the cost of reduced coverage.

2.3. Existing Value Predictors

The VPs can be divided into two groups: computational and contextual-based predictors [18]. Computational predictors apply some function to the previous value to predict the next value. The prominent representatives of this group are the Last Value Predictor [19], the Load Value Predictor [12], the Stride Predictor [19,20] and the 2-Delta Stride Predictor [13]. On the other hand, contextual-based predictors track context and the value associated with the context. Contexts consist of a finite number of previous values. When the same context is repeated, the predictor gives the value associated with the context as a prediction. The representative of this group is the Finite Context Method Predictor [21]. Contextual-based predictors can use branch history to capture the correlation of instruction results with branch history [11]. The VTAGE Predictor [1] is one of the VPs that uses branch history.

Apart from the predictors mentioned in the paragraph above, there are some predictors that are presented in the latest Championship Value Prediction [22]. These are hybrid predictors that combine two or more different predictors to predict value. Noticeable hybrid predictors are the EVES Predictor [23], the CBC-VTAGE Predictor [6] and the H3VP Predictor [24].

Stride and 2-Delta Stride are similar predictors that calculate the predicted value based on the last value and the delta between the last two. These predictors add the delta to the last value and produce the predicted value. The difference between these two predictors is only in the update process. The Stride Predictor always updates the delta based on the last two values. The 2-Delta Stride Predictor updates the delta only when the same delta occurs twice in a row. The most simplified predictor is the Last Value Predictor, which can be imagined as a Stride Predictor with a delta equal to zero. It means that it always returns the last value.

The Load Value Predictor is specialised for load instructions. Similarly to the Last Value Predictor, this predictor returns the last value. It has a mechanism that tracks only load instructions and keeps values loaded into destination registers. It stores a pair consisting of the load instruction’s address (PC) and loaded value. When the same load instruction comes again to execute, the predictor returns the value that was saved the last time for a particular destination register.
The Finite Context Method Predictor consists of two main parts: a value history table (VHT) and a value prediction table (VPT). VHT keeps contexts, each consisting of previously seen values. To select one entry, VHT is accessed with the architectural state, usually the instruction address. Further, context, i.e., the selected VHT entry and the architectural state, are inputs for a hash function that forms an index to access VPT. In that way, VPT, which contains a previously seen value for the particular context, gives a predicted value.

The Enhanced VTAGE Enhanced Stride (EVES) Predictor is a hybrid predictor that consists of two predictors: the Enhanced Stride Predictor (ESTRIDE) and the Enhanced VTAGE (EVTA GE) Predictor. The central part of EVES represents a confidence mechanism based on an algorithm that maintains confidence depending on the expected prediction benefit/loss. The VTAGE Predictor uses the instruction address and the branch history to form different indexes for access to the tables with previous values. Different indexes form from the instruction address and a different number of bits of global branch history. If there are hits in several tables, VTAGE chooses the value from the table accessed by the index formed from the most branch history bits. If both predictors, EVTAGE and ESTRIDE, have strong confidence in their predictions, EVES gives the final prediction from VTAGE. If the confidence of both is weak, then EVES gives up on predictions.

The Context-Base Computational Value Predictor (CBC-VTAGE) is a predictor whose base is the branch predictor TAGE [25]. The CBC-VTAGE Predictor accesses the tables in the same manner as the VTAGE Predictor. One table entry tracks the last value and its strides with correlated context. When some entry is hit, this predictor returns the last value increased by its stride. In that way, this predictor exhibits both computational and contextual-based properties.

The History-Based Highly Reliable Hybrid Value Predictor (H3VP) is a hybrid predictor consisting of three different predictors with a shared history table. This predictor’s authors classify three data types: arithmetic, two-periodic, and three-periodic. Each of the three predictors corresponds to one type of data. H3VP combines the predictions from all three predictors to determine the final prediction.

3. Preliminary Observations and Motivation

This section outlines the instructions considered for imprecise value prediction and provides an insightful explanation of the underlying motivation. Furthermore, this section examines the types of operands that have been observed to be suitable for prediction.

3.1. Observed Instructions

We want to observe an instruction with two operands that can keep the same result even when the value of one of the operands changes. Such an instruction can be described as follows:

\[ R_A = O1 \text{ operation } O2_A, \]  
\[ R_B = O1 \text{ operation } O2_B, \]

where instruction results \( R_A \) and \( R_B \) could be equal even though operands \( O2_A \) and \( O2_B \) have different values. This instruction feature can be used in cases where it is necessary to predict one operand. The first operand \( O1 \) is the known operand (register or immediate value), and operands \( O2_A/O2_B \) are the operands that originate from memory. Formula (1) presents the situation when the instruction executes with both correct operands, \( O1 \) and \( O2_A \). In Formula (2), the second operand \( O2_B \) has a predicted value, which may be an imprecise value, but the instruction result \( R_B \) remains the same as result \( R_A \). Depending on the operation, many imprecise values for the predicted operand \( O2_B \) still yield the correct instruction result.

It is necessary to define exactly what the instruction result is, i.e., the effect of the instruction execution. The effect can be writing a value into the destination operand \( [dst = src1 \text{ opr src2}] \) and/or affecting some status flags \([\text{affects flags}]\).
We choose to observe instructions that perform logical operations \textit{AND} and \textit{OR} with the effect of both writing a value into the destination operand and affecting some status flags (\([\text{dst} = \text{src1 opr src2}] \&\& [\text{affects flags}]\)). Also, we observe instructions that perform the logical operation \textit{AND} and subtraction, which only affect some status flags (\([\text{src1 opr src2}] \&\& [\text{affects flags}]\)).

For example, suppose we observe an instruction that performs the logical operation \textit{AND} (\([\text{dst} = \text{src1 and src2}] \&\& [\text{affects flags}]\)) and the first source operand is known. In that case, we do not need to know the value of every bit in the second source operand to produce a correct result of the instruction. Therefore, for the logical operation \textit{AND}, if a known operand has zeros at some bits, the second operand at the same bit positions can arbitrarily have ones or zeros, and the result will still be correct. The same applies to instructions that perform the logical operations \textit{OR} (\([\text{dst} = \text{src1 or src2}] \&\& [\text{affects flags}]\)). Bits whose value is one in a known operand in the other operand at the same bit positions can have arbitrary values, and the result will be correct for the logical operation \textit{OR}. Because of that, the VP does not need to predict the correct value for the operand. As we explained, some bits in the predicted operand do not affect the instruction result.

Representatives of the instructions whose result only affects some status flags are instructions that perform the logical operation \textit{AND} (\([\text{src1 and src2}] \&\& [\text{affects flags}]\)) or subtraction (\([\text{src1} - \text{src2}] \&\& [\text{affects flags}]\)). If they perform the logical operation \textit{AND}, the situation is the same as the instruction with operation \textit{AND} (\([\text{dst} = \text{src1 and src2}] \&\& [\text{affects flags}]\)), but they do not have a destination operand. Therefore, these instructions do not need the entirely correct value for the predicted operand in order for their results to be correct.

The situation with instructions that compare two operands, where one of the operands is known and the second has to be predicted, is similar. Such an instruction only performs subtraction to affect some of the status flags. In some cases where there is a need to compare two operands to determine which is greater or lesser, there is no need for the predicted operand to have the entirely correct value. It is sufficient to be greater or lesser than a known operand. These instructions may be followed by a conditional instruction that would not use all set flags, so it is sufficient to set correctly only the flags used by such an instruction. Also, in some cases, not all flags that are used by conditional instructions must have the correct value. This case occurs when a condition, tested by a conditional instruction, is the function of multiple flags. For that conditional instruction and correct program execution, the only important thing is that the condition has the correct value. We named the value of the condition a useful result.

### 3.2. Known and Memory Operands

The known operands are specified as registers or immediate values. Therefore, their values are immediately available to the processor. Because of that, there is no need to predict the value of these operands.

Operands whose values are not immediately available to the processor, which causes stall cycles, are values that originate from memory. VPs can be employed to predict these operands. We differentiate two scenarios in which operands originate from memory:

- **The first scenario is when the operand is specified with direct memory addressing. It means that the instruction has to fetch the operand from memory, making it the true memory operand (T\_MEM). If there is a cache miss, the processor must access the main memory to fetch data. This situation can cause stall cycles;**

- **The second scenario occurs when the instruction uses a register as the operand, but its value originates from memory, making it the register memory operand (R\_MEM). For example, some load instructions store values into the register, and some following instructions would use that register as the operand. Suppose no one instruction would use the value of that register between the load instruction and the following instruction. In that case, we can say that the following instruction uses a memory operand. If the following instruction is near the load instruction, there is a chance that the load**
instruction will not complete memory access before the execution of the following instruction. This situation, just like the first one, can cause stall cycles.

We will be using the ANY_MEM operand as a common name for memory operands in general, i.e., both the true memory operand (T_MEM) and the register operand that contains a value previously loaded from memory (R_MEM).

4. Predicting the Memory Operand

This section describes three models that represent the execution of instructions with ANY_MEM operands previously described in Section 3. The first model represents execution without value prediction. The remaining two models represent the execution with value prediction of operands. Models are in the form of analytical models and are described with formulas that describe the execution time. For all models, we will define mathematical expectations that represent the execution time.

The nodes represent states in which execution can be. A time representing the duration of state (time needed for transition to the next node) has been added above the edges. Figure 1 shows an example of a model with four states. Also, when there is a possibility of transition from one node to several others, the probabilities of transition to each of those nodes are listed below the edges (in Figure 1, the transition from state $b$ to states $c$ and $d$). When it is possible to move from a node to only one node, the transition probability is 1, and this probability is not shown in the model figures (in Figure 1, the transition from state $a$ to state $b$).

![Figure 1. Example of the execution model.](image)

4.1. Execution without Any Value Prediction of Operands

The first model represents an execution scenario of an instruction with the ANY_MEM operand when value prediction is not employed at all. Without value prediction, such an instruction has to wait for memory access completion. Figure 2 shows this model.

![Figure 2. Execution without value prediction.](image)

Since it is not known what types of instructions will be executed after instruction $e_1$, nor is the model tied to any specific architecture, the average duration of each instruction is adopted to be constant. If a specific system is considered, the average duration of instruction can be determined empirically. In the proposed models, the average duration of instruction is denoted by $t$ and is called a cycle in the rest of the paper.
Node $e_1$ is the instruction with the ANY_MEM operand. It has to wait for the operand to arrive from the memory, which could be due to its own initiation of a memory operand fetch or due to waiting for the completion of a previously started memory operand fetch. The waiting time is denoted as $t_{mem}(n)$, where argument $n$ represents how many cycles the instruction $e_1$ actually needs to wait for the memory operand. This means that the memory delay is modelled with a time equal to the time required to execute a certain number of instructions of average duration $t$. When the operand arrives, the execution of the other instructions $[e_2\ldots e_k]$ can continue. Probability $p(n)$ tells how likely it is that the operand will arrive from memory after precisely $n$ cycles.

If we observe some value for $n$ ($1 \leq n \leq k$), then we can define Formula (3) that describes the execution time of $[e_1\ldots e_k]$ instructions. The value $k$ is the number of cycles needed to complete a single data fetch from memory in general. Note that the argument $n$ can thus be smaller than value $k$ when some former load instruction fetches the value that instruction $e_1$ will subsequently use from the register (R_MEM operand). In a scenario where the instruction $e_1$ has a true memory operand (T_MEM operand), in a scenario where the instruction $e_1$ has a true memory operand (T_MEM operand), $n$ is equal to $k$.

$$t_c(n) = t_1 + t_{mem}(n) + \sum_{i=2}^{k} t_i = t_{mem}(n) + \sum_{i=1}^{k} t_i$$  \hspace{1cm} (3)

As mentioned, $t_{mem}(n)$ is the time needed for the operand to arrive from memory, and Formula (4) defines it, in which $t$ is the average duration of instruction (cycle). As the memory delay is equal to the time required to execute a certain number of instructions, the sum $S_k$, representing that time can be defined by Formula (5). Also, the sum $S_k$ is defined, which represents the total execution time of $[e_1\ldots e_k]$ instructions if the operand for $e_1$ is immediately available. Based on Formulas (3), (4), (6), and (5), we can write the final Expression (7) for the execution time of $[e_1\ldots e_k]$ instructions.

$$t_{mem}(n) = nt$$  \hspace{1cm} (4)

$$S_n = \sum_{i=1}^{n} t_i$$  \hspace{1cm} (5)

$$S_k = \sum_{i=1}^{k} t_i$$  \hspace{1cm} (6)

$$t_c(n) = S_n + S_k$$  \hspace{1cm} (7)

We can finally define mathematical expectation based on the probability $p(n)$ and on Formula (8) as follows:

$$E(t_c(n)) = \sum_{n=1}^{k} p(n)t_c(n) = \sum_{n=1}^{k} p(n)S_k + \sum_{n=1}^{k} p(n)S_n = S_k + \sum_{n=1}^{k} p(n)S_n$$  \hspace{1cm} (8)

4.2. Execution with Entirely Correctly Predicted Operands

The second model represents an execution scenario with value prediction where VP is employed to predict ANY_MEM operands. In this model, only correctly predicted operands are acceptable. Figure 3 shows such a model. The nodes $[s_1\ldots s_k]$ represent the speculative execution of instructions $[e_1\ldots e_k]$. Instruction $e_1$ has an ANY_MEM operand; it does not wait for the ANY_MEM operand to be fetched from memory but rather continues as speculative $s_1$ execution using the predicted operand. After some $n$ cycles ($1 \leq n \leq k$) with the probability $p(n)$, the true value for the predicted operand is available, i.e., memory access is completed. Observing the value fetched from memory, the processor can perform validation of the previously made value prediction. In the event of a hit, execution continues from instruction $e_n+1$ to $e_k$. Otherwise, in the event of a miss, the processor has to restore the previous architectural state. The procedure of the architectural state restoration is modelled...
with a penalty state with a duration of $t_{pen}$. In this state, it is necessary to restore the values of the architecture registers and the state of the memory before the start of speculative execution. As the model does not refer to any specific architecture, and the time required to perform the recovery process depends on the specific architecture, the general notation $t_{pen}$ is left to figure in the presented models. After recovery, the execution continues from instruction $e_1$ to $e_k$ using the correct value fetched from memory.

![Figure 3. Execution with entirely correctly predicted operand.](image)

We can define Formula (9) that describes the required time $t_{pOpr}(n)$ for the execution of the instructions $[e_1 \ldots e_k]$ depending on VP’s accuracy (the probability of hit operand $p_{hitOpr}(n)$ and miss operand $p_{missOpr}(n)$). The parameter $n$ represents how many cycles after $s_1$ the operand will be available.

$$t_{pOpr}(n) = \sum_{i=1}^{n} t_i + p_{missOpr}(n)(t_{pen} + \sum_{i=1}^{k} t_i) + p_{hitOpr}(n) \sum_{i=n+1}^{k} t_i = (9)$$

$$p_{missOpr}(n) = 1 - p_{hitOpr}(n) = (10)$$

If we apply Formula (10), we can simplify the expression in Formula (9) as follows:

$$t_{pOpr}(n) = \sum_{i=1}^{n} t_i + (1 - p_{hitOpr}(n))t_{pen} + \sum_{i=1}^{k} t_i + p_{hitOpr}(n) \sum_{i=n+1}^{k} t_i$$

$$t_{pOpr}(n) = \sum_{i=1}^{n} t_i + (1 - p_{hitOpr}(n))t_{pen} + \sum_{i=1}^{k} t_i - p_{hitOpr}(n) \sum_{i=n+1}^{k} t_i$$

$$t_{pOpr}(n) = \sum_{i=1}^{n} t_i + (1 - p_{hitOpr}(n))t_{pen} + \sum_{i=1}^{k} t_i - p_{hitOpr}(n) \sum_{i=n+1}^{k} t_i$$

$$t_{pOpr}(n) = \sum_{i=1}^{n} t_i + (1 - p_{hitOpr}(n))t_{pen} + \sum_{i=1}^{k} t_i - p_{hitOpr}(n) \sum_{i=n+1}^{k} t_i$$

$$t_{pOpr}(n) = \sum_{i=1}^{n} t_i + (1 - p_{hitOpr}(n))t_{pen} + \sum_{i=1}^{k} t_i - p_{hitOpr}(n) \sum_{i=n+1}^{k} t_i$$

$$t_{pOpr}(n) = \sum_{i=1}^{n} t_i + \frac{1}{(1 - p_{hitOpr}(n))t_{pen} + \sum_{i=1}^{k} t_i}$$

$$t_{pOpr}(n) = \sum_{i=1}^{n} t_i + p_{missOpr}(n)(t_{pen} + \sum_{i=1}^{k} t_i)$$
As previously mentioned, we observe instructions whose result can be correct even though the purpose of the penalty state is the same as in the model in Figure 3. The difference between the execution of instruction and matches the result calculated with the correct operand. Notably, these two results can be equal even if the predicted and correct operands are not equal for the instructions described in Section 3. In contrast, the second model does not require re-execution nodes for re-executing the instruction once the correct operand becomes available. Thus, for the second model, if the operand is correctly predicted, then the instruction result is correct.

Furthermore, new nodes have been introduced within this model to represent the re-execution of instruction \( s_1 \). This particular instruction necessitates re-execution to produce the instruction result with the correct operand once the correct operand becomes available. This step is necessary to verify whether the result calculated with the predicted operand matches the result calculated with the correct operand. Notably, these two results can be equal even if the predicted and correct operands are not equal for the instructions described in Section 3. In contrast, the second model does not require re-execution nodes because only correctly predicted operands are acceptable. Thus, for the second model, if the operand is correctly predicted, then the instruction result is correct.

Similarly, as we defined Formula (9) for execution with an entirely correct predicted operand, we can define Formula (13) for the situation presented in Figure 4. Also, the purpose of the penalty state is the same as in the model in Figure 3. The difference between these two formulas is the one extra time \( t_{re} \) that exists in Formula (13). This time is needed for re-executing the instruction \( s_1 \) when the correct operand is available.

\[
t_{pOpr}(n) = S_k + (t_{pen} + S_n)p_{missOpr}(n)
\]  

(11)

The mathematical expectation for this model based on the probability \( p(n) \) and on the final Formula (11) is presented in Formula (12).

\[
E(t_{pOpr}(n)) = \sum_{n=1}^{k} p(n)t_{pOpr}(n) = S_k + \sum_{n=1}^{k} p(n)p_{missOpr}(n)(t_{pen} + S_n)
\]  

(12)

4.3. Execution with Imprecisely Predicted Operands

The third model, depicted in Figure 4, incorporates value prediction for the ANY_MEM operands. However, in this model, it is not imperative to predict operand values correctly. As previously mentioned, we observe instructions whose result can be correct even though the ANY_MEM operand does not have the correct values. Because of this, this model integrates the probability of correct instruction results (\( p_{hitRes} \)) instead of the probability of the correct operand (\( p_{hitOpr} \)) as in the second model.

**Figure 4.** Execution with imprecisely predicted operand.

\[
E(t_{pRes}(n)) = \sum_{i=1}^{n} t_i + p_{missRes}(n)(t_{re} + t_{pen} + \sum_{i=1}^{k} t_i) + p_{hitRes}(n)(t_{re} + \sum_{i=n+1}^{k} t_i)
\]  

(13)
The final Formula (14) is obtained by arranging the formula in a similar manner to Formula (9). Based on it and on the probability \( p(n) \), Formula (15) presents the mathematical expectation for this model.

\[
\begin{align*}
t_{pRes}(n) & = S_k + t_{re} + (t_{pen} + S_n)p_{missOpr}(n) \\
E(t_{pRes}(n)) & = \sum_{n=1}^{k} p(n)t_{pRes}(n) = S_k + t_{re} + \sum_{n=1}^{k} p(n)p_{missRes}(n)(t_{pen} + S_n)
\end{align*}
\] (14) (15)

4.4. Research Questions

Our research aims to address several questions about imprecise value prediction when utilising VP to predict ANY_MEM operands of instructions described in Section 3:

- **Question 1**: Is it possible for the result of an instruction to be correct even though the predicted operand is not correctly predicted?
- **Question 2**: Can the useful result of an instruction be correct even though the predicted operand is not correctly predicted?
- **Question 3**: Does the accuracy of VP vary depending on the types of instructions discussed in Section 3?
- **Question 4**: Does the accuracy of VP vary depending on the types of operands involved, specifically distinguishing T_MEM operands from R_MEM operands?
- **Question 5**: Does the accuracy of the correct instruction result based on the predicted operand \( P_{hitRes} \) surpass the accuracy of the correctly predicted operand \( P_{hitOpr} \)?
- **Question 6**: Finally, is the mathematical expectation \( E(t_{pRes}) \) lower than \( E(t_{pOpr}) \)? In other words, under what conditions does the third model outperform the second model regarding execution time?

5. Methodology and Simulation

This section details the simulations conducted, focusing on the developed simulator and the upgrades made to the gem5 simulator. Additionally, it provides comprehensive insights into the benchmarks employed, specifically analysing the suitable operands for prediction.

5.1. Brief Overview of Simulations

In this work, we observe the x86 [26] architecture to evaluate imprecise value predictions. For the instructions described in Section 3, we notice four instructions: `cmp`, `test`, `and` and `or`—we named these four instructions CTAO. As we explained earlier, VPs will be used only for instructions where exactly one of the operands is the ANY_MEM operand. Therefore, in our experiments, we track these four instructions with one ANY_MEM operand.

We used the EEMBC benchmark [27] and a subset of the SPEC CPU2006 benchmark [28,29]. We upgraded the gem5 simulator [30–33] with a module that produces execution traces of the benchmarks. We implemented a Value Prediction Simulator (VPSim) that performs trace-driven simulation and collects statistics consisting of a large amount of data. Our VPSim incorporates the value predictors that were previously described in Section 2.3.

5.2. Benchmarks

CoreMark-Pro benchmark is one of the benchmarks belonging to the Embedded Microprocessor Benchmark Consortium (EEMBC). It contains nine workloads for testing processors, from low-end microcontrollers to high-performance computing processors [34]. It has five integer and four floating point workloads. All workloads are written in the C programming language.

The SPEC CPU2006 benchmark, which belongs to Standard Performance Evaluation Corporation (SPEC), is used for stressing a system’s processor, memory subsystem and compiler [29]. It contains workloads developed from real user applications. We used a
subset of workloads written in pure C/C++ language (without Fortran language) that can be compiled without source code modifications using newer gcc compiler versions (9.3+).

Table 1 shows the workloads that are used in our work. We used the gcc compiler to obtain executables from the workload’s source code. Those executables were executed in the upgraded gem5 simulator in order to produce execution traces. Traces that were 100 M instructions long were obtained from each workload execution.

Table 1. Overview of workloads used.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Integer Workloads</th>
<th>Floating Point Workloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoreMark-Pro</td>
<td>JPE compression, ZIP compression, XML parsing, SHA-256, CoreMark</td>
<td>Radix-2 Fast Fourier Transform, Gaussian elimination, Neural-net, Livermore loops</td>
</tr>
<tr>
<td>SPEC2006</td>
<td>473.astar, 464.h264ref, 462.libquantum, 429.mcf, 458.sjeng</td>
<td>433.milc, 470.lbm</td>
</tr>
</tbody>
</table>

CTAO instructions comprise 1% to 22% of all instructions in obtained traces, with an average of approximately 10%. Figure 5a illustrates the percentage of CTAO instructions with ANY_MEM operands from all CTAO instructions in traces, individually for instructions cmp, test, and or. The diagram reveals that SPEC benchmarks have a higher proportion of CTAO instructions with ANY_MEM operands than the CoreMark-Pro benchmark.

Figure 5b presents the distribution of CTAO instructions with ANY_MEM operands across used traces. Among these instructions, cmp and test instructions appear to be the most prevalent. Following them, and instructions are the next most common, while or instructions are the least frequent.

We shed light on the ones and zeros distribution in a binary notation of the known operands in the obtained traces. Figure 6a,b show the number of bits that must be correctly predicted for the ANY_MEM operand for instructions and or. The rest bits, equal to zero, of the known operand for these instructions do not affect the instruction results. Over 70% of test instructions and over 50% of and instructions with the ANY_MEM operand have the known operand with up to a max of 8 bits with value one. For both instructions, as we can see in Figure 6a,b, about 40% of these instructions have a known operand of none or only one bit with a value of one. That means that in 40% of these instructions, the VPs must correctly predict at most one bit of the ANY_MEM operand for the instruction result to be correct.
Furthermore, Figure 6c provides insight into the number of the known operand’s bits with value one for or instruction. Approximately 90% of or instructions have known operands, with a maximum of one bit having a value of one. This observation implies that most known operand values possess a considerable number of zeros; these instructions are commonly utilised as masks to set specific bits in other variables. Additionally, the authors are aware that the gcc compiler exclusively generates the instruction or when the logical operation OR is explicitly expressed in a high-level programming language. Consequently, among the four instructions considered, the instruction or is the least frequently encountered in the obtained traces.

5.3. Upgrading gem5 Simulator

The gem5 simulator is a simulation framework that supports multiple instruction set architectures (Alpha, ARM, MIPS, Power, SPARC, RISC-V and x86). Also, it supports several CPU and memory models that provide different simulation capabilities, balancing simulation speed and accuracy [31]. The simulator was chosen for its open-source nature, allowing for potential modifications [35]. It is noteworthy that the chosen simulator is actively supported.

We used the AtomicSimple CPU model in System-call Emulation (SE) mode. The AtomicSimple is an in-order single instruction per cycle CPU model. It employs atomic memory access, ensuring immediate completion of all memory accesses. SE mode emulates the most common Linux system calls and does not involve booting an operating system, contrary to the FullSystem mode. As a result, SE mode is well-suited for observing only the CPU without the operating system, leading to increased simulation speed. Given these considerations, we choose to use the AtomicSimple CPU in the SE mode.

We implemented a new module into the gem5 simulator to intercept instruction execution, enabling the generation of execution traces with the necessary information. For each instruction, this module records the instruction’s name, the current and next values of the PC register, the value of flags after instruction execution and whether the instruction is a jump instruction. Besides this information, during the execution phase of instructions, the module also keeps track of architectural registers and memory locations that were accessed for reading and writing. For architectural registers, the module saves a pair consisting of the register’s name and its corresponding value. Likewise, for memory locations, a pair consisting of the address and value read or written is saved. If an instruction has an immediate value as the operand, the value is also included in the trace.

5.4. VPSim

VPSim performs trace-driven simulations. It consists of a simple processor core and a value predictor. The processor core has architectural registers, and the component responsible for calculating flags based on the instruction’s type and its operands. Algorithm 1 illustrates the pseudo-code of VPSim internals for performing trace-driven simulation.
First, VPSim forms instructions based on information from the trace. Subsequently, it verifies whether the formed instruction is a CTAO instruction with the ANY_MEM operand. In the affirmative case, the value predictor makes a prediction for the instruction. Lastly, the update process is performed. Architectural registers are updated based on the destination register’s value from the instruction. If the prediction was made, then VP is also updated based on the true value of the predicted operand. Before progressing to simulate the next instruction, VPSim performs a statistics update.

**Algorithm 1** Pseudo-code of simulation

```plaintext
while (trace.has_instruction())
    do
        instruction = trace.get_next();
        prediction = null;
        if (instruction.is_CTAO())
            prediction = VP.predict();
            calculate_instruction_result(prediction)
            begin
                perform the operation on predicted and known operand;
                calculate flags based on operation’s result;
            end;
            VP.update(instruction, prediction);
        end if;
        update_architectural_state(instruction);
        update_statistics(instruction, prediction);
    end while;
```

In order to determine the correctness of the simulation, the observed instructions (cmp, test, or, and) were executed in the VPSim simulator, first without any prediction and only based on the correct values of the operands obtained from the benchmark traces described in Section 5.2. As said, the trace also contains the result of the instruction (the value of the destination operand and the flag register), so that result was used to compare with the obtained result of executing the instruction in the simulator. In this way, the correctness of the execution of the instructions in the implemented VPSim simulator was determined.

Within VPSim, two instances of the flags register are maintained: the true flags register and the predicted flags register. The true flags register preserves the actual flags’ value obtained from the trace. After each instruction that sets the flags, the true value is updated based on the flags’ value from the trace record. The predicted flags register stores the calculated value for flags based on the predicted operand. Additionally, the simulator keeps track of the most recent instruction that sets the flags. When a conditional instruction uses the flags, VPSim examines which instruction was responsible for the most recent setting of the flags. If the flags were set by CTAO instructions, VPSim performs the following check. It compares the outcome of a condition formed using the predicted flags register with one formed using the true flags register. This allows VPSim to track instructions that rely on the result (flags) of CTAO instructions and determine whether the condition’s outcome is correct based on predicted flags. As mentioned above, conditional instruction does not use all flags from the flags register. In certain cases, the condition’s outcome can be correct, regardless of whether all flags are correct.

Regarding the architectural registers, specifically the general-purpose registers, VPSim tracks whether the most recent modification was made by a load instruction. Suppose some CTAO instruction uses such a general-purpose register, and no one instruction accesses that register between load instructions and CTAO instructions. In that case, it means that the CTAO instruction uses the R_MEM operand.

VPSim contains seven value predictors: Last Value (LAVP), Load Value (LOVP), 2-Delta Stride (TDS), FCM, EVES, CBC-VTAGE and H3VP. The first four predictors predict each time due to a lack of confidence mechanism. They exhibit lower accuracy but have been
chosen as well-known predictors. The last three predictors heavily rely on the confidence mechanism, which boosts their accuracy but lowers coverage. The last three predictors rely heavily on the confidence mechanism, which raises their accuracy at the cost of lower coverage. VPSim is implemented as a command line tool and accepts a single parameter representing a configuration file. This file specifies the chosen value predictor, its configuration and paths to the execution traces. When the simulation is completed, VPSim produces output files containing collected statistics, with each trace having its own output file.

The described Algorithm 1 of trace-driven simulation aims to follow the behaviour of the predictor without going into the organisational details of the processor itself and its architecture. The main goal of this algorithm is to collect statistics on operand predictions in order to show the phenomenon that it is possible to reach the correct result of the instruction based on an imprecisely predicted operand. Looking at the part of Algorithm 1 that only refers to predicting operands and monitoring whether the result of the instruction is correct even with an imprecise predicted operand, the most important aspects of the algorithm can be singled out:

- The algorithm considers that even an imprecisely predicted operand is acceptable if the result of the instruction is correct, thus avoiding a recovery from misprediction;
- The algorithm considers whether the value in one of the registers originates from memory, which makes that register a R_MEM operand type;
- The algorithm introduces another instance of the flags register that stores the set flags based on the speculatively executed instruction.

Guided by the listed aspects, one can comment on the possibility of incorporating this way of predicting operands in processors with out-of-order execution, for example, on architectures based on Tomasulo’s algorithm. With such processors, instructions whose execution is independent of each other can be executed on free-executing units in an arbitrary order. An instruction suitable for prediction, meaning that it is of the appropriate type and does not currently have a value available for its operand, can be issued for execution even though its operand is not available. Then, it is inserted with the predicted operand into the corresponding execute unit. At the same time, marking it as a speculative instruction with the predicted operand is necessary. It can therefore be immediately executed with the predicted operand as soon as the execute unit becomes available. All instructions that directly or indirectly depend on its result should also be marked as speculatively executed, and in this way, a chain of speculative instructions would be formed according to the proposed model. In addition, for each dependent instruction, it is necessary to save which part of the result of the instruction whose operand is predicted has been used. When the correct operand becomes available, it is necessary to re-execute the instruction for which the operand was predicted, now with the correct operand. Suppose the used result of the instruction with the predicted operand and now with the correct operand is the same. In that case, it is only necessary to declare that all speculatively executed instructions are correct and that it is possible to commit them. In a situation where there is an instruction in the chain of dependent instructions for which a prediction was also made, only instructions that do not depend on it may be committed. In the case of a misprediction, when the used result of the instruction with the predicted operand and now with the correct operand is not the same, the aforementioned reissue recovery mechanism should be applied. Then, only instructions dependent on the instruction for which operand prediction was performed would be reissued for execution. The chain of instructions in our model is the same as in the case of in-order execution. The only difference is that in the case of out-of-order execution, there would only be instructions directly or indirectly dependent on the CTAO instruction for which the prediction was made because other independent instructions can be executed outside of this flow. In contrast, the chain of instructions during in-order execution can contain both dependent and independent instructions from the speculative CTAO instruction in the same order as in the assembler code.
6. Results and Discussion

This section addresses the previously defined research questions and provides comprehensive answers based on the simulations conducted. Also, the simulations themselves are described, outlining the specific experimental setups employed. Furthermore, this section contains threats to validity.

6.1. Correct Instruction Result with Imprecisely Predicted Operand

We aim to examine whether the result of CTAO instructions with an imprecisely predicted ANY_MEM operand could be correct. As mentioned earlier, it is not essential to predict the operand correctly for these instructions in order for their result to be correct. This characteristic distinguishes them from other types of instructions, where correct operand prediction is necessary for ensuring correct results.

We conducted trace-driven simulations on the previously described traces, whose length is 100 M instructions. Simulations utilised the implemented VPSim, where VPs are used to predict ANY_MEM operands for CTAO instructions. Each VP is configured to use up to 8 KB of storage for its structures. We performed simulations across the entire set of traces for every single VP described in Section 2.3. Throughout the simulation, we monitored a batch of parameters. The notable parameters are the number of predictions made (predicted), the number of correctly predicted operands (hitsOpr), and the number of correct instruction results calculated based on the predicted operand (hitsRes).

Figure 7 shows two diagrams pertaining to CTAO instructions. Figure 7a represents the accuracy of a correctly predicted operand, while Figure 7b depicts the accuracy of the correct instruction result calculated based on the predicted operand. The names of traces are indicated along the perimeter of the diagrams. The contour of a particular colour represents the accuracy of the particular predictor. Notably, in diagram (b), all contours are closer to the diagram’s perimeter compared to those in diagram (a). It means that the accuracy of the correct instruction result based on the predicted operand (\(P_{hitRes} = hitsRes/predicted\)) is higher than the accuracy of the correctly predicted operand (\(P_{hitOpr} = hitsOpr/predicted\)), which is the answer to research Question 5 (Does the accuracy of the correct instruction result based on the predicted operand \(P_{hitRes}\) surpass the accuracy of the correctly predicted operand \(P_{hitOpr}\)?)

Figure 7. (a) Accuracy of correctly predicted operand for CTAO instructions; (b) accuracy of correct result based on predicted operand for CTAO instructions.

The number of correct instruction results (hitsRes) is the sum of the number of correctly predicted operands (hitsOpr) and a number of situations where the operand is not correctly predicted, but the instruction result is correct (missOprHitsRes). Based on PhitRes > PhitOpr, it can be concluded that missOprHitsRes > 0. Therefore, it means that the result of the instruction can be correct even though the predicted operand is not correctly predicted,
which is the answer to research **Question 1** (Is it possible for the result of an instruction to be correct even though the predicted operand is not correctly predicted?).

During the simulation, we tracked which conditional instructions used flags previously set by the instructions `cmp` and `test` (CT instructions). We monitored situations in which the useful result of CT instructions was correct. As previously mentioned, the useful result is the value, i.e., the outcome of the condition for conditional instruction. Figure 8 shows two diagrams for CT instructions. Figure 8a showcases the accuracy of a correctly predicted operand, while Figure 8b represents the accuracy of a correct useful result of an instruction. Figure 8a is very similar to Figure 7a because CT instructions are more widespread than `and` and `or` instructions. The contours in Figure 8b are very close to the perimeter of the diagram, which means that the accuracy of the correct useful result is very high. It is interesting to note that even simple predictors achieved high accuracy of correct useful results. Consequently, in response to research **Question 2** (Can the useful result of an instruction be correct even though the predicted operand is not correctly predicted?), we conclude that the useful result can indeed be correct despite the predicted operand not being correctly predicted. This is attributed to the fact that the accuracy of the correctly predicted operand is lower than that of the useful correct result.

![Figure 8](image)

**Figure 8.** (a) Accuracy of correctly predicted operand for CT instructions; (b) accuracy of correct useful result for CT instructions.

### 6.2. Accuracy of VPs Depending on Operand Types and Instruction Types

To obtain an answer to research **Question 3** (Does the accuracy of VP vary depending on the types of instructions discussed in Section 3?), we performed simulations where predictors were configured to make predictions exclusively for one type of CTAO instruction. This allowed us to measure the accuracy of the correct instruction result calculated based on predicted ANY_MEM operands for each separate instruction (`cmp`, `test`, `and` and `or`). Figure 9a presents the accuracy of correct instruction results for each separate instruction and the accuracy when VPs made predictions for all four types of instructions (CTAO). The presented accuracies are the average accuracies that VPs achieved across traces. Some predictors have a deviation from the accuracy achieved for CTAO. This deviation exists for instructions `and`/`or` because they are not widespread across traces, and VPs need more occurrences of them to learn to make predictions for them. In general, the accuracy of VPs does not significantly differ based on the instruction type, thereby answering research **Question 3**.
Figure 9. (a) Accuracy of the correct instruction result based on the predicted operand for each instruction (cmp, test, and or); (b) accuracy of the correct instruction result based on the predicted operand for each operand type (R_MEM and T_MEM).

During the performed simulations, we also tracked the accuracy of correct instruction results for CTAO instructions that used predicted operands, individually for R_MEM and T_MEM operand types, to obtain an answer to research Question 4 (Does the accuracy of VP vary depending on the types of operands involved, specifically distinguishing T_MEM operands and R_MEM operands?). Figure 9b shows the accuracy of the correct instruction result separately for R_MEM and T_MEM operand types. As shown in Figure 9a, the accuracies are the average accuracies that VPs achieved across traces. Only two predictors, LAVP and FCM, have differences in the two curves on the diagram (about 5–10%), while the others have minor deviations between the two curves. Based on that, the accuracy of VP does not significantly depend on operand types, which is the answer to research Question 4.

6.3. Mathematical Expectations

To compare the mathematical expectations of the second and third models, which are described in Section 4, there is a need to perform subtraction of the mathematical expectations $E(t_{pOpr}(n))$ and $E(t_{pRes}(n))$ described in Formulas (12) and (15). The result of subtracting Formula (15) from Formula (12) is presented in Formula (16).

$$E(t_{pOpr}(n)) - E(t_{pRes}(n)) = \sum_{n=1}^{k} p(n)(p_{missOpr}(n) - p_{missRes}(n))(t_{pen} + S_n) - t_{re} \quad (16)$$

Based on the answer to research Question 4, which tells us that the VP accuracy does not depend on operand type, in Formula (16), it can be applied that $p_{missOpr}(n)$ and $p_{missRes}(n)$ are constants. In other words, $p_{missOpr}(n)$ and $p_{missRes}(n)$ do not depend on $n$ ($n$ represents how many cycles the instruction needs to wait for the ANY_MEM operand). Formula (16) can be transformed as follows:

$$E(t_{pOpr}(n)) - E(t_{pRes}(n)) = (p_{missOpr} - p_{missRes}) \sum_{n=1}^{k} p(n)(t_{pen} + S_n) - t_{re} \quad (17)$$

We want to observe a situation where $E(t_{pOpr}(n))$ is greater than $E(t_{pRes}(n))$, i.e., when the execution time of the third model is better than the execution time of the second model. Therefore, the expression of Formula (17) should be greater than zero, which is presented in Formula (18).

$$(p_{missOpr} - p_{missRes}) \sum_{n=1}^{k} p(n)(t_{pen} + S_n) - t_{re} > 0$$
The time it takes to re-execute the instruction for which the prediction was made can be represented as the average instruction execution time (cycle), denoted by $t$. Expected average execution time for $n$ instructions $S_n$ can be modelled with $nt$. Based on this and Formula (5), Formula (18) can be transformed as follows:

$$\left( p_{\text{missOpr}} - p_{\text{missRes}} \right) \left( \sum_{n=1}^{k} p(n)S_n + t_{\text{pen}} \right) - t_{\text{re}} > 0$$  \hspace{1cm} (18)$$

In the final Expression (19), the function $f$ is the function of two arguments, $\Delta$ and $s(k)$. The first argument $\Delta$ is the difference between probabilities $p_{\text{missOpr}}$ and $p_{\text{missRes}}$. The second argument $s(k)$ is the sum of the products of probability $p(n)$ and $n$, where $n$ belongs from 1 to $k$. Argument $k$ is the number of cycles needed to complete a single data fetch from memory. On the left side of inequality stands a ratio ($t_{\text{pen}}/t$) between the time needed to recover from misprediction and the average time needed for instruction execution.

The final expression for Formula (18) tells us that the mathematical expectation $E(t_{\text{pRes}}(n))$ is lower than $E(t_{\text{pOpr}}(n))$, while the inequality in Formula (19) holds. In other words, while a ratio $t_{\text{pen}}/t$ is greater than the value of function $f$, it means that the third model achieves better execution time than the second model while the inequality in Formula (19) holds, which is the answer to research Question 6 (Is the mathematical expectation $E(t_{\text{pRes}})$ lower than $E(t_{\text{pOpr}})$? In other words, under what conditions does the third model outperform the second model regarding execution time?).

During simulations, we measured how many times ANY_MEM operands occur across traces, separately for T_MEM and R_MEM operands. For R_MEM operands, we measured how many cycles before the CTAO instruction the operand fetch was initiated. Figure 10 shows the histogram that represents how many cycles the CTAO instruction must wait for an operand from memory without employing value prediction. Numbers representing the required cycles for completing an operand fetch from memory are shown on the horizontal axis. The number $k-0$ corresponds to the T_MEM operand, and the number $k-x$ ($x > 0$) corresponds to the R_MEM operand, which means that the operand fetch was initiated before $x$ cycles (some previous instruction initiated fetching before $x$ cycles). More than 50% of CTAO instructions have the T_MEM operand, meaning they must wait for $k-0$ cycles for the operand. The remaining ones have the R_MEM operand and must wait for $k-x$ cycles for the operand. Based on the histogram, about 99.5% of all occurrences of CTAO instructions with the ANY_MEM operand must wait up to $k-10$ cycles. Because of this, in inequality (19), we used values in the range 1 to 10 for argument $k$. On the vertical axis, the percentage of CTAO instructions is shown. This percentage is used as the probability $p(n)$, which tells how likely it is that the operand will arrive from memory after exactly $n$ cycles.
Two solutions can be used to solve the situation. The first solution is to use the current $\Delta$ value. If two occurrences of the CTAO instruction are close to each other, there is a chance that two occurrences of the CTAO instruction are close to each other, yielding a potentially problematic situation. The process of updating VP, caused by the former occurrence of the CTAO instructions, could still be ongoing when the latter occurrence of the CTAO instruction starts executing. The consequence would be that the later occurrence of the CTAO instruction would not have an updated state of the predictor. The process of updating VP is performed instantaneously after prediction. In pipelined processors, there is a chance that two occurrences of the CTAO instruction are close to each other, yielding a potentially problematic situation. The process of updating VP, caused by the former occurrence of the CTAO instructions, could still be ongoing when the latter occurrence of the CTAO instruction starts executing. The consequence would be that the latter occurrence of the CTAO instruction would not have an updated state of the predictor.

Table 2. The average difference between $p_{\text{miss\text{Op}}}$ and $p_{\text{miss\text{Res}}}$. (\(\Delta\)) that predictors achieved across traces, based on collected statistics from performed simulations, is presented in Table 2.

<table>
<thead>
<tr>
<th>Predictor</th>
<th>Average (\Delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAVP</td>
<td>0.449</td>
</tr>
<tr>
<td>LOVP</td>
<td>0.418</td>
</tr>
<tr>
<td>TDS</td>
<td>0.221</td>
</tr>
<tr>
<td>FCM</td>
<td>0.230</td>
</tr>
<tr>
<td>H3VP</td>
<td>0.011</td>
</tr>
<tr>
<td>EVES</td>
<td>0.008</td>
</tr>
<tr>
<td>CBC-VTAGE</td>
<td>0.123</td>
</tr>
</tbody>
</table>

6.4. Threats to Validity

The process of updating VP is performed instantaneously after prediction. In pipelined processors, there is a chance that two occurrences of the CTAO instruction are close to each other, yielding a potentially problematic situation. The process of updating VP, caused by the former occurrence of the CTAO instructions, could still be ongoing when the latter occurrence of the CTAO instruction starts executing. The consequence would be that the latter occurrence of the CTAO instruction would not have an updated state of the predictor. Two solutions can be used to solve the situation. The first solution is to use the current state of the predictor at the time of the arrival of the later instruction. The second solution represents speculative updating of the predictor, where the state of the predictor is updated immediately after the prediction for the former instruction based on that predicted value. In our simulations, the average number of instructions between two adjacent instructions for which a prediction is made on the used benchmarks ranges from 40 to 900. Since
the two adjacent instructions for which the prediction is made are far enough apart, this situation was not considered further.

Figure 11. The surface of function $f(\Delta, s(k))$.

Championship predictors have a powerful confidence mechanism that allows them to predict CTAO instructions with the ANY_MEM operand selectively. Consequently, they exhibit very high accuracy but limited coverage. They only predict when their confidence level is very high. Therefore, predictors H3VP and EVES have minimal differences between $p_{missOp}$ and $p_{missRes}$ (usually, when they make a prediction, they predict the operand correctly). In this scenario, where a powerful confidence mechanism and reduced coverage are present, the function $f$ approaches infinity, rendering the application of the third model unfeasible. Lowering the confidence level threshold can increase the coverage and delta, potentially creating additional opportunities for applying the third model.

7. Conclusions

The paper proposes a solution for exploiting the behaviour of imprecise value prediction when predicting operands for instructions whose results can be correct, even when the predicted operands may not have correct values. The four instructions ($cmp$, $test$, and $or$) whose one operand originated from memory were observed. Additionally, two execution models utilising value prediction were described as analytical models. One model considers only the correct prediction of the operand acceptable, while the other model considers the correct result of the instruction based on the predicted operand acceptable, even if the operand is imprecisely predicted. The proposed solution and described models were evaluated using standard benchmarks, SPEC and EEMBC. The answers to the research questions were given based on the experiments conducted.

The analysis showed that the accuracy of the correct result of the instruction based on the predicted operand, even when the predicted operand is imprecise, is higher than the accuracy of the correctly predicted operand. The accuracy improvement ranges from 0.8% to 44%, depending on the specific predictor used. Also, the accuracy of predicting an imprecise value does not depend significantly on the type of instruction or on whether the operand comes directly from memory or is previously loaded into a register from memory. The evaluation of two described execution models revealed the conditions under which the execution model with an imprecisely predicted operand, but correct instruction result, achieves better execution time than the model with the correctly predicted operand. Our findings shed light on the potential benefits and trade-offs associated with imprecise value prediction. The results highlight the feasibility of achieving correct instruction results, even with imprecisely predicted operands. It paves the way for further research into instruction
execution performance with a value prediction mechanism that prioritises the correct instruction output, regardless of the possibility of an imprecisely predicted operand.

Author Contributions: U.R.: formal analysis, investigation, methodology, software, validation, visualisation and writing—original draft; M.M.: investigation, validation, and writing—review and editing; Z.R.: conceptualisation, formal analysis, supervision and writing—review and editing. All authors have read and agreed to the published version of the manuscript.

Funding: This work was financially supported by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia, contract number: 451-03-47/2023-01/200103.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

References
13. Eickemeyer, R.J.; Vassiliadis, S. Load-Instruction Unit for Pipelined Processors. IBM J. Res. Dev. 1993, 37, 547–564. [CrossRef]
14. González, J.; González, A. Memory Address Prediction for Data Speculation. In Euro-Par’97 Parallel Processing—Euro-Par 1997; Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Springer: Berlin/Heidelberg, Germany, 1997; Volume 1300, pp. 1084–1091. [CrossRef]


Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.