uVP Design  
Layan Jarjoura

Contents

[1. uOp Cache: 2](#_Toc106295360)

[1.1. Structure: 2](#_Toc106295361)

[1.2. How it works: 2](#_Toc106295362)

[1.3. Dan’s Design: 3](#_Toc106295363)

[2. Simple Prediction in RISC: 4](#_Toc106295364)

[2.1. How it works: 4](#_Toc106295365)

[2.2. VP with uOp Cache: 4](#_Toc106295366)

[2.3. Dan’s Design: 4](#_Toc106295367)

[3. ROB-VP Interface (Deployment Phase): 7](#_Toc106295368)

[3.1. ROB: 7](#_Toc106295369)

[3.2. Initial Suggested Design: 7](#_Toc106295370)

[3.3. Our Suggested Design: 8](#_Toc106295371)

[3.4. Enhanced Design: 9](#_Toc106295372)

[3.4.1. Enter-block dependencies: 9](#_Toc106295373)

[3.4.2. Strided values: 10](#_Toc106295374)

[4. Instructions to Value-Predict: 11](#_Toc106295375)

[4.1. Focused Value Prediction Paper: 11](#_Toc106295376)

[4.2. BeBoP Paper: 11](#_Toc106295377)

[5. Training Phase: 12](#_Toc106295378)

[4. Meetings Videos: 12](#_Toc106295379)

# 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.

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 basic block can end when:

1. We get to a jump 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 the x86 instructions one by one, we use the pre-decoded uOps in the BB, which then are sent to the ROB. If it’s a miss, we decode the x86 instruction to uOps as usual, and simultaneously send them to the OoO part and build a new BB with them.

|  |  |  |
| --- | --- | --- |
| In order (Front End) | OoO (Back End) | In Order |
| Fetch and decode | Reorder, Execute | Write Back, 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, we enter it into the uOp Cache using Sets, Tages and Ways (like all other caches) extracted from the Tag field of the BB which includes the entry and exit addresses of the x86 instructions.

In a more complicated version, we have a confidence level for the uOp Cache entries. We add them to the cache only when we see the same instructions/BB more than once, so we don’t waste space in the cache.

We commit the complex of the uOps that constitute an x86 instruction together when they’re all done executing.

# 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:

A prediction connects a **PC** (Program Counter) to a **register**.

When an instruction enters the ROB, we have its PC. We use it to look up whether it has a prediction in the value predictor. If it’s a hit (there’s a prediction), we turn on a bit in the ROB that the says the instruction is *speculative*. If not speculative, we store something that implies its dependencies using Register Renaming. 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” 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 strided 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

Assumptions:

1. We 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**.

**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.

**Value Predictor**

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

**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, r3

t2 sub r7, r1\*, r5

t3 and r9, r1\*, r8

Branch

**Dictionary (VP-RF)**

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

snooping

**EXE units**

R1

Val2

A\_bus

V\_bus

snooping

snooping

**Comparator**

**ROB Builder**

**RAT**

R1<->d1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_

**ROB**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Op | Rs1 | Rs2 | Rd | … |
| Add | p2 | p2 | T1 |  |
| Sub | D1 (val1) | p5 | T2 |  |
| And | D1 (val1) | p8 | T3 |  |

Life of a BB with VP and uOp Cache:

1. uOp Cache has a BB. Registers that want to use a predicted value (dependent on a previous operation result), 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 ROB builder fetches BB from uOp Cache while getting relevant predicted values from dictionary and fills the ROB with them (to find the predicted values for the registers in the uOps in uOp Cache, we look-up from the dictionary using BB address and physical registers names using RAT or BB address and uOp offset).
4. Execute the operations to get the real results.
5. Forward the real result and broadcast it on the A\_bus (Address) and V\_bus (Value) busses.
6. The dictionary snoops on the busses and gets the real calculated values (val2) for the registers it predicts and compares them to the predicted value (val1 vs val2). If they match, everything continues as is, if they don’t, we need to revert to last checkpoint (flush) and retrain.

Or

The ROB snoops on the busses and gets the real calculated values (val2) and compares them to the predicted value it has (val1 vs val2). 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 provides values to the ROB builder, not to the snoop bus.
* Inter-block dependencies can be solved by using BB + phy Reg in dictionary.
* Dictionary entries Eviction/Replacement policy: On mis-prediction AND when dictionary gets full.
  + 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. This flush is different from Dan’s because we rollback to entry of last committed BB.

**TODO**: We still don’t know who fills the dictionary. We’ll talk later about this.

**TODO**: Who picks which uOps’ Rs are going to get the predicted values? (\* in pic)

# Enhanced Design:

# Enter-block dependencies:

Issue:

1. r1🡨5
2. if r2==9
   1. r1🡨7
   2. r2🡨r1
3. else
   1. r2🡨 r1 + 3

The issue with our design is that the dictionary would save the value of 5 for r1 at first, then it would override it with 7 and use it in instruction 2.b if the condition was satisfied. But, if the “if” condition wasn’t satisfied, then we want to use the value of 5 for instruction 3.a.

2.a and 2.b would be in BB1, and 3.a would be in BB2. The register renaming would give different physical registers to r1 in different BB, let’s say p1 for r1 of BB1 and p2 for r1 in BB2. To follow the dependencies and reuse the value of r1, **the dictionary needs to save the BB entry address so we can match the predicted value to the physical address, and the ROB needs to save the relevant dictionary entry (d3 for example for dictionary entry number 3) for R1 of the particular BB, thus no need to save the physical register in the dictionary anymore**.

# Strided values:

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

* Compare two mechanisms for thesis:

1. Multiple entries in dictionary for strided cases – need to decide on number of strided values to predict because we don’t have infinite space.
2. Single entry in dictionary + copy values in ROB. (Reasoning: Dictionary is not aware of the OoO of the execution. In case of misprediction in branch or value, we might receive this mispredition info OoO to the dictionary).

TODO: How will the VP provide different values each time without hitting 100% miss-rate?

# 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:

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:

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:

When the confidence of a predicted value is high.

* Access the VP using:

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

# 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! )