uVP Design  
Layan Jarjoura

Contents

[1. uOp Cache: 2](#_Toc111211676)

[1.1. Structure: 2](#_Toc111211677)

[1.2. How it works: 2](#_Toc111211678)

[1.3. Dan’s Design: 3](#_Toc111211679)

[2. Simple Prediction in RISC: 3](#_Toc111211680)

[2.1. How it works: 3](#_Toc111211681)

[2.2. VP with uOp Cache: 4](#_Toc111211682)

[2.3. Dan’s Design: 4](#_Toc111211683)

[3. ROB-VP Interface (Deployment Phase): 7](#_Toc111211684)

[3.1. ROB: 7](#_Toc111211685)

[3.2. Initial Suggested Design: 7](#_Toc111211686)

[3.3. Our Suggested Design: 8](#_Toc111211687)

[3.4. Enhanced Design: 9](#_Toc111211688)

[3.4.1. Inter-Block Dependencies issue: 9](#_Toc111211689)

[3.4.2. Strided values: 9](#_Toc111211690)

[4. Instructions to Value-Predict: 11](#_Toc111211691)

[4.1. Focused Value Prediction Paper: 11](#_Toc111211692)

[4.2. BeBoP Paper: 11](#_Toc111211693)

[5. Training Phase: 12](#_Toc111211694)

[6. Meetings Videos: 14](#_Toc111211695)

# uOp Cache:

# Structure:

Each **Basic Block (BB)/Cache Line** has:

1. **One Entry**
2. **One Exit**
3. **Fixed maximum size** (can hold up to 8 uOps for example). We need two lengths: one for #uOps and one for #x86\_instructions.

If an x86 instruction is decoded into multiple uOps, then all of them need to be in the same basic block. This can create “gaps” in the basic block, but we live with that. In this case a basic block will be spread across multiple cache lines.

Each line of the uOp cache has a **tag** which implies the x86 entry instruction address, the x86 exit instruction of the basic block, and the maximum length/size of the line.

If we get to a certain x86 instruction that doesn’t have a basic block which *starts* with it, then we build a new basic block for it (even if there are basic blocks that include it. Because a basic block has only one entry). We might get basic blocks with overlapping instructions, but it’s not an issue.

Tag **Instruction 0 Instruction 1 Instruction 2** …

uOp uOp uOp uOp …

BB1:

Tag **Instruction 0 Instruction 1 Instruction 2** …

uOp

BB2:

In the figure above, if we jump to instruction 1 of BB1 directly, we can’t access the BB. So, we build a second one (BB2) that starts with that instruction (instruction1 of BB1 is instruction 0 of BB2).

The instructions in the bold orange are x86 instructions and the operations in the thin orange lines are the uOps after decoding.

A uop cache block can end when:

1. We get to a jump/branch instruction.
2. The next x86 instruction (decoded to uOps) doesn’t fit in the remaining part of the BB.

# How it works:

Each fetched instruction’s address gets compared to the tags of the BBs (entry addresses). If it’s a hit, we take the whole basic block and instead of decoding each of the x86 instructions, we use the pre-decoded uOps in the BB, which then are allocated in the ROB. If it’s a miss, we decode the x86 instruction to uOps as usual, and simultaneously issue them to the OoO execution engine and build a new decoded BB to be allocated in the uop cache.

|  |  |  |
| --- | --- | --- |
| In order (Front End) | OoO (Back End) | In Order |
| Fetch and decode | Execute | Commit |

The division of the processor is shown in the figure above. Fetch and Decode stages are executed in the front end of the processor, in order. The execution is in Out of Order. The ROB connects the in-order with the OoO. The commit at the end is back to in order.

When a basic block we built is ready it is allocated into the uOp Cache . Every block includes the entry and exit addresses of the x86 instructions.

We commit the complex of the uOps that constitute an x86 instruction together and in order when theyretire execution.

# Dan’s Design:

Assumptions made:

1. uOp Cache size is infinite.

**ToDo:** We, instead, need to manage and simulate a cache. Shouldn’t be hard, Freddy has an example that he implemented. We need to store the x86 entry and exit address. For statistical purposes, we also need to store the number of instructions/uOps. Need a sensitivity stufy on the size of the cache too. Issue 8 in Google Sheets. How many ports do we want for the cache? Issue 4 in Google Sheets.

# Simple Prediction in RISC:

# How it works:

Typically, the **PC** (Program Counter) is used as a lookup key in the value predictor table.

When an instruction is fetched, we have its PC. We use it to look up whether it has a prediction in the value predictor. If it is a hit (there’s a prediction), we turn on a bit in the ROB indicating that a speculative value was provided by the value predictor for the outcome of the instruction (Note that the instruction may not be speculative at all). In case of case of miss, it is processed normally. See example below:

|  |
| --- |
|  |
| R1🡨1 |
| R2🡨R32 |

R1🡨1 ROB:

R2🡨R1

R32<-->R1 (Renaming)

When result of R1/R32 is ready in the RAT, it gets forwarded to R2. In VP, we predict a value for R32 and R2 uses it while turning on a bit implying its result is speculative so we can execute R1 and R2 in parallel. When the R1 instruction is executed, and the result matches the predicted value, then the speculative bit of the second instruction is turned off. If, on the other hand, the predicted value is wrong, we need to figure out what to do. Usually, it’s a full flush.

# VP with uOp Cache:

The problem:

Once we put the instructions/uOps in the uOp cache, we no longer have x86 instructions which means we don’t have their PC. But we need their PC to perform value prediction.

Naïve Solution:

Save the offset of each instruction in the BB from the entry. This complicates things because we need to calculate (ALU) and we need to take care of timing, etc. Not efficient.

# Dan’s Design:

Two ideas:

1. Putting most of the responsibility on the thing that builds the BB. How?

By adding a buffer that holds the predicted value to each uOp in the BB in the uOp Cache, and also a “speculative” bit. **While** building the BB for the uOp Cache, we store **values** **in** those **buffers** and turn on the speculative bit, which saves us the need to store the dependencies for the instruction.

In Dan’s design, the plan was to predict **only one value** for each BB (use only one buffer).

This mechanism **saves data movement** from and to the value predictor. When VP is wrong, we flush the BB and the following instructions, and we retrain the BB.

1. Instead of a buffer per uOp, we keep a table/”dictionary” – called **Value Renaming Table** - in a central place, thus saving area because if more than one BB use the same value, this value is stored only once. Also, we might be able to manage a stride value prediction here.

The BB looks like this (in black), and we decide to add a “speculative” bit (in red):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opc | R\_source | R\_dest | … | s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opc | R\_source | R\_dest | … | 0 |

When “s” is 0, it’s a regular entry.

|  |  |  |  |
| --- | --- | --- | --- |
| Pointer | R\_dest | “Might have extra space left” | 1 |

When “s” is 1, we can “piggy-back” on the existing entry and save a pointer to the predicted value in the dictionary (together with the R\_dest only, because we might use it for the dependency chain)

Dictionary

Layan – I see a problem here – if the same uop resides in 2 uop cache entries then obviously they both need to point to the same entry in dictionary. The problems that I see are as follows:

1. Since we use the BB address + uop offset in the BB as a lookup key in the dictionary then the duplicated uop will have 2 different keys. This can work as long as you have a last value predictor, however if you use a context or stride predictor this will be a problem.
2. We allow only one speculative uop in a BB (for now). If a duplicated uop is marked as speculative in one BB how will you know in the second BB that it is speculative?

Assumptions:

1. Note: We might want to commit the **whole BB** at once (if both value and branch predictions are correct), or none of its uOps (if the prediction is wrong). The flush is performed from the problematic BB **and forward**. Currently, we don’t do that.

**TODO**: Decide on **misprediction** handling: **Flush**? Save **Checkpoints**? Issue 21+24.

1. Dan’s Design assumes a fixed value for predictions.

**TODO**: Find a mechanism to handle strided values (**context based**), like a **pointer** in the **dictionary** that can update the values with an offset. Another suggestion is the use of DVTAGE. Issue 5.

1. Training happens in the **slow path**. And the Value Predictor has a pointer to the buffer in the uOp Cache.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
| Opc | R\_source | R\_dest | … | Predicted Value | s |

**Value Predictor**

**TODO**: We don’t like this idea. Need to rethink it. Maybe add another bit that says whether we need re-training of the value, and if it’s on, we rebuild the BB in a temporary buffer while we keep using the old one in the meanwhile. (**I didn’t understand this idea very well**). Issue 9.

1. We don’t want the value prediction to happen in the critical path. Only move the confidence-saturated values to the fast path.

**TODO**: Interface between Slow and Fast path. Issue 15+16.

In addition, **optimization**. For example, when the value of the R\_source is zero or whatever. Issue 22.

1. Dan’s design assumes one write port for the predictor.

**TODO**: Think about how many we want/ how many concurrent values we want to predict.

# ROB-VP Interface (Deployment Phase):

# ROB:

ROB has a CAM-based part (has register number for example=0) and an SRAM-based part (has the value of the registers for example). We snoop on the address bus, CAM-based, and if it matches, we copy the value from the value bus.

ROB options:

1. Keep an SRAM/Registers that has all speculative values and a comparator on the broadcast bus to compare the speculative value with the real one after execution.
2. Keep speculative values in CAM. Each entry has speculative value + relevant register number + Checkpoint in case of misprediction. (For validation stage)

# Initial Suggested Design:

**Dictionary**

\_[PredictedValue|PhyReg]\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

EXE

**Value predicted from uOp Cache**

A\_bus

V\_bus

**Comparator**

Provide values

snooping

**ROB**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ALU | Opc |  |  |  |  |
| Address/value | Rs1 |  |  |  |  |
| Address/value | Rs2 |  |  |  |  |
| [can get predicted value] | Rd |  |  |  |  |
| 1=Result or 0=dependency | v |  |  |  |  |

One uOp

If comparator result = match, means the ROB has correct predicted values and it’ll continue working as usual. If not, we’ll need to flush to checkpoint (entry of BB) and retrain.

# Our Suggested Design:

One whole BB

**uOpCache**

t1 add r1\*, r2, r2

t2 sub r7, r1, r5

t3 and r9, r1, r8

Branch



**Dictionary (VRT)**

[d\_idx|SpecValue|BB|stride..] \_[d1 | val1 | 0x1000 | …] \_[d2 | val4 | 0x2000 | … ] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

snooping



**EXE units**

R1

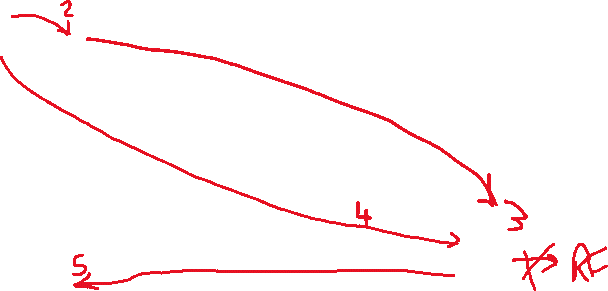
Val2

A\_bus

V\_bus

snooping

snooping



**ROB Builder**

**ROB**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Op | Rs1 | Rs2 | Rd | Real Val (from Exe) | … |
| Add | p2 | p2 | D1 |  |  |
| Sub | val1 | p5 |  |  |  |
| And | val1 | p8 | **Comparator** |  | **RAT**  R1<-> val1 / rob.predVal ptr \_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Life of a BB with VP and uOp Cache:

1. uOp Cache has a BB. Registers that we want to predict (in a value-generating instruction), will have a special “speculative” bit (symbol \* in figure above).
2. Register renaming (connecting r2 with p2) and Value renaming (Connecting r1 with d1 if predicted value existed with confidence in the VP).
3. The fetched BB from uOp Cache is allocated into the RoB with the corresponding predicted values from dictionary/RAT (to find the predicted values for the registers in the uOps in uOp Cache, we look-up from the dictionary using: **BB address and uOp offset**.)
4. Execute the operations (normally) to get the real results.
5. Forward the result and broadcast it on the A\_bus (Address) and V\_bus (Value) busses.
6. The *ROB* snoops on the busses and gets the calculated values (val2) and *compares* them to the predicted value it has (val1 vs val2), *in order* (commit stage). If they match, everything continues as is, else, we need to flush to last checkpoint. ROB needs comparator units, and it needs to signal to the dictionary in case of flush.

Notes:

* Architectural registers in uOp Cache. Physical Registers in Dictionary and ROB.
* Dictionary entries Eviction/Replacement policy: On mis-prediction AND when dictionary gets full.

The following is relevant only for contexts /strides:

* + We can evict a dictionary entry only if the uOp that produced the value has already committed and the dependent uOps already read the predicted value from the dictionary entry.
  + In case of loop, need to recognize the moment we finish the loop and get a misprediction 🡪 reset the value in entry, don’t evict.
  + If mispredict in first value 🡪 evict. (Need to add a bit for denoting first/last iteration).
  + We can start the simulations with a dictionary structure similar to a cache structure.
* Flushing needs to be ordered. If one uOp was mispredicted, we don’t flush immediately, we wait for all the previous uOps to finish/find a previous flush. We accomplish that by using the ROB and the same mechanism as the branch prediction. This flush is different from Dan’s because we rollback to entry of last committed BB.

Flush procedure:

1. Flush instruction at the commit point.
2. Flush corresponding dictionary entry
3. Move instruction back to training

**TODO**: We still don’t know who fills the dictionary. 🡪 Training phase explains this.

**TODO**: Who picks which uOps’ Rs are going to get the predicted values? (\* in pic) Need to agree on *eligibility rules*.

# Enhanced Design:

# Inter-Block Dependencies issue:

Solved using the RAT to host the most recent predicted value.

# Strided values:

The Value Predictor will provide different values each time and we’ll have a single entry in the dictionary for all of them, and we’ll verify the value with the different instructions in the ROB.

Assumption: the look-up in the dictionary is in-order.

Mechanism: Single entry in dictionary for each strided instruction. The dictionary will hold the last value and stride. The predicted value will be copied normally to the RAT. After each lookup made by a *value-generating* instruction, the last value in the dictionary will be updated as follows: Last Val = Last Val + stride. Since the look-up in the dictionary is in-order, the incremented predicted values will be increased each time by a fixed stride. Value prediction checking will be done at the ROB at the commit stage, in order. Value mis-predictions will flush the corresponding entry from the dictionary and the uOp will be downgraded to the training phase.

A general comment:

1. We suggest that the lookup in the dictionary will be performed in an earlier pipeline stage with respect to the VP.
2. Only in case of a dictionary miss (and the uop is eligible for VP), we will perform a lookup in the value predictor.

(= When an instruction is value-predicted, first we look-up in the dictionary. If no hit found, we look-up in the value predictor itself. If no hit, we add it to the VP and start training it, else, only the confidence mechanism is updated).

# Instructions to Value-Predict:

# Focused Value Prediction Paper:

Targeted instructions:

1. Delinquent loads that miss the last level cache (LLC) or branches that are wrongly speculated.
2. Value prediction has traditionally targeted only data-dependencies arising through registers. However, data dependencies also emerge from memory because of stores and loads to the same memory address. If the store to load forwarding can be accurately predicted for this load, then there is no need to predict any other register dependency that may be needed to calculate the address of the load.
3. Many load instructions do not always access constant data, and hence value predictors rely on large, expensive tables based on program context to predict them. We observed that a big fraction of such loads is actually forwarded by prior stores, necessitating storing and predicting different data for each program context. By incorporating memory dependence learning, our proposal eliminates the need of using context value prediction for such store forwarded loads, and thereby reduces the size of context tables that are needed for value prediction. A large body of research already exists for accurately predicting memory dependencies, and we can leverage those works in our value predictor

# BeBoP Paper:

* + - 1. Load immediates: If write ports are available at dispatch time to write predictions to the PRF, then there is no need for Load Immediates to be predicted since their actual result is available in the frontend. The predictor need not be trained for these instructions, and they need not be validated or even dispatched to the IQ. They can be processed in the front-end by simply placing the decoded immediate in the PRF.
* Difference between our paper and bebop: We have a Directory (while they keep their values in the predictor and do regular register renaming) and uOpCache.

# Training Phase:

* Placement: (3)

The snooping of the VP on the busses can be done right after Exe Stage or right at commit stage (After ROB so the uOps can be in order). As a first step, we’ll go with the commit stage.

* Start of Training: (1)

Suggestion is to start training only after the BB is already in the uOp Cache.

* Train on:

For simplicity, predict every load and simple add/sub in program, then check VP accuracy. We’ll complicate things later.

* Stop Training: (4)

When the confidence of a predicted value is high.

If the confidence doesn’t saturate on a high confidence after a certain amount of time or occurrences of the specific instruction, we evict the entry.

When the VP encounters a uOp (ROB entry) for the first time, its real value is snooped from the busses and a confidence counter is initialized.

On the second encounter, the ROB entry is updated, get real value again for comparison, and update confidence.

When confidence is saturated, flush entries from VP and copy values to Dictionary.

* Access the VP using: (2)

BB entry address + uOp offset + history/whatever the specific predictor needs.

An issue – there might be multiple BB entry addresses so this might lead to a duplication of the same instruction in the VP predictor. If we use last value prediction it is not a problem.

up offset is needed when we predict more than one value in a BB.

For now, we only predict one value per BB and decide on a priority to pick one instruction between the load/sub/add in a BB.

* VP fields:

The lookup key to the VP is PC address and uOp offset. When confidence is saturated for a certain instruction, we get its address from the lookup key and copy it to the relevant field in the dictionary.

Diagram of the flow:

Fetch BB into uOpCache

THEN start training.

**uOpCache**

**t1 add r1, r1, 3**

**t2 add r1, 100, 0**

**t2** Branch \_\_\_

Allocate entry for uop. Initialize conf=0. Lookup key is BB entry + uop offset+?

**Value Predictor**

BB |up offset | Value | Confidence=0 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

VP snoops on the busses right after **commit** stage (uops are in order)

A\_bus

V\_bus

On every encounter of the uop, get real value again for comparison, and update confidence.

When confidence is saturated, flush entries from VP and copy values to Dictionary.

If the confidence doesn’t saturate on a high confidence after a certain amount of time or occurrences of the specific instruction, we evict the entry.

# Possible design flows:

**UopCache** HIT MISS

**VRT** HIT MISS ~~HIT~~ MISS

**VP** HIT MISS

There are 5 legal paths.

MISS🡪MISS🡪MISS: green/bluish checked in log

HIT🡪MISS🡪HIT: yellow checked in log

HIT🡪HIT: blue checked in log

# Meetings Videos:

1. <https://panoptotech.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=95ce21d7-caf9-41c1-95f2-ae8a007f0a33>
2. C:\Users\layanj\OneDrive - NVIDIA Corporation\Documents\zoom\2022-05-19 11.37.57 layan jarjoura's zoom meeting
3. <https://technion.zoom.us/rec/share/QdjOAmbbNnjCsHFOagntPT_vYesUt8Wt3rCGhhMcBF2iYzFf85XL29UI_BReUfM-.e7ecu9IhVaWJU08G?startTime=1654699896000> (Passcode: v=Bw3mS! )
4. <https://panoptotech.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=1aa9f482-badd-48ea-89e1-aec000ce54e7>