EOLE: Toward a Practical Implementation of Value Prediction

A new architecture, Early/Out-of-Order/Late Execution (EOLE), leverages value prediction to execute a significant number of instructions outside the out-of-order engine. This approach reduces the issue width, which is a major contributor to both out-of-order engine complexity and the register file port requirement. This reduction paves the way for a truly practical implementation of value prediction.

Because legacy code wasn’t written with parallelism in mind and because of the existence of intrinsic sequential parts in parallel code, uniprocessor performance is—and will long remain—a major issue for the microprocessor industry.

Modern superscalar processor design leverages instruction-level parallelism (ILP) to get performance, but ILP is intrinsically limited by true (read after write [RAW]) dependencies. Moreover, scaling the hardware taking advantage of potential ILP (for example, the scheduler or reorder buffer [ROB]) is nontrivial because those structures are complex—thus impacting CPU cycle time and pipeline depth—and power hungry.

As a result, alternative ways to increase sequential performance that were previously proposed but eventually rejected as the multicore era began might now be worth revisiting. Among these methods is value prediction (VP), which increases performance by removing some RAW dependencies, hence increasing ILP and making better use of the instruction window. However, proposed implementations were considered too costly, because they introduced substantial hardware complexity and additional power consumption in almost every pipeline stage. As a result, VP went out of fashion in the early 2000s.

Nonetheless, recent work in the field of VP has shown that given an efficient confidence estimation mechanism, prediction validation can be removed from the out-of-order (OoO) engine and delayed until commit time. As a result, we can avoid having to recover from mispredictions via selective reissue and use a much simpler mechanism—pipeline squashing—while the OoO engine remains mostly unmodified.

Yet, VP and validation at commit time entails strong constraints on the physical register file (PRF). Write ports are needed to write predicted results and read ports are needed to validate the results at commit time, potentially rendering the overall number of ports unbearable. Fortunately, VP also
implies that many single-cycle arithmetic logic unit (ALU) instructions have their operands predicted in the front end and can be executed in place, in order. Similarly, the execution of single-cycle instructions whose result has been predicted can be delayed until commit time, because predictions are validated at commit time.

Consequently, a significant number of instructions—5 to 50 percent in our experiments—can bypass the OoO engine, allowing issue width reduction, which is a major contributor to both OoO engine complexity and the register file port requirement. This reduction paves the way for a truly practical implementation of VP. Furthermore, because VP in itself usually increases performance, our resulting Early/OoO/Late Execution (EOLE) architecture is often more efficient than a baseline VP-augmented six-issue superscalar while having a significantly narrower four-issue OoO engine.

Enabling primary work

Many value predictors have been proposed in the literature. However, to capture more refined patterns, predictors became more and more complex. This limited their potential implementation on real hardware. Regardless, when it was introduced, VP mainly suffered from the need for a complex repair mechanism to handle mispredictions.

Designing a practical predictor

Because of their operation mechanism, finite-context-method predictors aren’t simple enough to be implemented easily. In particular, they are unlikely to be able to provide predictions for two back-to-back instances of the same static instruction. In other words, part of the potential for predicting values would likely be lost with those predictors. However, recent advances in the field of branch target prediction can be leveraged to design better value predictors.

In particular, because indirect target prediction is a specific case of VP, we introduced the Value Tagged Geometric (VTAGE) predictor. We derived this predictor directly from research propositions on branch predictors and, more precisely, from the indirect target branch predictor, ITTAGE. VTAGE is the first hardware value predictor to leverage a long global branch history and the path history. VTAGE outperforms previously proposed context-based predictors such as the finite-context-method predictors.

Unfortunately, VTAGE isn’t suitable for predicting strided patterns, and these patterns occur frequently in programs (for example, such patterns as loop induction variables, regular data layout, and so on). In particular, each value in the pattern requires its own entry in VTAGE, whereas the whole pattern can be captured by a single entry in a common stride predictor. As a result, there is a case for hybridizing both predictors. However, a naive hybrid having the two components side by side would require a great amount of storage, because each component must be big enough to perform well. The fact that VTAGE stores 64-bit values in all its components only exacerbates this issue. That is, VTAGE’s storage footprint can’t easily be reduced, or at least not without greatly impacting its performance (for example, by implementing fewer entries).

As a result, we introduced the Differential VTAGE predictor, or D-VTAGE. This tightly coupled hybrid combines a VTAGE predictor storing smaller strides in its components and a last value table (see Figure 1). To make a prediction, this method generates a stride using the VTAGE prediction scheme and adds it to the last outcome located in the last value table. The size of D-VTAGE can easily be reduced to a reasonable amount (for example, from 16 to 32 Kbytes) by using small strides.

Removing prediction validation from the execution engine

Two hardware mechanisms are commonly used in processors to recover from mispeculation: pipeline squashing and selective reissue. These mechanisms induce different average misprediction penalties, but they also differ greatly in hardware complexity when considered for value misprediction recovery.

Pipeline squashing is already implemented to recover from branch mispredictions. On a branch misprediction, all the subsequent instructions in the pipeline are flushed and instruction fetch is resumed at the branch target. This mechanism is also
generally used on load/store dependency predictions. Using pipeline squashing is straightforward but costly, because the minimum value misprediction penalty is now the same as the minimum branch misprediction penalty. However, we can avoid squashing if the predicted result hasn’t been used yet, that is, if no dependent instruction has been issued.

Selective reissue is implemented in processors to recover in cases where instructions have been executed with incorrect operands, in particular to recover from level-1 (L1) cache hit/miss mispredictions (that is, load-dependent instructions are issued after predicting a L1 hit, but the load results in a L1 miss). When an instruction execution with an incorrect operand is detected, the instruction as well as all its dependent chain of instructions are canceled then replayed.

Validation at execution versus at commit time. Selective reissue at execution limits the misprediction penalty. However, pipeline squashing can be implemented either at execution or at commit time. Pipeline squashing at execution results in a minimum misprediction penalty similar to the branch misprediction penalty. However, validating predictions at execution necessitates redesigning the complete OoO engine: the predicted values must be propagated through all the OoO execution engine, and the predicted results must be validated as soon as the actual results are computed. Moreover, the repair mechanism must be able to restore the processor state for any predicted instruction. Regardless, prediction checking must also be implemented in the commit stage(s), because the value predictor must be trained even when predictions weren’t used due to low confidence.

On the contrary, pipeline squashing at commit time results in a high average misprediction penalty, because it can delay prediction validation by many cycles. Yet, this approach is much easier to implement for VP, because it doesn’t induce complex mechanisms in the OoO execution engine. Pipeline squashing essentially restraints the VP-related hardware to the in-order pipeline front end (prediction) and the in-order pipeline back end (validation and training). Moreover, checkpointing the rename table isn’t needed, because the committed rename map contains all the necessary mappings to restart execution correctly.

A simple synthetic example. Realistic estimations of the average misprediction penalty $P_{value}$ could be 5 to 7 cycles for selective reissue, 20 to 30 cycles for pipeline squashing at execution time, and 40 to 50 cycles for pipeline squashing at commit. (Selective reissue includes tracking and canceling the complete chain of dependent instructions as well as the indirect sources of performance loss such as resource contention due to reexecution, higher misprediction rate, and lower prediction coverage.)

For the sake of simplicity, we will respectively use the following value misprediction penalties in the two examples that follow: 5 cycles for selective reissue, 20 cycles for pipeline squashing at execution time, and 40...
cycles for pipeline squashing at commit time. We assume an average benefit of 0.3 cycles per correctly predicted value (taking into account the number of unused predictions).

In our example, predictors achieve around 40 percent coverage and 95 percent accuracy, as is often reported in the literature. Hence, with 50 percent of predictions used before actual execution, the performance benefit when using selective reissue would be around 64 cycles per 1,000 instructions, a loss of around 86 cycles when using pipeline squashing at execution time and of around 286 cycles when using pipeline squashing at commit time.

**Balancing accuracy and coverage.** The total misprediction penalty $T_{reco}$ is roughly proportional to the number of mispredictions. Thus, if we were to drastically improve accuracy at the cost of some coverage, as long as the coverage of the predictor remains quite high, there might be a performance benefit brought by VP, even though the average value misprediction penalty is high.

Using the same example, but sacrificing 25 percent of the coverage (now only 30 percent), and assuming 99.75 percent accuracy, the performance benefit would be 88 cycles when using selective reissue, 83 cycles when using pipeline squashing at execution time, and 76 cycles when using pipeline squashing at commit time per 1,000 instruction cycles.

**Commit time validation and recovery**

Assuming very high accuracy, validating at commit is a valid approach from a performance standpoint. In the following section, we introduce the hardware implications of doing so.

**Hardware implications on the OoO engine.** In the previous section, we pointed out that the hardware modifications induced by pipeline squashing at commit time on the OoO engine are limited. In practice, the only major modification compared with a processor without VP is that the predicted values must be written in the physical registers before dispatch. The remaining OoO engine components (scheduler, functional units, and bypass) aren’t impacted.

At first glance, if each destination register must be predicted for each fetch group, then the number of write ports should double. In that case, the overhead on the register file would be high. A register file’s area cost is approximately proportional to $(R + W)^+ (R + 2W)$, with $R$ and $W$ being the number of read and write ports. Assuming $R = 2W$, the area cost without VP would be proportional to $12W^2$, and the one with VP would be proportional to $24W^2$, or double. Energy consumed in the register file would also be increased by around 50 percent (using simple Cacti 5.3 approximation).

For practical implementations, several opportunities exist to limit this overhead. For instance, we can statically limit the number of extra ports needed to write predictions. Another opportunity is to allocate physical registers for consecutive instructions in different register file banks, limiting the number of additional write ports on the individual banks. Yet, even with the proposed optimizations, ports are still required in addition to those already implemented for OoO execution.

Fortunately, the EOIE architecture allows the implementation of VP without any additional ports on the PRF. Thus, commit time validation decouples VP from most of the OoO engine, and EOIE manages to reduce the overhead on the remaining junction between the value predictor and the OoO engine—that is, the PRF.

**Providing high accuracy on the predictor.** To enable validation at commit time, the predictor must be accurate. To our knowledge, all value predictors are amenable to high accuracy at the cost of moderately decreasing coverage. This feat is accomplished by using small confidence counters (for example, 3 bits) whose increments are controlled by probabilities (emulated via a linear feedback shift register). These forward probabilistic counters (FPC) effectively mimic wider counters at much lower hardware cost.

Figure 2 shows the speedups’ [min, max] boxplots brought by VP on SPEC00 and SPEC06 benchmarks using an eight-wide OoO processor. We consider simple 3-bit confidence FPC counters with the following probabilities: $\{1, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}\}$. In both cases, a prediction is used only if its
The two first bars in Figure 2 show performance using an idealistic 0-cycle selective-reissue mechanism. Performance with FPC is slightly lower because some coverage is lost to increase accuracy. The next two bars show performance for validation at commit time and recovery through pipeline squashing. Thanks to the higher accuracy of FPC, no slowdown is observed, and performance is similar to the 0-cycle selective-reissue mechanism. On the contrary, performance can decrease by up to 25 percent with regular counters (the third bar). Using FPC translates to an average speedup of 11.1 percent versus 5.8 percent for classic 3-bit counters.

That is, thanks to this simple scheme, no complex mechanism is required to handle mispredictions. Therefore, aside from the PRF, the execution core can remain oblivious to VP. In other words, VP doesn’t increase complexity in key structures such as the scheduler, the ROB, or the execution units and their bypass networks.

Remaining complexity issue: The PRF

For the processor to use predicted values, they need to flow from the value predictor to the execution core. An intuitive solution is to write them in the PRF at dispatch. Write ports on the PRF must be dedicated to these writes. Moreover, to enforce correctness and train the value predictor, the instructions’ computed results must be read from the PRF to be compared against their corresponding predictions at commit time.

For instance, to write up to eight predictions per cycle, eight additional write ports are needed. Similarly, to validate up to eight instructions per cycle, eight additional read ports are required (assuming predictions are stored in a distinct first-in, first-out structure). Adding that many ports is clearly unrealistic as power and area grow quadratically with the port count in the PRF.

Therefore, while the complexity introduced by validation and recovery doesn’t impact the OoO engine, many additional accesses to the PRF must take place if VP is implemented. To envision VP on actual silicon, we must devise a solution to address the complexity of almost doubling the number of PRF ports. EOLE addresses this issue and reduces the OoO engine complexity in the process.

Introducing EOLE

OoO execution typically implements as-soon-as-possible scheduling. That is, as soon as an instruction has all its operands ready, it’s issued if a functional unit is available, if there remains issue bandwidth, and if all older ready instructions have been issued. Unfortunately, this scheduling scheme isn’t aware of the execution critical path (the longest chain of dependent instructions in the program). As a result, scheduling is nonoptimal, because a noncritical instruction might be scheduled before a critical one simply because the former is older, and performance might be lost.

As-late-as-possible execution (late execution)

As prediction validation is delayed till commit time, a predicted instruction becomes noncritical by construction. Indeed, the instruction’s predicted result can be used by its dependents, so its execution automatically becomes noncritical.

Late execution targets instructions whose result has been predicted. This occurs just before validation time, that is, out of the
execution engine. We limit late execution to single-cycle ALU instructions and high confidence branches\(^\text{10}\) to minimize complexity. That is, predicted loads are executed in the OoO engine, but validated at commit time.

Late execution further reduces pressure on the OoO engine in terms of instructions dispatched to the scheduler. As such, it also removes the need for predicting only critical instructions\(^\text{11}\) because minimizing the number of instructions flowing through the OoO engine requires maximizing the number of predicted instructions. Hence, predictions usually considered useless from a performance standpoint become useful with late execution.

Due to the need to validate predictions (including reading results to train the value predictor) as well as late execution of some instructions, at least one extra pipeline stage after writeback is likely needed in EOLE. In the remainder of this article, we refer to this stage as the late execution and validation (LE/VT) stage. The left portion of Figure 3 illustrates how this stage is plugged to the back end of a regular OoO pipeline.***

**As-soon-as-possible execution (early execution)**

If predictions are available early enough (for example, at rename), some instructions can become ready to execute in the front end. We leverage this by executing those instructions in the front end, in order, using dedicated functional units. We can then consider the results computed early as predictions themselves: they simply must be written to the PRF at dispatch, just like predictions. Nonetheless, because those instructions are executed early, they aren’t dispatched to the scheduler, so, they don’t have to be woken up or selected for issue. The right portion of Figure 3 illustrates early execution and the value predictor interacting with the front end of a regular OoO pipeline.

Among early executable instructions, many are *load-immediate*, meaning that they load a register with an immediate. Thanks to

---

**Figure 3. Early/Out-of-Order/Late Execution (EOLE) pipeline organization.** The center shows a regular OoO pipeline. The left part shows where late execution and validation at commit are plugged into the back end while the right part shows how the predictor and early execution interact with the front end. As previously mentioned, the OoO engine remains untouched except for the physical register file.
VP and early execution, these immediate values can be put in the register file at dispatch without actually dispatching the instructions to the scheduler. This optimization is applicable even in the absence of early execution, as long as additional write ports are available on the PRF (which is assumed for regular VP).

**Potential OoO engine offload**

Figure 4 gives the ratio of retired instructions that can be offloaded from the OoO engine for each benchmark by using an eight-wide, six-issue processor. This ratio is dependent on the application, ranging from less than 10 percent for *equake*, *milc*, *gromacs*, *hmmer*, and *lbm*, to around 40 percent for *swim*, *mgrid*, *applu*, *art*, *perlbench*, *leslie*, *namd*, *GemsFDTD*, and *libquantum*. In most benchmarks, the ratio represents a significant portion of the retired instructions.

**Potential hardware complexity reduction with EOLE**

In EOLE, there is potential to bypass many instructions from the OoO scheduler. This can be leveraged in several ways to reduce complexity in the OoO engine by scaling it down. This in turn leads to savings in the register file, achieving the initial objective of this work.

**Shrinking the OoO engine**

Early and late execution relieve the scheduler from a substantial amount of pressure because they provide some additional issue width (albeit in-order).

**OoO scheduler** That is, thanks to these two dedicated execution stages (early and late), many instructions can simply bypass the OoO engine, reducing the pressure put on the scheduler and the functional units. As a result, optimizations such as OoO issue width reduction become possible. This would greatly impact wakeup because the complexity of each instruction queue (IQ) entry would be lower. Similarly, a narrower issue width mechanically simplifies instruction selection. As such, both steps of the wakeup and selection of the critical loop could be made faster and less power hungry.

Providing a way to reduce issue width with no impact on performance is also crucial because modern schedulers must support complex features such as speculative scheduling and thus selective reissue to recover from scheduling mispredictions.8

**Functional units and the bypass network** As the number of cycles required to read a register from the PRF increases, the bypass network becomes more crucial. The bypass network lets instructions catch their operands as they are produced and thus execute back to back. However, a full bypass network is expensive, especially as the issue width—hence, the number of functional units—increases.

EOLE enables reducing the issue width in the OoO engine. Therefore, EOLE reduces...
the design complexity of a full bypass by reducing the number of ALUs and thus the number of simultaneous writers on the network.

**Limited register file ports on the OoO engine.**
Through reducing the issue width on the OoO engine, EOLE mechanically reduces the number of read and write ports required on the PRF for regular OoO execution.

**Mitigating the hardware cost of early and late execution**
Early and late execution induce a significant increase in the number of ports on the PRF. However, this can be overcome through leveraging the in-order essence of these two execution stages.

**Mitigating early-execution hardware cost.**
Because early-executed instructions are processed in order and are therefore consecutive, we can use a banked PRF and force the allocation of physical registers for the same dispatch group to different register banks.

For instance, considering a four-bank PRF, out of a group of eight consecutive instructions, we could allocate two destination registers in each bank. A dispatch group of eight consecutive instructions would at most write two registers in a single bank after early execution. Thus, early execution would necessitate only two extra write ports on each PRF bank. For a four-issue core, this would add up to the number of write ports required by a baseline six-issue OoO core.

**Narrow late execution and port sharing**
Not all instructions are predicted or late executable (that is, predicted and simple ALU or high confidence branches). Moreover, entire groups of eight instructions are rarely ready to commit.

Thus, we can leverage the register file banking proposed previously to limit the number of read ports on each individual register file bank at LE/VT. To only validate the prediction for eight instructions and train the predictor, and assuming a four-bank PRF, two read ports per bank would be sufficient. However, not all instructions need validation and training (for example, branches and stores). Hence, some read ports might be available for late execution, although extra read ports might be necessary to ensure smooth late execution.

Our experiments showed that limiting the number of LE/VT read ports on each register file bank to four results in a marginal performance loss. For a four-issue core, adding four read ports adds up to a total of 12 read ports per bank (eight for OoO execution and four for LE/VT)—that is, the same amount of read ports as a six-issue core.

We should also emphasize that the logic needed to select the group of instructions for late execution and validation on each cycle doesn’t require complex control and isn’t on the processor’s critical path. This could be implemented either by an extra pipeline cycle or speculatively after dispatch.

**Overall register file complexity.**
On a reduced issue width EOLE pipeline, our proposed register file banking scheme leads to equivalent performance as a nonconstrained register file. However, the four-bank file has only two extra write ports per bank for early execution and prediction and four extra read ports for LE/VT. That is, we have a total of 12 read ports (eight for the OoO engine and four for LE/VT) and six write ports (four for the OoO engine and two for early execution/prediction), just as for the baseline six-issue configuration without VP.

As a result, EOLE virtually nullifies the additional complexity induced on the PRF by VP by diminishing port numbers required by the OoO engine. The only remaining difficulty comes from banking the PRF. Nonetheless, according to the previously mentioned area cost formula, the total area and power consumption of the PRF of a four-issue EOLE core is similar to that of a baseline six-issue core.

The EOLE structure naturally leads to a distributed register file organization with one file servicing reads from the OoO engine and the other servicing reads from the LE/VT stage. As a result, the register file in the OoO engine would be less likely to become a temperature hotspot than in a conventional design.

**Evaluation highlights**
Although EOLE addresses a salient remaining issue of VP through scaling down
the OoO scheduler, we need to ensure that performance isn’t hindered when doing so. In this section, we consider reducing both issue width and scheduler size and find that the former is a more appealing solution because almost no performance is lost.

Experimental framework

We used the x86_64 instruction set architecture to validate EOLE, even though EOLE can be adapted to any general-purpose instruction set architecture. We considered a relatively aggressive 4-GHz, six-issue superscalar baseline with a fetch-to-commit latency of 19 cycles (20 for EOLE due to the additional pipeline stage). Because we focused on the OoO engine complexity, we overdimensioned both the in-order front end and in-order back end to treat up to eight μ-ops per cycle. We modeled a deep front end (15 cycles) coupled to a shallow back end (three cycles) to obtain realistic branch/value misprediction penalties. Table 1 describes the characteristics of our baseline pipeline (all width are expressed in μ-ops because x86_64 instructions are cracked at fetch time by gem5). In particular, the OoO scheduler is dimensioned with a unified, centralized, 60-entry IQ and a 192-entry ROB, on par with Haswell, the latest commercially available Intel microarchitecture. We refer to this baseline as the Base_6_60 configuration.

Unlocking additional issue bandwidth with EOLE

The two first bars in Figure 5 show the performance of Base_6_60 and a six-issue EOLE model (EOLE_6_60) over Base_VP_6_60 on SPEC00 and SPEC06 benchmarks. We’ve

Table 1. Processor parameters.

<table>
<thead>
<tr>
<th>Processor component</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front end</td>
<td>L1 instruction, eight-way, 32-Kbyte, perfect translation look-aside buffer; eight-wide fetch (one taken branch/cycle), decode, rename; TAGE 1 + 12 components, 15,360 entries total, 20 cycles misprediction penalty; two-way 4,000-entry branch target buffer, 32-entry return address stack.</td>
</tr>
<tr>
<td>Execution</td>
<td>192-entry reorder buffer, 60-entry unified instruction queue; 72/48 load and store queues, 256/256 INT/FP four-bank register file; 1,024-entry Store Sets memory dependency predictor; six-issue, four arithmetic logic units (1c), one MulDiv (3c/25c), two FP (3c), two FPMulDiv (5c/10c), two LD/ST, one ST; full bypass; eight-wide writeback, eight-wide validation, eight-wide retire.</td>
</tr>
<tr>
<td>Caches</td>
<td>L1 data eight-way, 32-Kbyte, three-cycle load-to-use, 64 miss status holding registers (MSHRs), two reads and two writes per cycle; unified L2 16-way, 1 Mbyte, 12 cycles, 64 MSHRs, no port constraints, stride prefetcher, degree 8; all caches have 64-byte lines and least recently used replacement.</td>
</tr>
<tr>
<td>Memory</td>
<td>Single-channel DDR3-1600 (11-11-11), two ranks, eight banks/rank, 8, 192-byte row buffer, tREFI 7.8us; across an 8-byte bus; read latency min. ~75 cycles and max. ~185 cycles.</td>
</tr>
</tbody>
</table>
represented results as a [min, max] boxplot. We also give the gmean of the speedups. The main observation is that by adding EOLE on top of VP, we can slightly increase performance—by up to 15 percent in xalancbmk. We observe a small slowdown in three benchmarks due to some branch mispredictions being resolved at late execution.

Reducing OoO aggressiveness with EOLE

The third and fourth bars in Figure 5 show performance when issue width is reduced from six to four in both the EOLE and simple VP model. As expected, performance loss is marginal in EOLE (3.5 percent at most in povray), but performance still increases by 1.5 percent on average. Conversely, the simple VP scheme suffers more noticeably (17.5 percent in namd; 4.4 percent slowdown on average).

The two last bars in Figure 5 show performance when issue width is kept the same, but the number of scheduler entries is decreased by 20 percent, or down to 48. We can observe the same trend as when reducing the issue width, however, in EOLE, performance loss is more significant than before (11 percent in hmmer).

EOLE mitigates performance loss induced by a reduction in both the issue width and scheduler size. However, EOLE is more efficient in the former case, because it can gracefully make up for the lost issue bandwidth, but not for the smaller window size. In addition, reducing the issue width has more benevolent side effects on the OoO engine: fewer PRF ports, less bypass, and simpler wakeup and selection. As a result, we argue that applying EOLE on a VP-enhanced core architecture can keep performance roughly constant while decreasing complexity in the OoO engine through a reduction of the issue width. This paves the way for a truly practical implementation of VP without additional ports on the PRF.

Through EOLE, we propose a VP-enhanced core architecture with limited hardware overhead. Remaining complexity is mostly located in the value predictor itself and can be mitigated in a cost-effective fashion. Such a practical implementation of VP not only increases sequential performance, it also opens the way to many optimizations. For example, there exist several propositions to leverage VP to perform prefetching. Predictability could also be leveraged in the cache for better management. Lastly, results computed on the wrong path could be reused greedily (that is, without identifying them as data- and control-flow independent beforehand) by considering them as value predictions.

Acknowledgments

The European Research Council Advanced Grant DAL no. 267175 partially supported this work.

References


Arthur Perais is a PhD candidate with the Amdahl’s Law is Forever research group at INRIA Rennes. His research interests include core microarchitecture and increasing sequential performance. Perais has an MS in computer science from the Université of Rennes. He is a student member of IEEE. Contact him at arthur.perais@inria.fr.

André Seznec is a research director at INRIA Rennes, where he leads the Amdahl’s Law is Forever research group. His research focuses on microprocessor architecture, including caches, branch predictors, and all forms of speculative execution. Seznec has a PhD in computer science from the Université of Rennes. He is an IEEE Fellow. Contact him at andre.seznec@inria.fr.