uVP: Efficient implementation of value prediction via Micro-op cache

**Dan Recher**

uVP: Efficient implementation of value prediction via Micro-op cache

**Research Thesis**

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering

**Dan Recher**

Submitted to the Senate of the Technion Israel Institute of Technology

Tishri 5781 Haifa September 2020

This research was carried out under the supervision of Prof. Avi Mendelson, in the Andrew & Erna Viterbi Faculty of Electrical Engineering.

**Acknowledgments**

I would like to thank several people for helping with this research project:

1. My supervisor, Prof. Avi Mendelson, for his dedicated supervision and support. Thanks for your help, advice, irreplaceable guidance, and interesting conversations.
2. Thanks to the Technion, the Andrew & Erna Viterbi Faculty of Electrical Engineering, in which the research was carried out.
3. Finally, thanks to my wife, Or, for her patience and solidarity and for being a great wife.

Contents

[List of Figures](#_Toc54200630)

[1 Abstract 1](#_Toc54200631)

[2 Introduction 2](#_Toc54200632)

[3 Background 5](#_Toc54200633)

[4 Related works 10](#_Toc54200634)

[5 Terms and conditions 18](#_Toc54200635)

[5.1 Terms 18](#_Toc54200636)

[6 Our approach 19](#_Toc54200637)

[7 Experimental testbed 24](#_Toc54200638)

[7.1 The testbed simulation environment 24](#_Toc54200639)

[7.2 Implementation in Sniper simulator 24](#_Toc54200640)

[7.2.1 Micro-op cache 26](#_Toc54200641)

[7.2.2 Last Value Predictor - LVP 27](#_Toc54200642)

[7.2.3 VTAGE 28](#_Toc54200643)

[7.2.4 VP in Micro-op cache 29](#_Toc54200644)

[7.3 Measurements Methodology 30](#_Toc54200645)

[7.4 Workloads 30](#_Toc54200646)

[7.5 System configuration 31](#_Toc54200647)

[7.5.1 Reference - Clean Sniper environment 31](#_Toc54200648)

[7.5.2 LVP 31](#_Toc54200649)

[7.5.3 VTAGE value predictor 31](#_Toc54200650)

[7.5.4 Micro Op cache + LVP 31](#_Toc54200651)

[7.5.5 Micro Op cache + VTAGE Value predictor 31](#_Toc54200652)

[8 Experiments 32](#_Toc54200653)

[8.1 Reference environment 32](#_Toc54200654)

[8.1.1 Penalty sensitivity check 32](#_Toc54200655)

[8.1.2 Size impact 33](#_Toc54200656)

[8.1.3 SPEC2017 breakdown 34](#_Toc54200657)

[8.2 Micro-op cache characterization 35](#_Toc54200658)

[8.2.1 Micro-op cache impact on power and performance 35](#_Toc54200659)

[8.2.2 Micro-op integration impact on IPC 37](#_Toc54200660)

[9 Summary and future work 40](#_Toc54200661)

[9.1 Summary 40](#_Toc54200662)

[9.2 Ideas for future work 41](#_Toc54200663)

[10 References 42](#_Toc54200664)

[**Hebrew Abstract**](#_תקציר) [**i**](#_תקציר)

# List of Figures

[Figure 1- RAW dependencies 2](#_Toc54123902)

[Figure ‎3.1 Intel X86 instruction format, taken from intel 64 and IA-32 Architectures Software Developer’s Manual[2] 5](#_Toc54123903)

[Figure ‎3.2 Example for pipeline with Micro-op cache 6](#_Toc54123904)

[Figure ‎3.3 Dependencies between subsequent instructions 7](#_Toc54123905)

[Figure ‎3.4 - Data flow execution improvement due to using OOO execution 9](#_Toc54123906)

[Figure ‎4.1 ILP improvement due to VP prediction 11](#_Toc54123907)

[Figure ‎4.2 LVP - Last Value Predictor structure 12](#_Toc54123908)

[Figure ‎4.3 Stride predictor structure 13](#_Toc54123909)

[Figure ‎4.4 FCM predictor - 2 level predictor 14](#_Toc54123910)

[Figure ‎4.5 - VTAGE predictor structure 16](#_Toc54123911)

[Figure ‎6.1 “Virtual-link” 21](#_Toc54123912)

[Figure ‎6.2 Micro-op fields after adding Value predictor integration 22](#_Toc54123913)

[Figure ‎8.1 – Value prediction Miss penalty impact on IPC in reference environment 33](#_Toc54123914)

[Figure ‎8.2 - IPC impact by the size of the value predictor in reference environment 34](#_Toc54123915)

[Figure ‎8.3 – IPC of different 8k value predictors SPEC2017 breakdown of reference 35](#_Toc54123916)

[Figure ‎8.4 Micro-op cache IPC improvement over disabled Micro-op cache by application 36](#_Toc54123917)

[Figure ‎8.5 Micro-op cache Cdyn improvement over disabled Micro-op cache by application 36](#_Toc54123918)

[Figure ‎8.6 - IPC by type of the predictor, while integrating with Micro-op cache and without, without Micro-op cache additional IPC improvement 38](#_Toc54123919)

[Figure ‎8.7 - IPC by type of the predictor, while integrating with Micro-op cache and without, contain Micro-op cache improvement 39](#_Toc54123920)

# Abstract

The use of VP (Value Prediction) has already shown its potential benefit in enhancing the performance of OOO (Out-Of-Order) processor architectures in many studies. Thus, its complexity and additional power consumption prevent implementing it as part of current processors. In this work, we present the uVP method that aims to predict the value of operations while still preserving the internal micro-architectural features like Micro-op operations in general and in particular with the Micro-op cache. This method provides the most potential benefit of using value prediction, with manageable power, area, and complexity overhead.

This work suggests allowing value prediction to be integrated as part of the Micro-op cache without significant changes so it can leverage its power and performance benefits without disrupting the performance advantage of value prediction; it describes the new mechanism, discusses its characteristics, and compares it with a simple value prediction mechanism, named LVP (last value prediction) and state-of-the-art mechanisms, such as VTAGE which uses context-based methods like using global branch history. We will show that the new proposed novel integration enables an efficient implementation of value prediction without any significant performance loss in comparison with existing VP implementation by implementing it in the Sniper simulator and running SPEC2017 benchmarks.

# Introduction

There is a growing demand for increasing the performance of computer systems. Such a demand can be achieved either by using many simple cores (many core architectures) or improving the performance of single cores. The traditional method for increasing single-thread performance is either by increasing the frequency via pipeline and better process technology or by increasing the IPC (Instructions Per Cycles). Those ways could be shown in the following equation:

Equation 1 .

The use of OOOE (Out Of Order Execution) based architectures has been proven to be a very effective way of increasing the IPC of a given machine. It was developed in the mid-1990s and has been widely implemented in modern CPUs since. The OOOE machine enables better utilization of the execution units and supports running parallel independent instructions in the same cycle, increasing IPC and optimizing silicon use.

The RAW (Read-After-Write) dependencies and control hazards could impair OOOE capabilities; those dependencies occur when an instruction depends on the calculated value from previous instructions. For example:

R1 = R1 + 4 R3 = LD(r5)

R3 = R1 +2 R4 = R3 + R8

1. (b)

Figure 1- RAW dependencies

Figure 1 presents two sequences of RAW dependencies. Figure 1.a presents a RAW dependency between arithmetic operations, while Figure 1.b shows how the calculation needs to wait for a value being loaded from memory. Such a memory operation can last hundreds of cycles if the main memory needs to be accessed; thus, the entire execution sequence may be postponed accordingly.

Value Prediction (VP) has been suggested to break RAW dependencies, support OOOE executing instructions in a more parallel way, increase ILP, which, as a result, further increases IPC. VP is used to predict the result value of some instruction allowing dependent instruction to be executed before the value is being calculated by exploiting the value locality of the program. As prediction is the value being used, all instructions with this value are running speculatively and should not be committed until the value is validated with the result of the predicted instruction. If the predicted value is correct, all dependent instruction can also commit, gaining the speedup from the early execution. However, when the speculation is bad, all speculated instruction should re-execute with the correct value at a penalty to the overall program. To ensure the predictor is effective, the average gain of using it should be larger than the average of the miss penalty resulting to an increase in IPC.

One of the major hurdles for implementing value prediction in modern processors is its cost in terms of area, power consumption, and the potential conflict with other architectural features, such as the Micro-op cache. For example, when the processor is fetching instructions from the Micro-op cache, it may create an internal program counter, which is used to point to the next execution instruction and does not directly correlate to the “external” (program, view) program, counter and so, the Value Prediction (VP) implementations may not correlate with the micro-ops unless special hardware is added. Thus, the use of uop cache is essential for reducing the overall power consumption of the entire core. Thus, we need to strive to use VP and uop cache to achieve an efficient implementation. Therefore, this work suggests a new technique that allows implementing VP jointly with the uop-cache.

Our contribution in this work is analyzing the power and performance aspects of not using Micro-op cache, implementing Micro-op cache, LVP and VTAGE into the SNIPER simulator, suggesting methods of integration of value prediction with Micro-op cache enabling Micro-op cache goodness, and analyzing the impact of this integration.

In this work, we will show how to integrate Value predictor with Micro-op in a way that improves the number of obtainable predictions, reduces value predictor array size, saves power, and helps to make VP practical to integrate with existing modern processors. To examine the proposed technique, we simulated the proposed system using SNIPER simulator. We added Micro-op cache ability and value predictor to the SNIPER simulator, demonstrating such integration on two types of value predictors. Based on the SPEC2017 benchmark, the results show power improvement in management while demonstrating the benefits of utilizing the Micro-op cache and value prediction at the same time.

# 

# Background

CISC (Complex Instruction Set Computers) architecture is commonly used in computer architectures like X86. CISC instructions use complex instruction that can be in different sizes and receive different operands in different locations. In contrast, RISC (Reduced Instruction Set Computers)[1] allow only simple instructions that perform specific operation on every instruction like addition and subtract, runs on the local register or load and stores that move data from and to the local register from the main memory. In CISC, complex instructions may represent several simple operations in one instruction. To implement an add instruction with RISC between two sources in memory, the data must be loaded from the memory to the local registers before the addition operation can commence. however, in CISC, it could be done in one instruction. While CISC advantages are short code with variable length instructions that utilize the RAM efficiently, instructions are harder to decode, creating a complex decoding stage hardware and structure which is composed of the length and instruction decode since each instruction may be of different length (depending on the opcode and attributes) as demonstrated on the Intel x86 instruction format example which can be found in [Figure ‎3.1]. Consequently, the next instruction would not be found until we analyze the current instruction length only by serial implementation. After the length decode, the instructions could be decoded in parallel. The decode stage is power-hungry and can take several cycles. The final product of the decoding stages is an “RISC” like instructions called Micro-ops (micro operation) or Uops.

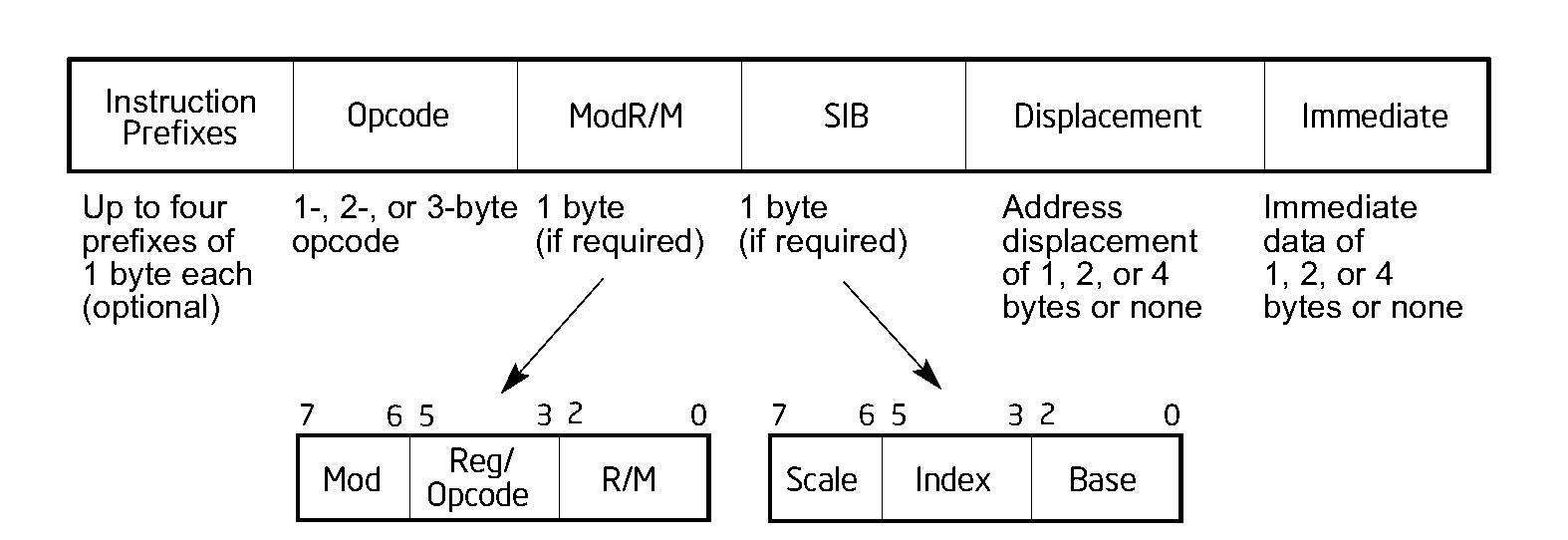


Figure ‎3.1 Intel X86 instruction format, taken from intel 64 and IA-32 Architectures Software Developer’s Manual[2]

“Micro-operation Cache [3] was proposed to close the gap in complexity, and power consumption between the CISC architectures and the RISC architecture. Micro-op cache aims at saving power and decreasing the complexity by eliminating the need to re-decode them again in the next fetch of the same instruction. The cache works in basic-block granularity; basic blocks are a continuous set of instructions that could be entered only via the first instruction in the block and exited by the last one. In other words, it is a set of continuous instructions that, when required to fetch the first instruction, fetches all instructions in the block. Thus, the Micro-op cache will have only one entry pointed to by first instruction pointer of each block containing the decoded Micro-ops of the block that could contain more than one instruction and had different instruction pointer (IP). Micro-op cache supports duplicating the pipe for two directions, as seen in the example in [Figure ‎3.2]. At fetch time, IP is sent to regular fetch pipe and in parallel to the Micro-op cache. On cache hit, Micro-ops will be sent by the Micro-op cache allowing the regular pipe and decoders to be powered off. This process saves significant power and also optimizes performance as the Micro-op cache has the already decoded the instructions and can supply a higher number of Micro-ops without significant hardware impact when compared to increasing the number of decoders. Micro-op cache power and performance benefits are significant and widely implemented in modern architecture[4][5].

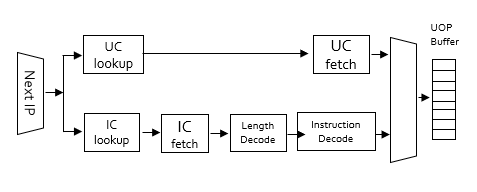


Figure ‎3.2 Example for pipeline with Micro-op cache

On the execution side of the machine, works done to exploit parallelism execution from sequential programs, improve performance, increase the instructions that could be executed per cycle, Instructions Per Cycle (IPC), and improve execution unit utilization by running several Micro-ops in parallel without compromising the correctness of the program can be achieved by either finding the non-dependent instructions and running them simultaneously or by removing dependencies.

When changing the order of the instructions, dependents between them should be reviewed. There are three types of possible data dependences[6]: the first, Read-after-write (RAW), also known as true-data dependence; this is when instruction uses the result of previous instruction causing the instruction to be ready for execution only after the first instruction is finished. The second type is write-after-read (WAR), also known as false dependence; when two or more instructions use the same register or memory location, the program order should observe the stored data after those instructions. If the instructions that should read the former data are overwritten because of the parallel execution or write-after-write (WAW), that explains which data should be stored after those instructions when executed by program order; an example of data dependence is displayed in [Figure ‎3.3]. In addition to data dependencies, control dependence is also a possibility when a branch predictor is used to speculate the direction of a path; instructions that follow the branch are speculatively executed as they are dependent on the result of the branch instruction.

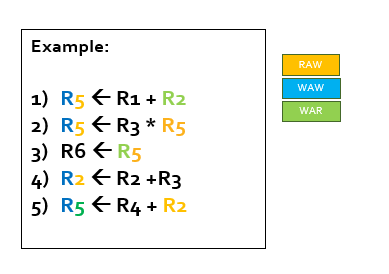


Figure ‎3.3 Dependencies between subsequent instructions

OOO (Out of order) machine was developed to replace the regular execution pipe, such after decoding, instructions should be executed and commit in the order of the program. OOO is used to exploit ILP (instruction-level parallelism) in the program and, in this way, execute in parallel when possible. ILP is a measure of how many of the instructions in a computer program can be executed simultaneously. Computer programs translate commands to binary instruction meant to be run serially, one instruction at a time; however, some instructions are dependent on the proper execution of the preceding instruction.

After Micro-ops are decoded from the instructions, they are sent to OOO for scheduling. The Micro-op writes to the OOO in two places: The Reorder Buffer (ROB) and the Instruction Queue (IQ). The role of the ROB queue is saving the in-order of the program; hence, it has an inbuilt First In First Out (FIFO) structure. When an older Micro-op is ahead, and each Micro-op is waiting to commit by the order, a commit can be done only when all preceding Micro-ops have been committed (removed from the queue). The second queue is the IQ; it is not like the ROB as it is a fully associative buffer of Micro-ops that are waiting to be executed not necessarily in the order of the program but by the readiness of the operands. When some Micro-ops operands are ready, and the execution units are free, Micro-op would be sent to be executed. After execution, the Micro-op is marked as waiting to commit in ROB and marked as ready for next dependent Micro-ops to be executed.

Micro-ops go through Register Renaming, which is the first step in OOO[7]. In this process, the register used in the Micro-ops receives internal and unused registers from the micro-architectures register pool that are larger than the exposed registers. By this, RAW and WAW dependencies are removed because subsequent writes to the same register will now write to different ones when renamed, eliminating the need to reorder the Micro-ops because of such dependencies. After the renaming, only RAW dependence should exist, allowing the program to run by a data flow graph instead of a control flow. In [Figure ‎3.4], the data flow execution example in two systems is displayed, the first using in-order execution and the second using out-of-order execution.

After analyzing the independents Micro-ops, they are sent to execution units in parallel as early as their operands are ready. Finally, a commit should be in the order of the program, making it invisible to the user by saving the correctness of the program.

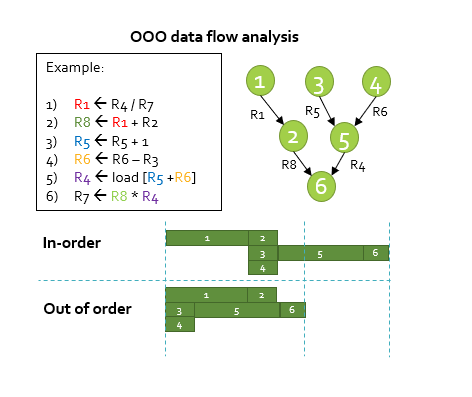


Figure ‎3.4 - Data flow execution improvement due to using OOO execution

# 

# Related works

In this work, we will focus on value prediction and present a new method of using value predictor integrated with the Micro-op cache. Subsequently, we will explain related works that support our hypothesis.

Value Prediction (VP) is a micro architecture mechanism to improve ILP[8]. The OOOE machine was first proposed in the mid-90s by Gabbay, Mendelson [9][10], and Lipasti et al.[11]. In their work, they suggested a concept of exceeding the bounds of true-data dependencies without violating the correctness of the program. Their stipulated method was by predicting at run-time the outcome values of instructions before they are executed, allowing the instructions that depend upon these values to bypass their dependency then get executed in parallel or even in reverse order speculatively. Prediction leverages value locality; this mean that instances of same static instruction produce same values or could be calculated by constant pattern. As result, the value prediction ILP of the program will increase. The example shown in [Figure ‎4.1] displays running 4 subsequent instructions with ILP of 1. When each of them is dependent on the one that precedes it and using prediction for the 2nd instruction, the 3rd one can run in parallel to the 1st one resulting ILP of 2, increasing the performance by two.

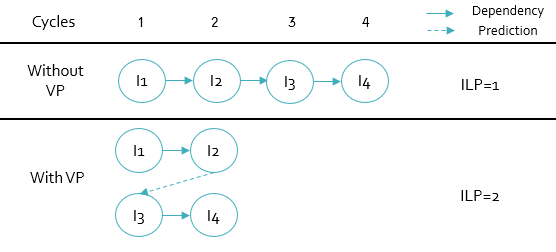


Figure ‎4.1 ILP improvement due to VP prediction

The mechanism for VP that was suggested was like a branch predictor containing the following elements: prediction schemes to generate the prediction and tagging instructions that were executed upon a prediction, validating the correctness of the prediction and recovery mechanism. Unlike branch predictor that predicts only branch instructions which gain significant performance improvement instead of stalling the pipe, VP can predict any ALU or load instruction with varied performance gain between instructions from a small gain in the example ALU instruction which is not blocking the pipe and is dependent on other ALU instruction and ready after one cycle, to a more substantial gain of using prediction in load instruction of data that exist in memory that blocks the pipe for a longer period.

A few known recovery mechanisms were examined to be used for value prediction[12]. The first was Selective replay. This method is used to mark the ROB on every prediction of the Micro-op that used this speculative value and every dependent Micro-ops. Upon recovery is needed as we did not commit any speculative Micro-op, we can only re-execute those speculative Micro-ops with the right value. With this method, we get a low recovery penalty; also, the complexity of using this method is high and affects sensitive mechanisms of the OOOE. The second option considered was Pipeline squashing. This method supports branch predictors; when recovery is needed, the pipe will be flushed and re-fetched from the missed predicted Micro-op. This method will cost a high recovery penalty but can be easily implemented, and it is already used in branch predictor architectures.

Several types of predictors were also suggested. The first is LVP (Last Value Predictor); this predictor predicts the result of instructions already produced by the previous instance of the same instruction; it will perform optimally when the value on the same instruction remains the same. It is built from two fields of tag and value, as shown in [Figure ‎4.2]. At the fetch stage, the current PC is being looked up using a subset of the address as SET, and the rest of the bits will be compared to the TAG field. Once the TAG match hit flag is raised, indicating the value for speculative execution, validation will be done at the commit stage, updating the array.

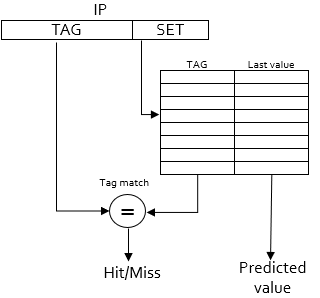


Figure ‎4.2 LVP - Last Value Predictor structure

The second type of predictor considered is the Stride predictor. Instead of computing the last value of previous instances like the LVP, it can also calculate instructions that have constant stride between the instances, i.e., instruction inside a loop that looks like: C=constant\*i, when ‘i’ is the loop index, and it increases by one every loop iteration. The C value should be increased in every iteration by the constant number. In this case, we can say that the stride is equal to constant, and to predict the next instance result value, we need to calculate “last value + constant” although the predicted value has never been used, it can be predicted successfully. In addition to the LVP structure, in this predictor, we should save the stride of each entry, as can be seen in [Figure ‎4.3]. Look up is done in the same way as LVP, but also, the current value Stride should be added to the Last value, and then the new value should be written to the “Last value” field every read. In contrast to the LVP predictor, here we will suffer from back-to-back instructions in OOO architecture, and as some of the instructions were fetched speculatively, we cannot be sure when to add the stride to the updated value.

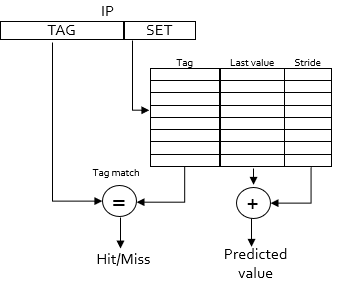


Figure ‎4.3 Stride predictor structure

The third type of predictors is the context-based predictor. The assumption behind this type of predictor is that in some cases, different traces that lead to the same instruction could cause a different value result from a specific instance, .i.e. this could be caused by instructions that could be reached by three jumps instructions, and in each source of jump trace one register is used with specific instructions to receive different value causing two different results from the instruction. Sazeides and Smith suggested in their work[13] a two-level value predictor. The first level using Value History Table (VHT) contains the local value history of an instruction accessible with PC. Access is done with SET (partial bits from the PC), and subsequently, TAG match is done. The second level, the Value Prediction Table (VPT), is accessed with a hash of the VHT; the value will be taken on tag match of the first level [Figure ‎4.4]. In addition to those types, there are also hybrid solutions that suggest using several types of prediction together[14].

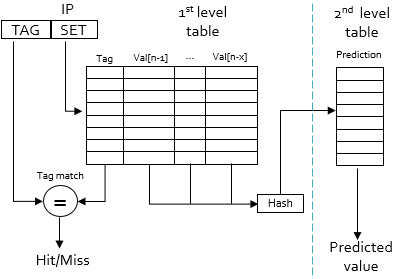


Figure ‎4.4 FCM predictor - 2 level predictor

We have presented recent predictor mechanisms, but for more reliability, confidence mechanism would be added to each predictor. Before making any prediction, there must be a tendency for the prediction to be correct. In the past, before making the first prediction, the value is being attached to the predicted instruction, and when committing the instruction, the prediction value is validated, and the confidence counter will be increased with every correct prediction and decreased on every bad prediction.

Although most research on the subject of VP showed significant performance benefits, it has never been tested in real hardware. The reason for that is the complexity of the implementation in critical stages of the pipeline; while prediction should be made at fetch time without critical timing, the validation and recovery should be made inside the already complicated OOO machine.

In the last years, value predictor became a hot topic, and new studies into the subject emerged and were encouraged. Champion Value Prediction (CVP)[15] competition arose a few years ago; in the contest, the three leaders use context-based predictor[16][17][18]. Those studies focus on context-based predictors suggesting improving prediction accuracy that is critical due to the misprediction penalty[19]. Arthur Preais and André Seznec suggested one of the VP suggestion from the contest; the Value tagged geometric predictor (VTAGE)[20] that was derived and took the same principals and methods from research on indirect branch predictor named ITTAGE [21]. VTAGE is context-based as it uses long global branch history and path history to provide prediction, as can be seen in [Figure ‎4.5]. This predictor aims to solve the hardware complexity that previous ones suffered. They suggested a new method of confidence mechanism, the FPC (Forward Probability Counter); in each correct prediction, the counter will advance forward with a certain probability that depends on the type of the instruction. As we described before, each type of instruction could differ in its reach gain, which favors giving correlate priority to the prediction gain. Using the FPC, they reached an accuracy of at least 0.997 in different benchmarks, but with some reduction in predictor coverage. Also, they used Pipeline squashing instead of Selective reply, as we described earlier that it is easier for implementation already exists in modern branch predictor architectures but with the price of penalty. When used in high accuracy as is often the case, the penalty price on average is low. Another reason why their predictor practical uses validation at commit time instead of execution time is that validation at commit time is easier for implementation without complex hardware interference in the OOO machine and eliminating the need for additional register file read ports as shown in their work.

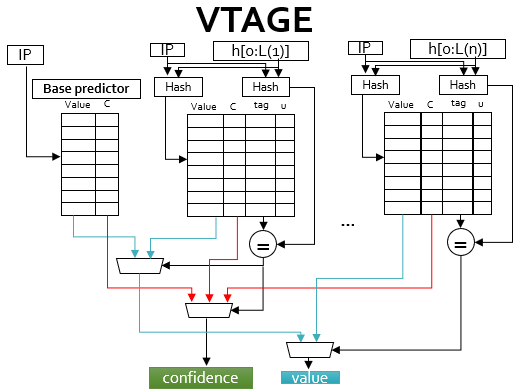


Figure ‎4.5 - VTAGE predictor structure

VTAGE structure is built in this manner: array access is done by using global branch history hashed with PC and have different levels of history tables when the match with the longest history should be taken; this allows the predictor to get the direction the instruction came from. Array fields are: “useful” used for LRU (Least recently used) for eviction policy, “c” confidence counter, “tag” the remaining bits of PC to be compared if we have tag match, “Val” the last updated value.

POWER8[22] of IBM possesses a Micro-operation cache. In this architecture, they found a practical way to inject Micro-operation data on branch prediction using predicates; prediction information is then added to the next Micro-op and validated afterward. This architecture serves as the inspiration for our work on value prediction to be injected as a predicate to the Micro-operation cache.

At the time of this work, VP solutions were not commonly used, and although VTAGE shows the practicality of VP integration in their work, there is still some complexity in integrating such predictors to modern CPU architectures. Micro-op cache architectures were not even mentioned in existing works; we found it really challenging to make VP practical. Without a Micro-op cache, the power consumption should be increased significantly, and performance should be decreased, making VP inexpedient. Integrating VP to those architectures is not straight forward, while VP uses IP (Instruction Pointer) to get access to the arrays, Micro-op cache is working in basic-block granularity powering off the normal Fetch/Decode pipes. In this work, we will address the impact of Micro-op cache using value predictor; we will suggest a practical way to integrate any existing VP to these architectures and would present the results of experiments using specific VTAGE and LVP while articulating its pros and cons.

# Terms and conditions

This section aims to define the terms and the conditions we will use during this thesis:

## Terms

**CPU**: Central Processing Unit, the main processing unit of modern computer systems. The CPU receives a compiled program made by the user and executes the operations.

**ISA**:Instruction Set Architecture defines the supported data types, the registers, the hardware for managing main memory, fundamental features, and the input/output model. It also specifies the behavior of machine code running on the implementation of that ISA.

**IQ**: Instruction Queue, as described in “Background”.

**IPC**: Instructions Per Cycle; the number of instructions that can be performed by the CPU in one cycle.

**OOO**: Out of order, as described in “Background” section.

**ROB**: Reorder Buffer, as described in “Background”.

**UOP**: micro operation; name of already decoded instruction to the number of small operations.

**ILP**: Instruction-Level parallelism; indicates the possibility to run some programs in parallel by running more than one instruction in one cycle, this could be achieved if some instructions are not dependent on the previous one.

**VP**: Value predictor, as described in “Background” section.

**LVP**: Last Value Predictor, as described in “Related works” section.

# Our approach

Modern architectures use Micro-op cache that works in basic blocks granularity. Since the ISA uses CISC instructions and the uop instructions are of RISC type with fixed size, there is no direct correlation between the IP address of the CISC instruction and the internal uop address within the block. Thus, integrating VP within a uop cache is not trivial and is more complicated than implementing branch predictors, which can be imposed at a basic block granularity. In this work, we proposed integrating value prediction within the Micro-op cache while using already existing value prediction methods. The section will describe the algorithm as well as the pros and cons of using it.

Parallel decoding of CISC instructions consumes much power. Thus, Micro-op cache, as described in the “Background“ section, can provide significant power saving and can also improve performance in modern architectures, as decoding stages are powered off while using the cache to retrieve already decoded instructions. Improved performance is achieved due to farther optimizations that take advantage of the grouping of instructions into basic blocks. The Micro-op cache hit rate in previous Intel architectures is at least 80%, and in hot spots can achieve hit rates closed to 100% from intel optimization guide[23], making the power saving of using Micro-op cache to become very efficient. Micro-op cache also gains performance over the usual decoding pipe, shortening the path from instruction to the generated decoded Micro-ops and can supply a bigger capacity of decoded uops than the regular decoder without the immense hardware complexity. These benefits make value prediction integration to Micro-op cache very valuable.

In this section, we will present an integration between value predictors and Micro-op cache, allowing them to work together; we will focus on already existing mechanisms, such as VTAGE and LVP already described in the “

Related works“ section. It is important to note that although we illustrate the results on those predictors, the integration should work with any other existing predictor with a similar structure of update mechanism, prediction, validation, and recovery.

Our new proposed technique is divided into two phases; the first one is the study phase; during this stage, a new value prediction will be trained before it can be used in confidence. The second phase is the prediction phase; during this phase, only reputable prediction will be used.

In the study phase, training a new value scheme is essential to make accurate predictions and reduce miss-penalty. Thus, using the integration to existing value prediction, as described in the previous section, will provide the best outcome. Accessing those predictors is achieved using PC but that does not exist on every Micro-op when using Micro-op cache working in a basic block granularity, the only PC that is known while working from Micro-op cache is the one of the basic blockhead. Thus, our suggestion is to provide a “virtual link” to the existing arrays allowing them to work, as usual, building the confidence for the prediction. During this phase, the prediction fetch and validation is not on the critical path of the CPU as we are not using the value for real execution; the data read from the prediction array can arrive as late as the validation will take place with the result data, this will ensure the hardware is not located in a critical area and connected in un-critical paths.

In the second phase (the prediction phase), after the confidence counter is saturated and the value is ready to be used in real prediction, the prediction value will be copied to the Micro-op cache, and the “virtual-link” generated in the first phase will be removed. The prediction usage contrary to the value build now sits on the critical path. Copying the value to the internal Micro-op while removing the link to the arrays will simplify the algorithm, making predictions easier and faster while freeing up the used entry in the prediction array, allowing new allocations.

The first change that will be made will be to the actual indexing of the value predictors. Instead of using the actual PC of the Micro-op, we will use a hashed and compact version of PC information attached to a basic blockhead PC allowing only small data to be attached to the Micro-op cache; this hashed version of PC is what we call “virtual link.” The access behavior from the value predictor side will be identical to using the hashed PC with the basic blockhead PC. An illustration of the link generation could be found in [Figure ‎6.1].

We found that the simplest way to generate the “virtual link” is by indexing each Micro-op from the blockhead with 3 bits, allowing 8 Micro-ops on each instruction pointer. Access to the value predictor array address can be achieved by the following: virtual link bits will be used as the lower bits of the address, and the remaining bits will be derived from the address of the basic blockhead. For an array size of , access set should be in the size of K; 3 bits from the virtual link, and (k-3) bits will be the lower bits of the basic blockhead without the offset. Access to value predictor using the “virtual-link” will be used until the build is complete, or, in other words, when the confidence counter is saturated.

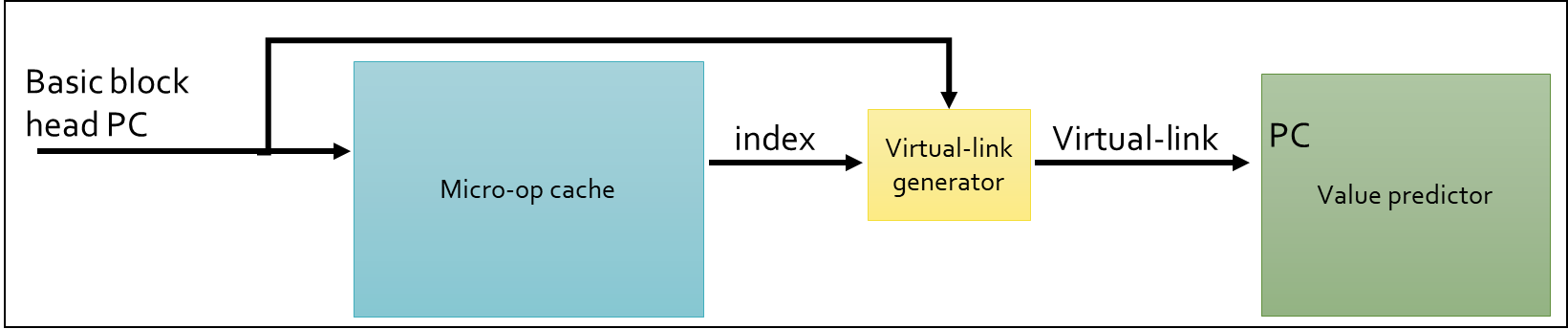


Figure ‎6.1 “Virtual-link”

When saturation is achieved, prediction information will be copied into the Micro-op cache replacing the “virtual-link” data. Overloading the “virtual-link” data is possible as we do not need the link information anymore and this also helps restrict the Micro-op cache entry size; this information should contain the prediction value and the valid bit that tells the hardware to use prediction. At this point, VP entry in the VP array should be removed and we should not allocate the same instruction again in VP array while it is still running in the Micro-op cache. As the advantage of removing the entry from the main predictor array is to free space and reduce array size, the disadvantage is that the recovery mechanism of current predictors is lost. In a context-based predictor, the “context” information predicting that works like LVP will also be lost.

Removing the entry from the value predictor is straight forward since we have the “virtual-link” that can be used to invalidate the correct entry. In LVP, it will remove one entry that matches the specific Micro-op, while in VTAGE context-based predictor, each Micro-op can live in several entries that depend on the global branch history, the entry that will be removed will be the one to arrive at the saturated confidence first.

Micro-op cache entry should be transformed to support this algorithm; each relevant Micro-op entry will get additional fields for VP information: valid link, valid predict, “Virtual link,” and value as shown in [Figure ‎6.2]:

* Valid link – indicate that this Micro-op has an active link to the learning VP.
* Virtual link – hashed PC of the instruction to be used to access VP predictor tables alongside basic blockhead. As described before in our work, we used 4 bits of index from the basic blockhead.
* Valid predict – indicates that this Micro-op has a valid value to be predicted.
* Value – prediction value; will be stored after saturation.

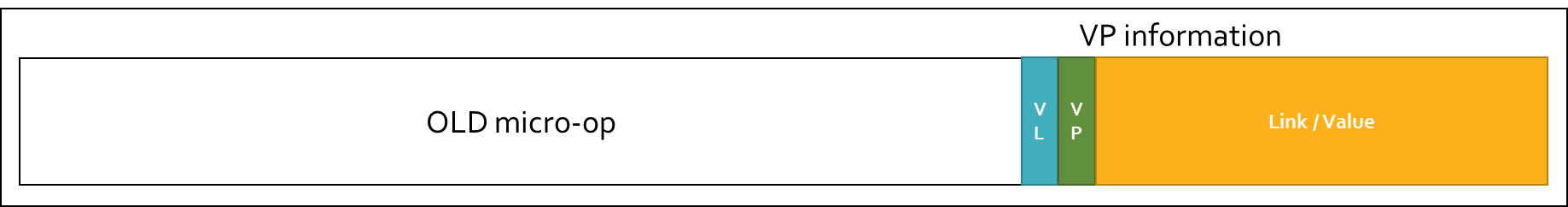


Figure ‎6.2 Micro-op fields after adding Value predictor integration

Although we have a different field for virtual link and value, they can be overloaded, indicating which one should be used by the valid bit; those fields can also overload existing Micro-op fields, for example, if the Micro-op already has fields that are not used with any type of instruction.

While valid predict bit is enabled, the Micro-op arrives at ROB and IQ. OOO can work similarly as previously suggested, with little changes. Instead of receiving the predicted value from the array, the value already exists with the Micro-op structure. The corresponding Micro-op is marked as “speculative ready,” allowing dependent Micro-ops to execute in parallel, without removing the Micro-op from IQ until it will be validated. Storing the value locally in the Micro-op supports hardware relief on critical path.

The validation or update mechanism should also be very similar to previous works with little change; while the valid bit is off, and the valid link is on, update works in the “study phase” and is done in the external predictor accessed with the virtual-link mechanism described before. When predict valid is on, working from “prediction phase,” validation mechanism works on commit time, using the same mechanism described in VTAGE work; in the wrong prediction, flushing the pipe is done, clearing the VP information from the wrong predicted Micro-op, causing the specific IP to go back to the “study phase” from the start.

# Experimental testbed

In this section, we introduce the simulation testbed along with the tools and system methodology used for conducting our experiments.

## The testbed simulation environment

In this work, we used Sniper simulator[24], a parallel high-speed and accurate x86 simulator. This multi-core simulator is based on the interval core model and the Graphite simulation infrastructure, allowing for fast and accurate simulation and trading off simulation speed for accuracy to allow a range of flexible simulation options when exploring different homogeneous and heterogeneous multi-core architectures.

|  |  |
| --- | --- |
| Component Name | Sniper X86 simulator v7.3 |
| System | Ubuntu 16.04.6 LTS |
| Platform | Micro op with rob performance model |
| CPU | Frequency 1.66 |
| DRAM entries | 1048576 |
| Max Memory Bandwidth | 25.6 GB/sec |

Table 1 - Testbed configuration

As Sniper simulator uses PIN and binary translation to simulate the trace on a real machine with binary translation wrappers collecting and add functionalities, when using speculative predictors, only the true path is really running on the machine, while bad prediction will only be simulated virtually without really running. Some of our penalty estimations will use a constant number – for this, we will run our experiments with the penalty as a variable.

## Implementation in Sniper simulator

To test our approach, we implemented some addons into the Sniper simulator; the components that we implemented are; basic Micro-op cache, LVP, and VTAGE value predictor while the Micro-op cache and LVP were implemented from scratch. VTAGE source code was taken from the CVP contest[15] with some adjustments to support integration with Sniper. All those addons are contributions that can be used in Sniper to future works in any topic and could be fetched by GIT. In the following, we will describe each component's implementation.

### Micro-op cache

Micro-op cache described in “Background,” while it is significant for our work, SNIPER simulation does not have any available Micro-op cache implementation. For our experiments, we implemented a Micro-op cache into the SNIPER simulator source code. We chose the simple way for implementing Micro-op cache, using n-associative cache with LRU as eviction policy, which is saving any decoded instruction in every basic block pointed by the basic blockhead; in the next fetch, it will be read from the cache.

In this implementation, we did not record any impact on performance if some instruction got hit and fetched from the Micro-op cache but only as a preparation to the VP integration that will be described next; since our work was not on the Micro-op cache, those numbers alone are not relevant for this work.

#### Structure

The cache was built as a n-ways associative cache, and the read address is a partial set of the PC – lower bits. In Sniper, we do not need to read the Micro-ops from the array; thus, we do not save the Micro-ops in any way, but only the address, and that is enough for marking hit/miss in the Micro-op cache. All array data should not be relevant as this cache is used only to integrate Micro-op cache to value predictor and not for testing Micro-op cache effectiveness that will be reviewed from existing architectures.

#### Update algorithm

Implementation update will be done on each instruction fetch of the basic blockhead with the same mechanism of prediction in simulation when the full picture is known. If some PC already exists in the array, LRU will be updated; either way, a new entry will be allocated using LRU for eviction when needed.

Set is calculated by simple hash of PC mod, by the number of cache lines, and PC tag is the remaining number of bits.

#### Prediction algorithm

Micro-op cache read is achieved in each instruction fetch; prediction is achieved by PC of basic blockhead, and the set calculation is done by hashing PC with the number of the cache lines. We will read all ways in the calculated set, and then we will compare the tag with the current tag to find a match. If we find a tag match, a Micro-op cache hit is marked.

As Sniper simulation is done by binary translation engine of Pin, the predictor does not need to provide real data but indicates that the prediction is completed.

#### Eviction policy

The LRU mechanism does the eviction. The new entry would be allocated if an empty entry is existing in one of the ways and will be written in the empty place; else, the oldest entry will be evicted, allowing the new entry to be allocated.

### Last Value Predictor - LVP

We implemented Last Value Predictor into the Sniper simulator to give a basic idea of how a simple implementation of value predictor will behave with our integration. The LVP was implemented as an array directed by lower bits of the PC address as the SET of the entry. The remaining bits of the PC saved as TAG in the entry will be compared later; also, we save the confidence counter, the last value, and LRU bits. The prediction should be taken care of only when confidence count saturates. Saturation is done after eight times of correct prediction so that the value will be taken as prediction only confidence of 8 times. If the prediction is correct, we can obtain the right positive feedback. We cancel dependencies in the SNIPER simulator, making the internal OOO mechanism to run dependent instructions. On a bad prediction, a constant penalty will be taken to reflect the recovery price.

#### Update

Because simulation in Sniper is running by the side of the real program in Pin, the prediction could be made when the real value is already known. Update in the simulation is done at the same time of prediction. When checking if prediction is existing (Prediction section), we can perform a later the update. Entry is chosen by hashing PC with the number of entries in the array, then we check for tag match in all available ways. If a match is found, we increase confidence and allocate a new entry using the Eviction policy.

In tag match, the same value that stored the confidence counter is increasing by one until it will arrive at saturation.

#### Prediction

Prediction happens in the same way as an update. In a simulation, as is not the real world, we can call the prediction at usage time instead of fetch time without any real impact. To obtain a prediction, the PC is hashed with entries number, reading all ways in one sweep before checking for a tag match. If there is a tag match that arrived at saturation, the hit signal is raised up, and prediction will be used. The prediction will be checked immediately as we already know the result; for a good prediction, we are canceling all dependencies of the following Micro-ops, and in a bad prediction, the penalty will be calculated.

#### Eviction

If a new set IP is written to value prediction array after hashing the PC to SET, all ways with the same address are reviewed. If the IP already exists in one of the ways, it will be updated in the same manner; if not, it will check for an empty entry, and if there is no empty way, LRU will be reviewed to find the least recently used entry to be evicted.

### VTAGE

VTAGE implementation was taken from CVP contest[15], while in the CVP contest, it was implemented into the CVP simulator as we wanted results to be relevant to our work and to be integrated with Micro-op cache with some adjustment to Sniper; hence we commenced the implementation.

#### Update

As we know the real value in prediction time, we can update the predictor at the same time as prediction. VTAGE predictor is using two types of updates – speculative update and normal update. While the former should be done before commit at the early stages of prediction, the second one should be done after commit. In Sniper, we use these two methods simultaneously as we do not experience early or late differences in simulation. As part of the VTAGE, it will calculate hash based on global branch history with PC; we implemented a global branch history that gets updated in each update by saving the history of PC trace.

#### Prediction

As LVP predictor, the prediction mechanism works at the same stage of the update mechanism. Prediction is made at usage time in the Sniper simulator, allowing the use of prediction, validation, and update at the same time. At prediction time, all components accessed take the match component with the longest history to obtain the prediction.

#### Eviction

As we do not have a single entry per IP and the prediction is based on hashing PC

Global branch history, there is no LRU mechanism needed. We only need to update the correct prediction that matches the current history with the matching history level found.

### VP in Micro-op cache

#### Structure

We added a new field of VP information to the Micro-op cache. This information contains the fields we talked about in “Our approach“ section.

#### Update

Before getting the prediction confidence, a virtual-link field is added to each Micro-op used to access the external value predictor using the same value predictors mechanism that already exists in those predictors. After confidence is saturated, the link is removed while copying the value to the relevant Micro-op to be used internally.

#### Prediction

Prediction is made at usage when value already exists in the Micro-op entry, making it easy to be implemented. If the entry contains a valid prediction, we will apply it since we know it already got the confidence before it was copied to the Micro-op entry. On a bad prediction, the value will be removed by disabling the valid bit.

## Measurements Methodology

## Workloads

In order to measure the impact of our work, we used SPEC2017 workloads in mode of SPEC Speed integer:

|  |  |  |  |
| --- | --- | --- | --- |
| Test name | Language | KLOC | Application Area |
| 600.perlbench\_r | C | 362 | Perl interpreter |
| 602.gcc\_r | C | 1304 | GNU C compiler |
| 605.mcf\_r | C | 3 | Route planning |
| 620.omnetpp\_r | C++ | 134 | Discrete Event simulation - computer network |
| 623.xalancbmk\_r | C++ | 520 | XML to HTML conversion via XSLT |
| 625.x264\_r | C | 96 | Video compression |
| 631.deepsjeng\_r | C++ | 10 | Artificial Intelligence: alpha-beta tree search (Chess) |
| 641.leela\_r | C++ | 21 | Artificial Intelligence: Monte Carlo tree search (Go) |
| 648.exchange2\_r | Fortran | 1 | Artificial Intelligence: recursive solution generator (Sudoku) |
| 657.xz\_r | C | 33 | General data compression |

Table 2 - SPEC2017 workloads

## System configuration

### Reference - Clean Sniper environment

In the reference, we used a clean Sniper environment without any additional configuration – this will set the “Reference” to be compared to our work tests.

### LVP

In the LVP configuration, we only enabled the LVP predictor. In the following experiments, we used some configurations of different sizes; “unlimited” means that the size is large enough to contain all the entries without the need for eviction policy, 32K, and 8K. Another configuration implemented was the miss penalty used in case of miss prediction.

### VTAGE value predictor

In the VTAGE configuration, we only enabled the VTAGE predictor. In the following experiments, we used some configurations of different sizes; “unlimited” means that the size is large enough to contain all the entries without the need for eviction policy, 32K, and 8K. Another configuration implemented was the miss penalty used in case of miss prediction.

### Micro Op cache + LVP

Here we are enabling the integration of the Micro-op cache and the LVP predictor.

### Micro Op cache + VTAGE Value predictor

Here we are enabling the integration of the Micro-op cache and the VTAGE predictor.

# 

# Experiments

In this section, we present all the experimental results and discuss them. We also validate the observations presented at the “Our approach” section.

## Reference environment

In order to review our work, we first implemented the reference environment to the Sniper simulator. We did not find any existing implementation for the following in SNIPER: Micro-op cache, LVP, and VTAGE; thus, we implemented them into the Sniper simulator as described above.

Implementing the Micro-op cache and LVP as simple predictors and VTAGE was achieved in the CVP contest[15] with published source code; we took the source code and integrated it with SNIPER. More details can be found under the “Experimental Testbed” section.

Reference environment experiments are done to make a clear reference area that matches as many results as possible to the test area allowing for a better comparison of results which was influenced by eliminating background noise.

### Penalty sensitivity check

We found that in the Sniper simulator, there is no inherited way to simulate an accurate miss penalty as it always runs the correct path. The way to obtain a miss penalty impact is by counting the constant miss penalty on every miss. To find the most correlative penalty, we did a penalty sensitivity check. We postulate that when the miss penalty is increased, the IPC should get worst.

In this experiment, we took LVP and VTAGE with an unlimited array size and compared it to a clean Sniper environment.

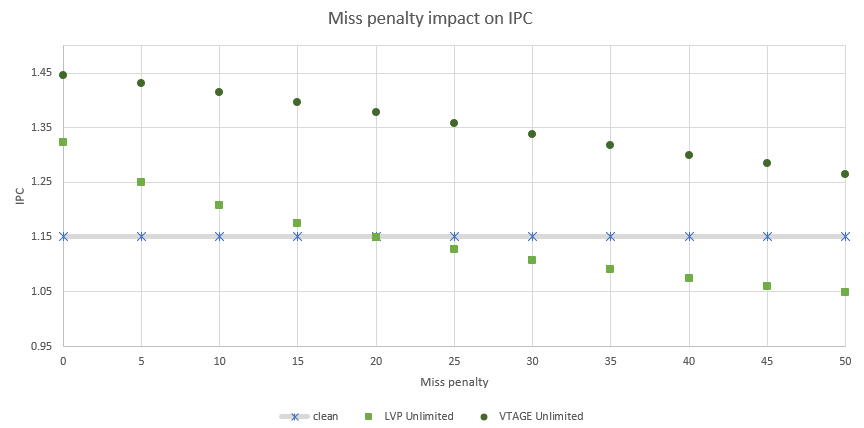


Figure ‎8.1 – Value prediction Miss penalty impact on IPC in reference environment

The results show what we expected. As the penalty increases, IPC getting worst. We can also see that our implementation of LVP gets the worst IPC when running a clean environment with a penalty bigger than 20 cycles. In [Table 3], we can see the Hit rate with a Penalty of 20; although the hit rate seems high and bigger than 99%, the small changes after the point show how LVP is sensitive to penalty; thus, we learn that the gain of successful hit is much lower than the price of miss penalty.

|  |  |
| --- | --- |
| **Predictor** | **Hit Rate** |
| LVP unlimited | 99.465% |
| VTAGE unlimited | 99.969% |

Table 3 - Hit Rate of LVP and VTAGE with penalty of 20 cycles

For the next experiment, we will use 5 as the default penalty cycles because we are not checking the effectiveness of the predictor, and we want to compare it to our work with the Micro-op cache, so it is better to take the predictors at their best point.

### Size impact

Next, we checked the impact of limiting the size of the predictor versus clean a Sniper environment. In the following, we checked 3 predictor sizes: 8K, 32K, and Unlimited. Our first guess was that a bigger array should result in a better IPC.

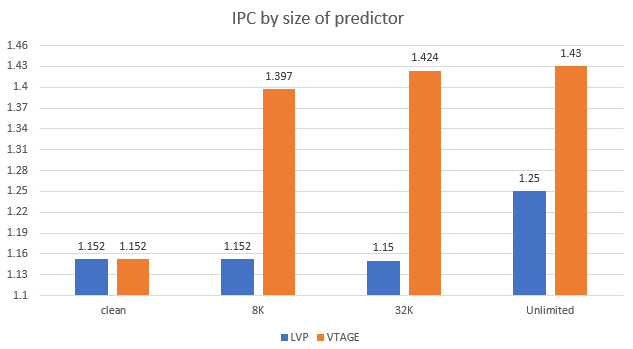


Figure ‎8.2 - IPC impact by the size of the value predictor in reference environment

In VTAGE, we obtain an expected behavior; in LVP, we see a glitch in IPC using the 32K predictor. This glitch is because a linear behavior is not expected since the predictor is much larger, the coverage should increase, creating a larger number of predictions, and in the case of LVP, increase the number of chances for the miss prediction to obtain a miss penalty. Although we increased the size of the predictors, we do not see significant changes in the IPC number. For our test, we will use the 8K predictor as the baseline, as we expect it to be best suited for a practical predictor.

### SPEC2017 breakdown

In this test, we run LVP and VTAGE with 5 cycle penalty and size of 8K, showing the breakdown of IPC by each SPEC2017 test.

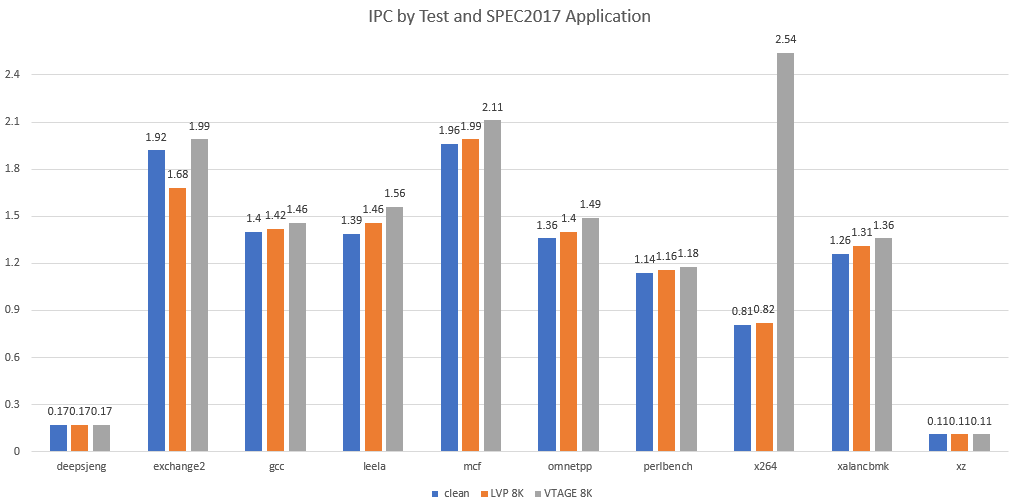


Figure ‎8.3 – IPC of different 8k value predictors SPEC2017 breakdown of reference

As we see from this test results, the VTAGE predictor is better from Clean/LVP models in all application tests. However, in two applications, the results are the same. The effectiveness and performance gain of VTAGE, therefore, is acceptable. In LVP, we see that in exchange2 application, it is worse than the clean model, but in the others, it also gives some performance gain.

## Micro-op cache characterization

While in the previous sub-section, we showed the reference implementation on the SNIPER simulator. In this section, we want to show the integration result, and impact then compare it to the reference.

### Micro-op cache impact on power and performance

In Micro-op implementation, every instruction that runs more than once should be stored in the Micro-op cache until it will be evicted. Although we built micro-op cache into the Sniper simulator, the net profit of Micro-op cache cannot be generated from this implementation as there is no clear way of showing performance gain and power gain of this implantation. So for the Micro-op cache reference, we ran Keiko model on Intel architecture to show the increase of IPC and Power saving during those runs. From the Intel optimization manual[23], we can see that the Micro-op cache hit rate is above 80%, and from the cycle accurate simulation model, we can save 14% of power while improving IPC by 7.5% over the reference.

In the following diagram, we can see the IPC improvement while using the Micro-op cache enabled by SPEC2017 application:

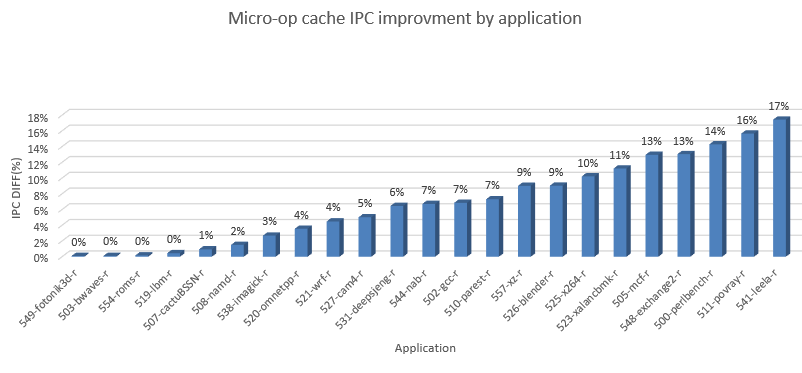


Figure ‎8.4 Micro-op cache IPC improvement over disabled Micro-op cache by application

In the following diagram, we can see the Power saving while using the Micro-op cache enabled by SPEC2017 application:

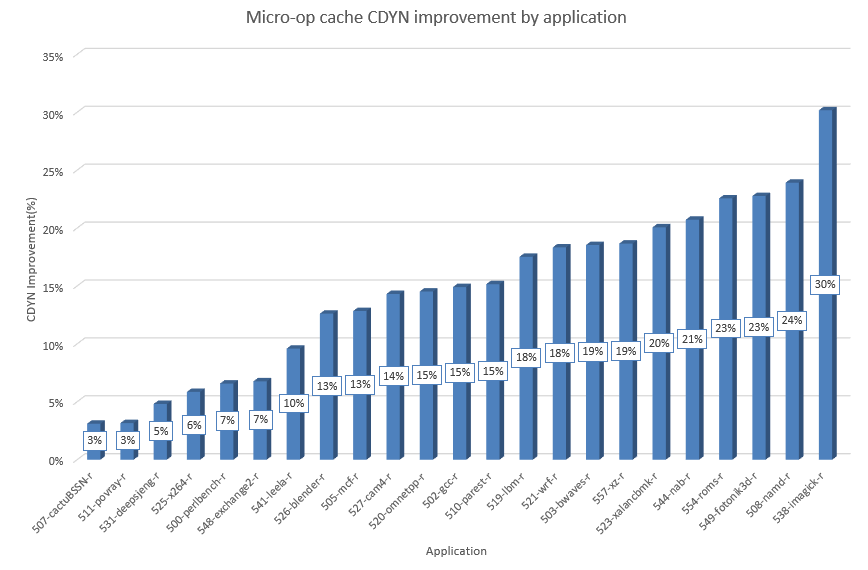


Figure ‎8.5 Micro-op cache Cdyn improvement over disabled Micro-op cache by application

As we can see from [Figure ‎8.4] and [Figure ‎8.5], enabling the Micro-op cache will improve IPC and CDYN. Improvement can reach about 30% of Cdyn improvement, and in some applications, can reach 17% improvement.

### Micro-op integration impact on IPC

In this test, we run our integration of Micro-op cache with each of the predictors LVP & VTAGE, using 8K predictors and a penalty of 5 cycles. As after getting the confidence to predict during the “study phase,” the value is moved from the base predictor to Micro-op entry, removing the entry from the predictor, and canceling the link between the Micro-op to the predictor and now the Micro-op cache is used to be the prediction provider. This behavior removes the context-based behavior of the VTAGE for the “prediction phase” on the specific value and makes it work similar to normal LVP; thus, we should expect performance impact in comparison to the normal VTAGE predictor. On the opposite side, for LVP, we should expect a small increase in IPC as we remove the entry from the small 8K array when moving to the Micro-op cache, allowing LVP to allocate new candidates for prediction.

In the following graph, we can see the integration impact on LVP and VTAGE predictors without the real IPC benefit from Micro-op cache. In other words, the Micro-op cache performance impact alone is eliminated from this view; this will help us analyze the clean impact on each predictor type.

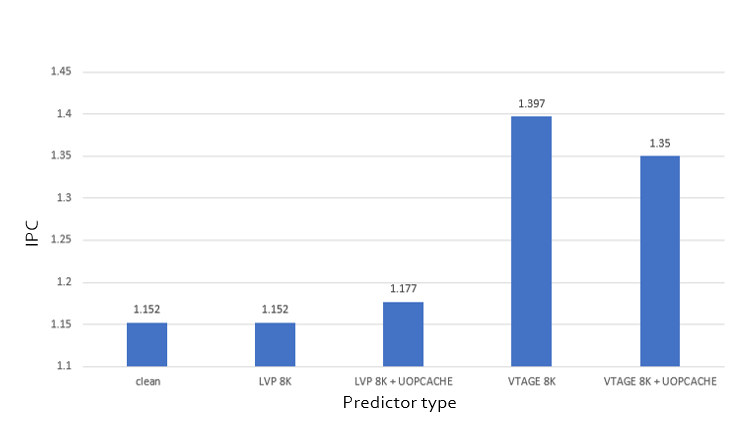


Figure ‎8.6 - IPC by type of the predictor, while integrating with Micro-op cache and without, without Micro-op cache additional IPC improvement

As we see from this test, we got the expected behavior; LVP IPC increased while VTAGE decreased. Although we are using small predictors, we still see some IPC improvement over the clean environment while enabling the use of value predictor with Micro-op cache.

In the next graph, the UOPCACHE column was added, reflecting the IPC improvement over a clean model as shown above. The impact also reflected in the relevant columns of “+uopcache” as well, allowing us to see the numbers to be impacted not only from the integration but also from the micro-op cache existence.

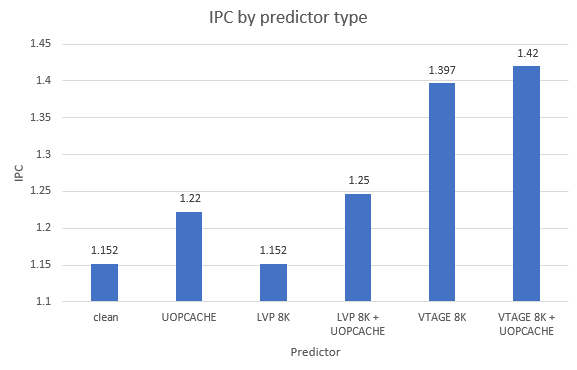


Figure ‎8.7 - IPC by type of the predictor, while integrating with Micro-op cache and without, contain Micro-op cache improvement

As we can see from this test, a Micro-op cache integration with a value predictor provides great performance improvement combined with a simple predictor as LVP and an advanced predictor like VTAGE.

# 

# Summary and future work

## Summary

In this work, we demonstrate how to add a value predictor to modern architectures that contain Micro-op cache. First, we presented a “virtual-link” to be used in the study phase allowing a connection between Micro-ops that exist in Micro-op cache working in basic block granularity to existing value predictors. While the connection is not on the critical path, this allows for easy hardware integration. Secondly, we introduced our suggestion for the “prediction phase.” In this phase, the value should be copied to the Micro-op entry allowing better hardware implementations on the critical path. To show the impact, we worked with Sniper simulator, implementing all required features like Micro-op cache, ability to store register values in PIN, LVP predictor, VTAGE predictor, and also the integration between the value predictors to the Micro-op cache. At the first stage of the experiments, we set the reference area as a clean environment without any predictor, value prediction of each LVP, and VTAGE; in this stage, we tuned the miss penalty and predictor size. In the next stage, we implemented a real word Micro-op cache impact test on Intel modern architectures and recorded the impact on power and performance. At the final stage, we ran the integration between Micro-op cache to the value predictors.

The results showed that using Micro-op cache has significant power and performance improvements over running without Micro-op cache. While using value predictor without Micro-op cache also has a significant performance improvement, integrating value predictor into Micro-op cache modern architectures get the performance improvement and power saving results as using them separately.

This integration did not only improve performance and hardware, it also improved the integration by allowing the split between critical and un-critical path, making the critical path shorter and easier to be implemented while the predicted value exists inside the Micro-op entry shortening the hardware path.

## Ideas for future work

In this work, we only suggested a way for implementing LVP type of value predictors. For future works, we suggest a review of stride predictors. It is apparent that its implementation basis should be very similar; however, in addition, the Micro-op cache should get additional data on the stride while doing the computation. I believe that the stride predictor implementation should get simplicity benefits when moving the stride calculation to the OOO in Micro-op like structure.

In addition, context-based predictors should be researched further. In our research after confidence saturated and copying the information into the Micro-op cache, the prediction loses the context making the prediction to behave like normal LVP. In the research, the technique should be upgraded to save not only the prediction data, but to know the context or the trace it should give the prediction.

Recovery algorithms in my implementation are very aggressive, removing the prediction from Micro-op in bad prediction. Softer algorithms should be reviewed, maybe in some way returning the data back to the original predictor, or maybe instead of removing the entry, some bit can indicate the entry has been “copied” and can be easily evicted when needed. However, if we have a bad prediction case and the entry still exists in the prediction table, it can be revived easily.

# References

[1] E. Blern and K. Sankaralingarn, “Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures.”

[2] Intel, “Intel® 64 and IA-32 Architectures Software Developer’s Manual,” vol. 2.

[3] B. Solomon, A. Mendelson, R. Ronen, D. Orenstien, and Y. Almog, “Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 5, pp. 801–811, 2003, doi: 10.1109/TVLSI.2003.814327.

[4] M. Clark and S. Fellow, “A NEW X86 CORE ARCHITECTURE FOR THE NEXT GENERATION OF COMPUTING,” 2016.

[5] C. O. D. E. A. S. Kylake, J. Doweck, W. Kao, A. K. Lu, and A. Yasin, “INSIDE 6TH -GENERATION INTEL CORE : NEW MICROARCHITECTURE,” pp. 52–62, 2017.

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Amsterdam: Morgan Kaufmann, 2012.

[7] R. M. Tomasulo, “An Efficient Algorithm for Exploiting,” *IBM J. Res. Dev.*, no. January, pp. 25–33, 1967.

[8] J. Gonzalez and A. Gonzalez, “Potential of data value speculation to boost ILP,” *Proc. Int. Conf. Supercomput.*, pp. 21–28, 1998, doi: 10.1145/277830.277840.

[9] F. Gabbay, “Speculative Execution based on Value Prediction,” *Electr. Eng.*, 1996.

[10] F. Gabbay, “Speculative Execution based on Value Prediction Research Proposal towards the Degree of Doctor of Sciences,” 1996.

[11] and J. P. S. Mikko H. Lipasti, Christopher B. Wilkerson1, “Value Locality,” 1996.

[12] D. M. Tullsen and J. S. Seng, “Storageless Value Prediction Using Prior Register Values,” vol. 00, no. c, pp. 270–279, 1999.

[13] Y. Sazeides and J. E. Smith, “Implementations of Context-Based Value Predictors,” *Tech. Rep. ECE-97-8*, 1997.

[14] K. Wang and M. Franklin, “Highly accurate data value prediction using hybrid predictors,” *Proc. Annu. Int. Symp. Microarchitecture*, pp. 281–290, 1997, doi: 10.1109/micro.1997.645819.

[15] “Championship Value Prediction (CVP).” https://www.microarch.org/cvp1/index.html.

[16] I. Irisa, “Exploring value prediction with the EVES predictor ∗,” 2019.

[17] Y. Ishii, “Context-Base Computational Value Prediction with Value Compression,” *CVP - Championsh. Value Predict.*, pp. 2–7, 2018.

[18] K. Koizumi, K. Hiraki, and M. Inaba, “H3VP : History Based Highly Reliable Hybrid Value Predictor,” *CVP - Championsh. Value Predict.*, no. 1, pp. 3–6, 2018.

[19] H. Zhou, C. ying Fu, E. Rotenberg, and T. Conte, “A study of value speculative execution and misspeculation recovery in superscalar microprocessors,” *Tech. report, ECE Dep. N. C. State Univ.*, pp. 1–23.

[20] A. Perais and A. Seznec, “Practical data value speculation for future high-end processors,” *Proc. - Int. Symp. High-Performance Comput. Archit.*, pp. 428–439, 2014, doi: 10.1109/HPCA.2014.6835952.

[21] A. Seznec and P. Michaud, “A case for (partially) TAgged GEometric history length branch prediction,” 2006.

[22] B. Sinharoy *et al.*, “IBM POWER8 processor core microarchitecture,” *IBM J. Res. Dev.*, vol. 59, no. 1, pp. 1–21, 2015, doi: 10.1147/JRD.2014.2376112.

[23] Intel, “Intel \textregistered{}64 and IA-32 Architectures Optimization Reference Manual, pp.366-369,” no. September, pp. 366–369, 2019, [Online]. Available: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[24] “The Sniper Multi-Core Simulator.” https://snipersim.org/w/The\_Sniper\_Multi-Core\_Simulator.

השימוש במנגנון חיזוי ערכים כבר הראה את הפוטנציאל לשיפור ביצועים במעבדים המבוססים על ארכיטקטורת OOO. למרות פוטנציאל הזה, בגלל צריכת חשמל גבוהה וקושי בביצוע יעיל בחומרה טרם בוצע שימוש של מנגנון חיזוי ערכים במעבדים מהדורות האחרונים. בעבודה זו נציג את שיטת uVP המאפשרת חיזוי ערכים תוך כדי שמירה על תכונות מיקרו-ארכיטקטוניות כמו מיקרו-פקודות באופן כללי ובזיכרון מטמון של מיקרו-פקודות באופן ספציפי. באמצעות כך, נוכל להרוויח את מירב הפוטנציאל של שימוש במנגנון חיזוי ערכים תוך כדי שמירה על צריכת חשמל נמוכה, גדילה מינימלית בשטח וסיבוכיות נמוכה בביצוע התכנון בחומרה. עבודה זו מציעה אפשרות לבצע אינטגרציה בין מנגנון חיזוי ערכים לבין זיכרון מטמון של מיקרו-פקודות כך שיהיה אפשר להמשיך לנצל את חיסכון החשמל שלו תוך כדי קבלת שיפור הביצועים של מנגנון חיזוי ערכים. בנוסף, בעבודה זו נתאר את המנגנון החדש, נדון במאפיינים ונבצע השוואה מול שני מנגנוני חיזוי ערכים קיימים, הראשון בשם LVP (Last Value Predictor) אשר ישמש כהשוואה למנגנון פשוט, והשני VTAGE מנגנון חדיש ומורכב שהראה פוטנציאל ביצועים גדול. כמו כן, נראה שמנגנון האינטגרציה המוצע מאפשר ביצוע יעיל של מנגנון חיזוי ערכים ללא הפסד של ביצועים ונשווה עם מנגנוני חיזוי הערכים הקיימים.

# תקציר

כיום יש דרישה גדולה לשיפור הביצועים במערכות מחשב, שיפור ביצועי המחשב יכול להיות מושג על ידי שימוש בהרבה יחידות עיבוד (many core architectures) או על ידי שיפור ביצועי יחידת עיבוד בודדת. הדרכים המוכרות על מנת לשפר ביצועי תהליכון בודד הן על ידי הגדלת תדר יחידת העיבוד על ידי שימוש בצינור וטכנולגיית ייצור טובה יותר, או על ידי הגדלת מספר הפקודות אשר ניתן לבצע בעבור כל מחזור שעון. כמו שניתן לראות במשוואה הבאה:

Equation 2 .

שימוש בארכיטקטורה מסוג OOO (Out Of Order) אשר מבצע את הפקודות לפי סדר התלויות בהם ולא לפי סדר כתיבתם, הוכיחה את יעילותה בהגדלת מספר הפקודות שניתן לבצע בעבור מחזור שעון בודד. הארכיטקטורה הנ"ל הומצאה במהלך שנות ה-90 ומאז נמצאת בשימוש נרחב בכל טכנולוגיית מעבד מודרני. ארכיטקטורה זו מאפשרת נצילות טובה יותר של יחידות הביצוע ותומכת בביצוע מקבילי של מספר פקודות באותו מחזור שעון, ובעצם על ידי כך משפרת את ביצועי המערכת ומאפשרת אופטימיזיה הרבה יותר טובה של השימוש בסיליקון.

תלויות מסוג RAW (Read-After-Write) ותלויות שיכולות להיווצר כתוצאה מחיזוי תוצאה של פקודות מסוג קפיצה מותנית, פוגעים ביכולות ארכיטקוטרת OOO, תלויות מסוג זה יכולים לקרות כאשר חישוב של פקודה מסויימת תלוי בתוצאה שמחושבת על ידי פקודה שקודמת לה. שימוש במנגנון חיזוי ערכים, מציע לשבור תלויות אלה ועל ידי כך לשפר את תכונת המקביליות של התכנית באופן שתומך בארכיטקטורת OOO ובצורה כזאת אף לשפר את ביצועי המעבד. מנגנון חיזוי ערכים מבצע ניחוש של ערך התוצאה של מספר פקודות באמצעות עיקרון הלוקאליות של הערכים בתכנית ועל ידי כך מאפשר לפקודות התלויות בערכים אלה לעבור לשלב הביצוע עוד בטרם התוצאה האמיתית חושבה. עקב כך, ערך החיזוי נמצא בשימוש עוד טרם הערך האמיתי חושב, כל הפקודות אשר השתמשו בערך זה נחשבות ככאלה שרצות באופן ספקולטיבי ולכן אסור לערך החישוב שלהן להשתקף כלפי המשתמש, אלא רק באופן פנימי עד אשר הערך האמיתי יחושב. אם ערך החיזוי הוא נכון, אז כל הפקודות התלויות בו ורצו באופן ספקולטיבי יכולות להשתקף כלפי המשתמש באופן מיידי, ועל ידי כך להרוויח האצה במהירות עקב הביצוע המוקדם של הפקודות. אבל, כאשר החיזוי שגוי, כל הפקודות התלויות שבוצעו באופן ספקולטיבי חייבות להישלח לביצוע מחדש עם הערכים הנכונים ובעצם לגרום לפגיעה בריצת התכנית ובביצועיה. על מנת להיות בטוחים ביעילות מנגנון החיזוי, צריך להבטיח כי ההאצה הממוצעת גדולה ממחיר הפגיעה הממוצע ככה שבממוצע הכללי התכנית תקבל האצה.

עבודה זאת בוצעה בהנחיית פרופ. אבי מנדלסון, בפקולטה להנדסת חשמל ע"ש אנדרו וארנה ויטרבי, טכניון.

ברצוני להודות למספר אנשים שעזרו לי במהלך המחקר והפכו אותו לאפשרי:

1. תודה לפרופ. אבי מנדלסון על ההנחיה המסורה ועל כל העזרה.
2. תודה לטכניון והפקולטה להנדסת חשמל ע”ש אנדרו וארנה ויטרבי, בהם ביצעתי את המחקר.
3. תודה לאשתי, אור, על הסבלנות והתמיכה במשך כל התקופה הזאת.

**uVP—שיטה חדשה ליישום יעיל של חיזוי ערכים בארכיטקטורות מעבדים המבוססות על מטמון מיקרו-פקודות**

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת תואר מגיסטר למדעים בהנדסת חשמל

**דן רצ'ר**

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

תשרי תשפ”א חיפה ספטמבר 2020

**uVP—שיטה חדשה ליישום יעיל של חיזוי ערכים בארכיטקטורות מעבדים המבוססות על מטמון מיקרו-פקודות**

**דן רצ'ר**