uVP Design  
Layan Jarjoura

Contents

[1. uOp Cache: 2](#_Toc102734424)

[1.1. Structure: 2](#_Toc102734425)

[1.2. How it works: 2](#_Toc102734426)

[1.3. Dan’s Design: 3](#_Toc102734427)

[2. Simple Prediction in RISC: 3](#_Toc102734428)

[2.1. How it works: 3](#_Toc102734429)

[2.2. VP with uOp Cache: 4](#_Toc102734430)

[2.3. Dan’s Design: 4](#_Toc102734431)

[3. Meetings Videos: 6](#_Toc102734432)

# 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 statistic purposes, we also need to store the number of instructions/uOps.

# 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**?

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.

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**).

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.

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

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.

# Meetings Videos:

1. <https://panoptotech.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=95ce21d7-caf9-41c1-95f2-ae8a007f0a33>