EOLE: Toward a Practical Implementation of Value Prediction

Arthur Perais, André Seznec

EOLE: Toward a Practical Implementation of Value Prediction

Arthur Perais André Seznec
INRIA/IRISA
Campus de Beaulieu, 35042 Rennes, France
{arthur.perais,Andre.Seznec}@inria.fr

***Abstract*—We propose a new architecture, EOLE, that leverages Value Prediction to execute a significant number of instructions—5% to 50%—outside the out-of-order engine. This allows the reduction of the issue width, which is a major contributor to both out-of-order engine complexity and register file port requirement. This reduction paves the way for a truly practical implementation of Value Prediction.**

Keywords—Pipeline processors, superscalar, dynamically-scheduled, micro-architecture, speculation.

I. Introduction and Context

Because of legacy code that was not written with parallelism in mind, and because 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*) dependencies. Moreover, scaling the hardware taking advantage of potential ILP (*e.g.* Scheduler, ROB) is non-trivial as those structures are complex—thus impacting CPU cycle time or/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 may now be worth revisiting. Among them is Value Prediction (VP) [3]**Error! Reference source not found.**. VP 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, since they were introducing substantial hardware complexity and additional power consumption in almost every stage of the pipeline. As a result, VP went out of fashion in the early 00's.

Nonetheless, recent work in the field of VP has shown that given an efficient confidence estimation mechanism, prediction validation could be removed from the out-of-order engine and delayed until commit time [4]. As a result, recovering from mispredictions via *selective reissue* can be avoided and a much simpler mechanism—*pipeline squashing*—can be used, while the out-of-order 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 in order to validate them at commit time, potentially rendering the overall number of ports unbearable. Fortunately, VP also implies that many single-cycle 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 since predictions are validated at commit time.

Consequently, a significant number of instructions—5% to 50% in our experiments—can bypass the out-of-order engine, allowing the reduction of the issue width, which is a major contributor to both out-of-order engine complexity and register file port requirement. This reduction paves the way for a truly practical implementation of Value Prediction. Furthermore, since Value Prediction in itself usually increases performance, our resulting {Early | Out-of-Order | Late} Execution architecture, EOLE[1] [5], is often more efficient than a baseline VP-augmented 6-issue superscalar while having a significantly narrower 4-issue out-of-order engine.

II. An enabling Primary Work

A. Designing a Practical Predictor: D-VTAGE

Due to their operation mechanism, *Finite Context Method* (FCM) predictors [8], may not be adapted for implementation. 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, potential for VP is likely to be lost with those predictors. However, recent advances in the field of branch target prediction could be leveraged to design better value predictors.

In particular, since indirect target prediction is a specific case of VP, the *Value TAGE* predictor (VTAGE) was introduced in [4]. This predictor is directly derived from research propositions on branch predictors [10] and more precisely from the indirect 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 FCM [8].

[1] Eole is the French translation of Aeolus, the keeper of the winds in Greek mythology.

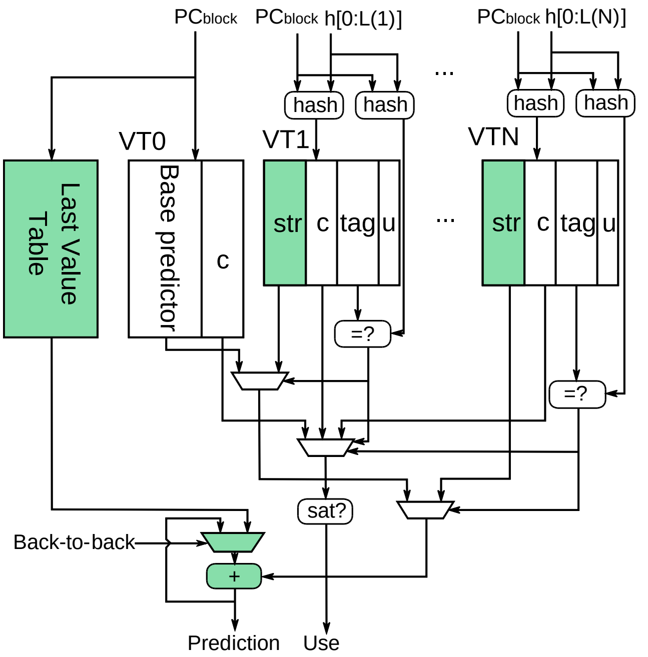

Figure 1: Differential VTAGE predictor layout.

Unfortunately, VTAGE is not adapted for predicting strided patterns, while those patterns are frequent in programs (*e.g.* loop induction variables, regular layout, *etc.*). In particular, each value in the pattern requires its own entry in VTAGE while the whole pattern can be captured by a single entry in a common Stride predictor **Error! Reference source not found.**. As a result, there is a case for hybridizing both predictors. However, a naïve hybrid-having the two components side-by-side would require a great amount of storage since each component would need to 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 storage footprint cannot easily be reduced, or at least not without greatly impacting its performance (*e.g.* by implementing less entries).

As a result, the *Differential* VTAGE predictor, D-VTAGE, was introduced in [6]. It is a tightly coupled hybrid that combines a VTAGE predictor storing smaller strides in its components and a *Last Value Table* (LVT). To predict, a stride is generated using the VTAGE prediction scheme, and added to the last outcome located in the LVT. The size of D-VTAGE can easily be reduced to a reasonable amount (*e.g.* 16/32kB) by using small strides [6].

B. Removing Prediction Validation from the Execution Engine

Two hardware mechanisms are commonly used in processors to recover from misspeculation: *pipeline squashing* and *selective reissue*. They induce very different average misprediction penalties, but are also very different from a hardware complexity standpoint 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 mispredictions. Using *pipeline squashing* is straightforward, but costly as the minimum value misprediction penalty is the same as the minimum branch misprediction penalty. However, , squashing can be avoided if the predicted result has not been used yet, that is, if no dependent instruction has been issued.

Selective reissue is implemented in processors to recover in case where instructions have been executed with incorrect operands, in particular this is used to recover from L1 cache hit/miss mispredictions [2] (*i.e.* load-dependent instructions are issued after predicting a L1 hit, but finally the load results in a L1 miss). When the execution of an instruction with an incorrect operand is detected, the instruction as well as all its dependent chain of instructions are canceled then replayed.

1) Validation at Execute Time vs. Validation at Commit Time

On the one hand, *selective reissue* at execution time limits the misprediction penalty. On the other hand, *pipeline squashing* can be implemented either at execution time or at commit time. *Pipeline squashing* at execution time results in a minimum misprediction penalty similar to the branch misprediction penalty. However, validating predictions at execution time necessitates redesigning the complete out-of-order engine: The predicted values must be propagated through all the out-of-execution engine stages and the predicted results must be validated as soon as they are produced in this out-of-order execution engine. Moreover, the repair mechanism must be able to restore processor state for any predicted instruction. Prediction checking must also be implemented in the commit stage(s) since predictors have to be trained even when predictions were not used due to low confidence.

On the contrary, *pipeline squashing* at commit results in a quite high average misprediction penalty since it can delay prediction validation by a substantial number of cycles. Yet, it is much easier to implement for Value Prediction since it does not induce complex mechanisms in the out-of-order execution engine. It essentially restrains the Value Prediction related hardware to the in-order pipeline front-end (prediction) and the in-order pipeline back-end (validation and training). Moreover, it allows not to checkpoint the rename table since the committed rename map contains all the necessary mappings to restart execution in a correct fashion.

2) A Simple Synthetic Example

Realistic estimations of the average misprediction penalty P_{value} could be 5-7 cycles for *selective reissue* [2], 20-30 cycles for *pipeline squashing* at execution time and 40-50 cycles for *pipeline squashing* at commit.

For the sake of simplicity, we will respectively use 5, 20 and 40 cycles in the small example that follows. We assume an average benefit of 0.3 cycles per correctly predicted value (taking into account the number of unused predictions).

With predictors achieving around 40% coverage and around 95% accuracy as often reported in the literature, 50% of predictions used before execution, the performance benefit when using *selective reissue* would be around 64 cycles per Kinstructions, a loss of around 86 cycles when using *pipeline squashing* at execution time and a loss of around 286 cycles when using *pipeline squashing* at commit time.

3) Balancing Accuracy and Coverage

The total misprediction penalty T_{recov} is roughly proportional to the number of mispredictions. Thus, if one drastically improves the accuracy at the cost of some coverage then, as long as the coverage of the predictor remains quite high, there might be a performance benefit brought by Value Prediction, even though the average value misprediction penalty is very high.

Using the same example as above, but sacrificing 25% of the coverage (now only 30%), and assuming 99.75% accuracy, the performance benefit would be around 88 cycles per Kinstructions cycles when using *selective reissue*, 83 cycles when using *pipeline squashing* at execution time and 76 cycles when using *pipeline squashing* at commit time.

C. Commit Time Validation and Recovery

1) Hardware Implications on the Out-of-Order Engine

In the previous section, we have 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 Value Prediction is that the predicted values must be written in the physical registers before *Dispatch*. The remaining OoO engine components (scheduler, FUs, bypass) are not impacted.

At first glance, if each destination register has to be predicted for each fetch group, one would conclude that the number of write ports should double. In that case the overhead on the register file would be high. The area cost of a register file is approximately proportional to $(R+W)*(R+2W)$, R and W respectively being the number of read ports and the number of write ports [11]. Assuming $R=2W$, the area cost without VP would be proportional to $12W^2$ and the one with VP would be proportional to $24W^2$ *i.e.* the double. Energy consumed in the register file would also be increased by around 50% (using very simple Cacti 5.3 approximation).

For practical implementations, there exists several opportunities to limit this overhead. For instance one 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.

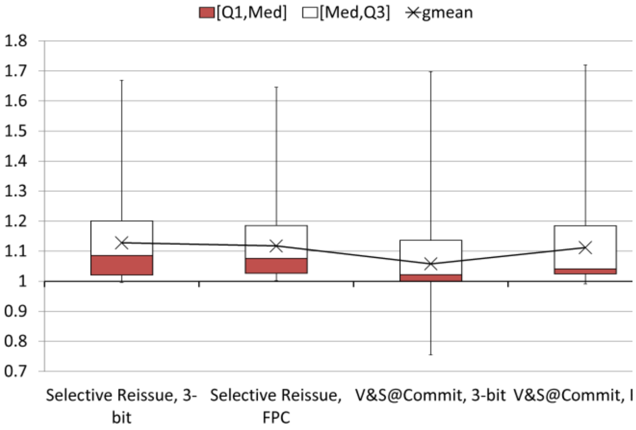

Figure 2: Impact of confidence estimation (3-bit counters vs. 3-bit FPC) on the speedup brought by VP with validation and squash at commit. 8-wide OoO, SPEC00/06 benchmarks.

Yet, even with the proposed optimizations, ports are still required *in addition* to those already implemented for out-of-order execution.

Fortunately, we will show that the EOLE 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 EOLE manages to reduce the overhead on the remaining junction between the value predictor and the OoO engine: the PRF.

2) Providing Very High Accuracy on the Predictor

To enable validation at commit time, the predictor must be very accurate. To our knowledge, all value predictors are amenable to very high accuracy at the cost of moderately decreasing coverage. This feat is accomplished by using small confidence counters (*e.g.* 3-bit) 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 [min,max] boxplots of the speedups brought by VP on SPEC00 and SPEC06 benchmarks using an 8-wide out-of-order processor (further detailed in VI.A). Simple 3-bit confidence counters and 3-bit FPC counters with the following probabilities $\{1,\frac{1}{16},\frac{1}{16},\frac{1}{16},\frac{1}{16},\frac{1}{32},\frac{1}{32}\}$ are considered. In both cases, a prediction is used only if its confidence counter is saturated, and the counter is reset on a misprediction.

The two first bars show performance using an idealistic *0-cycle selective* reissue mechanism. Performance with FPC is slightly lower because some coverage is lost to increase accuracy.

[2] Including tracking and canceling the complete chain of dependent instructions as well as the indirect sources of performance loss encountered such as resource contention due to re-execution, higher misprediction rate (e.g. a value predicted using wrong speculative history) and lower prediction coverage

The two next bars show performance for validation at commit and recovery through *pipeline squashing*. Thanks to the higher accuracy of FPC, no slowdown is observed, and performance is very similar to the *0-cycle selective reissue* mechanism. On the contrary, performance can decrease by up to 25% with regular counters (third bar). Using FPC translates in an average speedup of 11.1% vs. 5.8% for classic 3-bit counters.

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

III. A Major Remaining Complexity Issue: The Physical Register FILE

Predicted values need to flow from the value predictor to the execution core in order to be used. 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 computed results of instructions must be read from the PRF to be compared against their corresponding predictions at commit time.

For instance, to write up to 8 predictions per cycle, 8 additional write ports are needed. Similarly, to validate up to 8 instructions per cycle, 8 additional read ports are required (assuming predictions are stored in a distinct FIFO 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 does not impact the OoO engine, many additional accesses to the PRF must take place if VP is implemented. To envision VP on actual silicon, a solution to address the complexity of almost doubling the number of PRF ports must be devised. EOLE addresses this issue and reduces the OoO engine complexity.

IV. Introducing {Early | OoO | Late} Execution (EOLE)

Out-of-order execution typically implements *As Soon As Possible* (ASAP) scheduling. That is, as soon as an instruction has all its operands ready, it is issued if a functional unit is available, if there remains issue bandwidth, and if all older ready instructions have issued. Unfortunately, this scheduling scheme is not aware of the execution critical path (the longest chain of dependent instructions in the program). As a result, scheduling is non-optimal because a non-critical instruction might be scheduled before a critical one simply because the former is older, and performance might be lost.

A. As Late as Possible Execution (Late Execution)

As prediction validation is delayed till commit time, a predicted instruction becomes non-critical by construction. Indeed, its predicted result can be used by its dependents, so its execution automatically becomes non-critical.

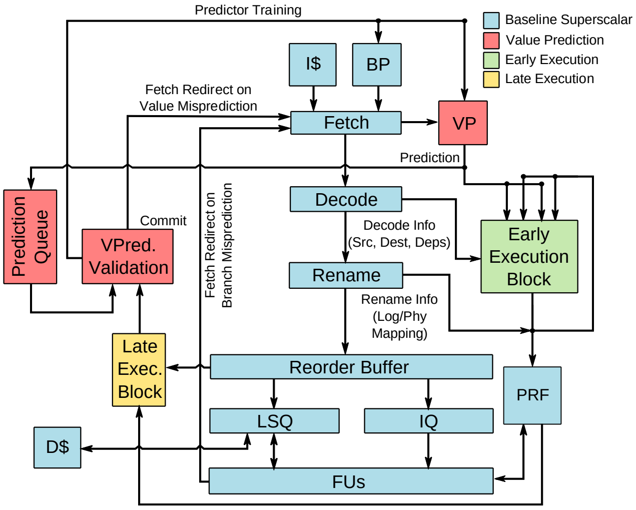

Figure 3: EOLE Pipeline Organization

Late Execution (LE) targets instructions whose result has been predicted. It is done just before validation time, that is, out of the execution engine. We limit Late Execution to single-cycle ALU instructions and high confidence branches [9] to minimize complexity. That is, predicted loads are executed in the OoO engine, but validated at commit.

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 [7] since minimizing the number of instructions flowing through the OoO engine requires maximizing the number of predicted instructions. Hence, predictions usually considered as 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-execute some instructions, at least one extra pipeline stage after *Writeback* is likely to be required in EOLE. In the remainder of this paper, we refer to this stage as the Late Execution/Validation and Training (LE/VT) stage.

B. True ASAP Execution (Early Execution)

If predictions are available early enough (*e.g.* at *Rename*), then 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. The results computed early can then be considered as predictions themselves: They simply have to be written to the PRF at *Dispatch*, just like predictions. Nonetheless, since those instructions are executed early, they are not dispatched to the scheduler, hence, *they do not have to be woken-up* or *selected* for issue.

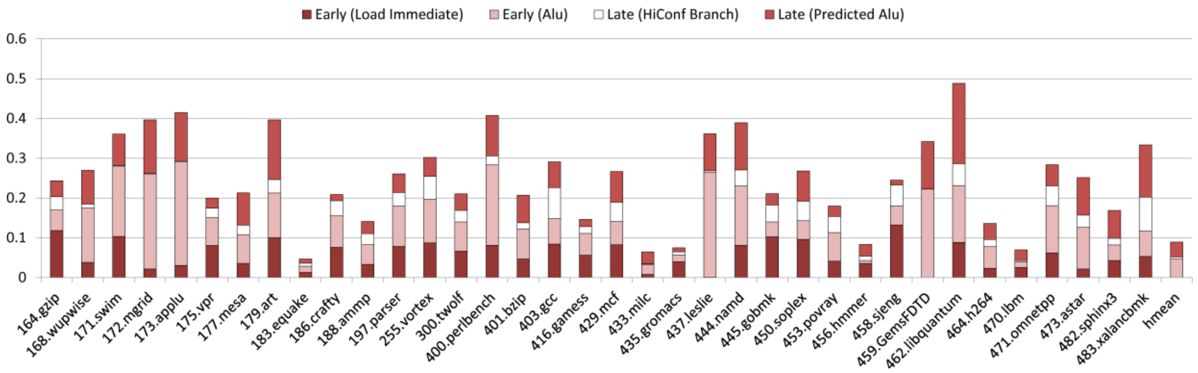

Figure 4: Ratio of early/late executed instructions on a 6-issue EOLE model .

One has to note that among early executable instructions, many are *load immediate* that load a register with an immediate. Thanks to VP and EE, these immediate values can be put in the register file at *Dispatch* without actually dispatching the instructions to the scheduler. Also note that this optimization is applicable even in the absence of EE, as long as additional write ports are available on the PRF (which is assumed for regular VP).

C. *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 8-wide, 6-issue processor described in Section VI.A. This ratio is very dependent on the application, ranging from less than 10% for *equake, milc, gromacs, hmmer* and *lbm* to around 40% for *swim, mgrid, applu, art, perlbench, leslie, namd, GemsFDTD* and *libquantum*. In most benchmarks, it represents a significant portion of the retired instructions.

V. Potential Hardware Complexity Reduction With EOLE

A. *Shrinking the Out-of-Order Engine*

1) *Out-of-Order Scheduler*

Thanks to these two dedicated execution stages (*Early* and *Late*), many instructions can simply bypass the out-of-order engine, reducing the pressure put on the scheduler and the functional units. As a result, optimizations such as the reduction of the out-of-order issue width become possible. This would greatly impact *Wakeup* since the complexity of each IQ entry would be lower. Similarly, a narrower issue width mechanically simplifies *Select*. As such, both steps of the *Wakeup & Select* critical loop could be made faster and/or 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 [2].

2) *Functional Units & Bypass Network*

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

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

3) *A Limited Number of 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.

B. *Mitigating the Hardware Cost of Early/Late Execution*

Early and Late Execution induce a significant increase of the number of ports on the PRF. However, this can be overcome through leveraging the in-order essence of these two execution stages, as illustrated below.

1) *Mitigating Early-Execution Hardware Cost*

Because Early Executed instructions are processed in-order and are therefore consecutive, one 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 4-bank PRF, out of a group of 8 consecutive µ-ops, 2 could be allocated to each bank. A dispatch group of 8 consecutive µ-ops would at most write 2 registers in a single bank after Early Execution. Thus, Early Execution would necessitate only two extra write ports on each PRF bank. For a 4-issue core, this would add-up to the number of write ports required by a baseline 6-issue OoO core.

2) *Narrow Late Execution and Port Sharing*

Not all instructions are predicted or late-executable (*i.e.* predicted and simple ALU or high confidence branches). Moreover, entire groups of 8 µ-ops are rarely ready to commit.

Thus, one can leverage the register file banking proposed above to limit the number of read ports on each individual register file bank at Late Execution/Validation and Training. To only validate the prediction for 8 µ-ops and train the predictor, and assuming a 4-bank PRF, 2 read ports per bank would be sufficient. However, not all instructions need validation/training (*e.g.* branches and stores). Hence, some read ports may be available for LE, although extra read ports might be necessary to ensure smooth LE.

Table 1: Processor parameters.

Front End	L1I 8-way, 32kB, Perfect TLB; 8-wide fetch (1 taken branch/cycle), decode, rename; TAGE 1+12 components [10], 15k-entry total, 20 cycles mis. penalty; 2-way 4K-entry BTB, 32-entry RAS.
Execution	192-entry ROB, 64-entry unified IQ; 72/48 LQ/SQ, 256/256 INT/FP 4-bank register file; 1K-SSID/LFST Store Sets; 6-issue, 4ALU(1c), 1MulDiv(3c/25c), 2FP(3c), 2FPMulDiv(5c/10c), 2Ld/Str, 1Str; Full bypass; 8-wide WB, 8-wide validation, 8-wide retire.
Caches	L1D 8-way, 3 cycles load-to-use, 64 MSHRs, 2 reads and 2 writes/cycle; Unified L2 6-way, 12 cycles, 64 MSHRs, no port constraints, Stride prefetcher, degree 8; All caches have 64B lines and LRU replacement.
Memory	Single channel DDR3-1600 (11-11-11), 2 ranks, 8 banks/rank, 8K row-buffer, tREFI 7.8us; Across an 8B bus; Min Read lat.: ~75 cycles, Max.: ~185 cycles.

Our experiments showed that limiting the number of LE/VT read ports on each register file bank to 4 results in a marginal performance loss. For a 4-issue core, adding 4 read ports adds up to a total of 12 read ports per bank (8 for OoO execution and 4 for LE/VT), that is, the same amount of read ports as a 6-issue core.

It should also be emphasized that the logic needed to select the group of µ-ops to be Late Executed/Validated on each cycle does not require complex control and is not on the critical path of the processor. This could be implemented either by an extra pipeline cycle or speculatively after dispatch.

3) The Overall Complexity of the Register File

On a reduced issue-width EOLE pipeline, the register file banking proposed above leads to equivalent performance as a non-constrained register file. However, the 4-bank file has only 2 extra write ports per bank for Early Execution and prediction and 4 extra read ports for Late Execution/Validation/Training. That is a total of 12 read ports (8 for the OoO engine and 4 for LE/VT) and 6 write ports (4 for the OoO engine and 2 for EE/Prediction), just as the baseline 6-issue configuration without VP.

As a result, if the additional complexity induced on the PRF by VP was commonly believed to be intractable, EOLE allows to virtually nullify this complexity by diminishing the number of ports required by the OoO engine. The only remaining difficulty comes from banking the PRF. Nonetheless, according to the previously mentioned area cost formula [11], the total area and power consumption of the PRF of a 4-issue EOLE core is similar to that of a baseline 6-issue core.

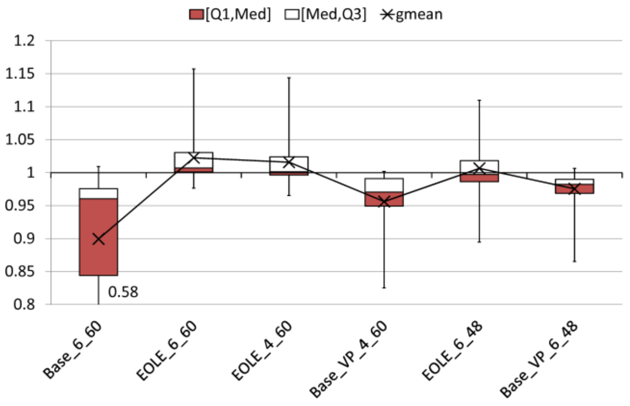

Figure 5: Speedup of EOLE over *Base_VP_6_60* when varying key core parameters.

It should also be mentioned that 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.

VI. Evaluation Highlights

A. Experimental Framework

We use the x86_64 ISA to validate EOLE, even though EOLE can be adapted to any general-purpose ISA.

We consider a relatively aggressive 4GHz, 6-issue superscalar baseline with a fetch-to-commit latency of 19 cycles (20 for EOLE due to the additional pipeline stage). Since we focus on the OoO engine complexity, both in-order front-end and in-order back-end are overdimensioned to treat up to 8 µ-ops per cycle. We model a deep front-end (15 cycles) coupled to a shallow back-end (3 cycles) to obtain realistic branch/value misprediction penalties. Table 1 describes the characteristics of the baseline pipeline we use in more details. In particular, the OoO scheduler is dimensioned with a unified centralized 60-entry IQ and a 192-entry ROB on par with Haswell's, the latest commercially available Intel microarchitecture. We refer to this baseline as the *Base_6_60* configuration (6-issue, 60-entry IQ). Adding a 256kB D-VTAGE predictor to this baseline (with FPC and validation/squash at commit [4]) yields the *Base_VP_6_60* configuration.

B. Unlocking Additionnal Issue Bandwidth with EOLE

The two first bars in Figure 5 shows the performance of *Base_6_*60 and a 6-issue EOLE model (*EOLE_6_60)* over *Base_VP_6_60* on SPEC00 and SPEC06 benchmarks. Results are represented as a [min,max] boxplot. The *gmean* of the speedups is also given. The main observation is that by adding EOLE on top of VP, we are able to slightly increase performance, by up to 15% in *xalancbmk.* A small slowdown is observed in 3 benchmarks due to some branch mispredictions being resolved at Late Execution.

C. Reducing Out-of-Order Aggressiveness with EOLE

1) Reducing the Issue Width

The two next bars show performance when issue-width is reduced from 6 to 4 in both the EOLE and simple VP model. As expected, performance loss is marginal in EOLE (3.5% at most in *povray*) but performance is still increased by 1.5% on average. Conversely, the simple VP scheme suffers more noticeably (17.5% in *namd, 4.4% slowdown* on average).

2) Reducing the Scheduler Size

The two last bars show performance when issue-width is kept the same, but the number of scheduler entries is decreased by 20%, down to 48. The same trend can be observed as when reducing the issue-width, however, in EOLE, performance loss is more significant than before (11% in *hmmer).*

3) Summary

EOLE allows to mitigate the performance loss induced by a reduction in both the issue-width and scheduler size. However, it is more efficient in the former case because EOLE 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 out-of-order engine: less PRF ports, less bypass, simpler *Wakeup & Select*. As a result, we argue that applying EOLE on a VP-enhanced core architecture can keep performance roughly constant while decreasing complexity in the out-of-order engine through a reduction of the issue width. This paves the way for a truly practical implementation of Value Prediction without additional ports on the PRF.

VII. Conclusion

Through EOLE, we propose a Value Prediction enhanced core architecture with limited hardware overhead. First, the out-of-order execution engine is made agnostic on VP. Second the OoO issue width can be reduced thanks to the additional issue bandwidth brought by Early and Late Execution. VP complexity is reported to the in-order frond-end and the in-order retire stage. The only part of the out-of-order engine touched by VP is the PRF, but we provide solutions to avoid actually increasing its complexity. As a result, VP with EOLE can be viewed as a mean to simplify designs rather than the opposite.

In summary, augmenting a current generation processor core with EOLE would allow a performance increase on-par with those observed between two Intel/AMD microarchitecture generations, without actually touching the execution core.

Acknowledgment

This work was partially supported by the European Research Council Advanced Grant DAL No. 267175.

References

[1] F. Gabbay and A. Mendelson, “Using Value Prediction to Increase the Power of Speculative Execution Hardware”, ACM Transactions on Computer Systems, vol. 16, 1998, pp. 234-270.

[2] I. Kim and M. Lipasti, “Undestanding Scheduling Replay Schemes”, Proceedings of the Annual International Symposium on High-Performance Computer Architecture, 2004, pp. 198-209.

[3] M. Lipasti and J. P. Shen, “Exceeding the Dataflow Limit via Value Prediction”, Proceedings of the Annual International Symposium on Microarchitecture, 1996, pp. 226-237.

[4] A. Perais and A. Seznec, “Practical Data Value Speculation for Future High-End Processors”, Proceedings of the Annual International Symposium on High-Performance Computer Architecture, 2014, pp. 428-439.

[5] A. Perais and A. Seznec, “EOLE: Paving the Way of an Effective Implementation of Value Prediction”, Proceedings of the Annual International Symposium on Computer Architecture, 2014, pp. 481-492.

[6] A. Perais and A. Seznec, “BeBoP: A Cost Effective Predictor Infrastructure for Superscalar Value Prediction”, Proceedings of the Annual International Symposium on High-Performance Computer Architecture, 2015.

[7] B. Rychlik, J. Faistl, B. Krug, A. Kurland, J. Sung, M. Velev and J. P. Shen, “Efficient and Accurate Value Prediction Using Dynamic Classification”, Canergie Mellon University Technical Report, CMuart-1998-01, 1998.

[8] Y. Sazeides and J. Smith, “The Predictability of Data Values”, Proceedings of the Annual International Symposium on Microparchitecture, 2011, pp. 248-258.

[9] A. Seznec, “Storage Free Confidence Estimation for the TAGE Branch Predictor”, Proceedings of the Annual International Symposium on High-Performance Computer Architecture, 2011, pp. 443-454.

[10] A. Seznec and P. Michaud, “A Case for (Partially) Tagged Geometric History Length Banch Prediction”, Journal of Instruction Level Parallelism, vol. 8, 2006.

[11] V. Zyuban, P. Kogge, “The Energy Complexity of Register Files”, Proceedings of the International Symposium on Low Power Electronics and Design, 1998, pp. 305-310.

Arthur Perais is PhD candidate with the Amdahl’s Law is Forever (ALF) research group at INRIA Rennes. His research focuses on core microarchitecture and how to increase sequential performance. He has a master’s degree in Computer Science from the Université of Rennes. He is a student member of IEEE.

André Seznec is a Directeur de Recherche at INRIA Rennes and the head of the Amdahl’s Law is Forever (ALF) research group. He got a PhD in computer science from Université of Rennes in 1987. His research interests include microprocessor architecture including caches, branch predictors and all forms of speculative execution. André Seznec is an IEEE fellow.