On the Interactions Between Value Prediction and Compiler Optimizations in the Context of EOLE

Fernando Endo, Arthur Perais, André Seznec

▶ **To cite this version:**

Fernando Endo, Arthur Perais, André Seznec. On the Interactions Between Value Prediction and Compiler Optimizations in the Context of EOLE. ACM Transactions on Architecture and Code Optimization, 2017. hal-01519869

HAL Id: hal-01519869
https://inria.hal.science/hal-01519869

Submitted on 9 May 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Interactions Between Value Prediction and Compiler Optimizations in the Context of EOLE

FERNANDO A. ENDO, ARTHUR PERAIS, and ANDRÉ SEZNEC, IRISA/Inria

Increasing instruction-level parallelism is regaining attractiveness within the microprocessor industry.

The EOLE microarchitecture and D-VTAGE value predictor were recently introduced to solve practical issues of value prediction (VP). In particular, they remove the most significant difficulties that forbade an effective VP hardware.

In this study, we present a detailed evaluation of the potential of VP in the context of EOLE/D-VTAGE and different compiler options. Our study shows that if no single general rule always applies—more optimization might sometimes leads to more performance—unoptimized codes often gets a large benefit from the prediction of redundant loads.

CCS Concepts: •**Computer systems organization** → **Pipeline computing;** •**Hardware** → *Emerging architectures;*

Additional Key Words and Phrases: {Early | Out-of-order | Late} Execution microarchitecture, high performance, general-purpose processor, single-thread performance.

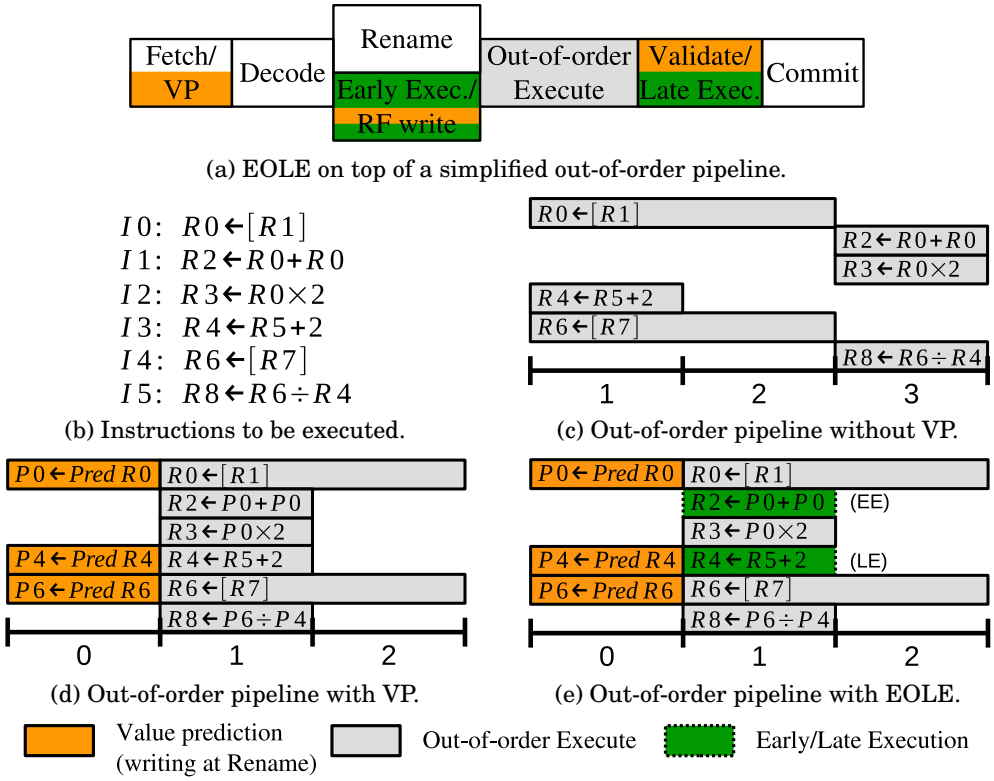
ACM Reference Format:
Fernando A. Endo, Arthur Perais, and André Seznec, 2016. On the Interactions Between Value Prediction and Compiler Optimizations. *ACM Trans. Architec. Code Optim.* 0, 0, Article 0 (0), 23 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Value prediction (VP) gained momentum in the late 90s as a way to increase the instruction-level parallelism (ILP) by exceeding the dataflow limit [Lipasti et al. 1996; Gabbay 1996]. However, at that time several major implementation obstacles precluded an effective hardware implementation. In the following decade, the microprocessor industry and the research community lost interest in VP, because of the advent of multicores and the quest of thread-level parallelism (TLP). However, increasing the core count alone does not provide more performance in many applications, since they do not exhibit sufficient TLP. In addition, with the advent of the Dark Silicon [Merritt 2009; Esmaeilzadeh et al. 2011], the idea of having a variety of general purpose cores inside a processor is spreading in the industry. The aim is to have small energy-efficient cores for workload throughput and big power-hungry ones to accelerate single-threaded or the serial portions of applications. In this context, a VP-enabled core could serve as a general-purpose accelerator core, for applications that benefit from it. Hence, increasing single thread performance becomes attractive again for designers and researchers.

Value prediction increases ILP by speculating the result of instructions that follows repetitive patterns. To illustrate how this is achieved, Figure 1 presents an example of a code being executed with and without value prediction. Consider the instruction flow in Figure 1b, that is executed in an out-of-order pipeline. For simplicity, load

Authors' address: Fernando A. Endo, Arthur Perais and André Seznec, IRISA/Inria, Campus de Beaulieu, 263 Avenue du Général Leclerc, 35042 Rennes Cedex, France; emails: {fernando.endo, arthur.perais, andre.seznec}@inria.fr.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 0 ACM. 1544-3566/0/-ART0 $15.00
DOI: 0000001.0000001



(a) EOLE on top of a simplified out-of-order pipeline.

(b) Instructions to be executed.

(c) Out-of-order pipeline without VP.

(d) Out-of-order pipeline with VP.

(e) Out-of-order pipeline with EOLE.

Fig. 1: Execution flow in a superscalar out-of-order pipeline. For simplicity, load instructions have a latency of 2 cycles, all other instructions take 1 cycle and there is no issue restriction. In the abscissa, the numbers indicate the cycle of execution from the point of view of the execute stage, except the Late Execution, which happens even later.

instructions have a latency of 2 cycles, all other instructions take 1 cycle and there is no issue restriction. Assuming no hazards, Figure 1c shows the execution flow without value prediction. In order to respect the data dependencies, instructions $I1$, $I2$ and $I5$ have to wait for their sources, resulting in 3 cycles of execution. Again in the best case, Figure 1d shows the execution with value prediction. In this case, the registers $R0$, $R4$ and $R6$ have been predicted at rename time (the predicted values are denoted by $P0$, $P4$ and $P6$) and the out-of-order engine can issue 6 instructions in cycle 1, twice as much compared with the case without value prediction, yielding a 2-cycle execution time.

Recently Perais and Seznec revisited hardware value prediction. They addressed several issues that were considered as major obstacles to effective hardware implementations in a processor [Perais and Seznec 2014b; Perais and Seznec 2014a; Perais and Seznec 2015b; Perais and Seznec 2015a; Perais and Seznec 2016]. In summary, they proposed the {Early | Out-of-order | Late} Execution (EOLE) microarchitecture, which changes the design of the out-of-order core, opening new possibilities in microarchitecture and compiler research.

The main VP issue was related to selective replay, which had been previously assumed as the required recovery mechanism together with prediction validation in the execute stage. Contrary to what conventional wisdom suggested, repair on VP can be implemented at the commit stage without sacrificing the potential VP performance,

if only predictions with very high confidence are issued [Perais and Seznec 2014b]. This allows to remove the hardware complexity of VP repair from the out-of-order stages. The second main issue was the extra ports requirement in the register files to write the predictions. This was solved with the {Early | Out-of-order | Late} Execution (EOLE) microarchitecture [Perais and Seznec 2014a; Perais and Seznec 2015b; Perais and Seznec 2015a; Perais and Seznec 2016]. EOLE allows to reduce the hardware complexity of out-of-order cores, including the register file issue, through an "early" and "late" in-order execution units placed before and after the out-of-order execute stage, as Figure 1a shows. VP and EOLE are not only complementary—but de facto required together to make VP implementable.

Considering again the instruction flow in Figure 1b, with EOLE, the best case execution flow is depicted in Figure 1e: Because $I1$ depends only on a predicted value in the preceding cycle, it can be "early executed" (EE) at rename, eluding the out-of-order engine. Early and late execution only target ALU instructions for timing and area concerns, therefore load and complex ALU instructions are normally executed in the out-of-order execute stage. $I3$ is an ALU instruction and its result has been predicted. Therefore, it can be "late executed" (LE) at commit to validate the prediction, relieving the out-of-order engine even more. With EOLE, the execution time is also 2 cycles, but 2 instructions are early/late executed, then 4 instructions are issued in the out-of-order engine, instead of 6 with VP alone.

With VP, measuring figures of merit alone, i.e., accuracy and coverage, may not bring new insights, because the performance of value prediction strongly depends on program behavior. The achievable VP performance depends on two main factors: The criticality of the predicted instructions, i.e., if they are in the critical path of the program; and their *usefulness*. A prediction is *useful* if the predicted result is read before the actual result is computed.

Therefore, characterizing value prediction is paramount to improve performance and to propose new mechanisms. Previous work that characterized value prediction date back to the 90s [Gabbay 1996; Lipasti et al. 1996; Lipasti and Shen 1996; Sazeides and Smith 1997], and none of them ever analyzed the potential of VP in the presence of different compiler optimizations.

The contribution of this paper is an extensive characterization of VP with EOLE.

The main results and findings are:

- Programs compiled with lower optimization levels benefit more from value prediction, and the extra performance brought by EOLE almost doubles with unoptimized code (gcc `-O0`), especially because of redundant and unscheduled loads (Section 5);
  - Loads are relatively more useful with `-O0` in opposition to higher levels (Finding #1). This happens because GCC forcibly enables simple strength reduction and register allocation passes starting from `-O1` (Finding #4);
  - We define a new metric called *usefulness* and we show that it correlates with speedup 14 % better than coverage does (Finding #2);
  - High coverage levels comes from several types of runtime constants, which we detail in Finding #3;
- We analyze the interaction between individual compiler optimizations and value prediction and find some counter-intuitive results (Section 6);
  - Disabling individual flags does not mechanically increase or decrease value prediction performance (Finding #5). VP indeed needs some optimizations more than ordinary processors (Finding #8);
  - Disabling some standard optimization flags in some cases may actually benefit EOLE while not changing or impairing performance on a standard processor, because those flags have been developed for non-EOLE architectures (Finding #6);

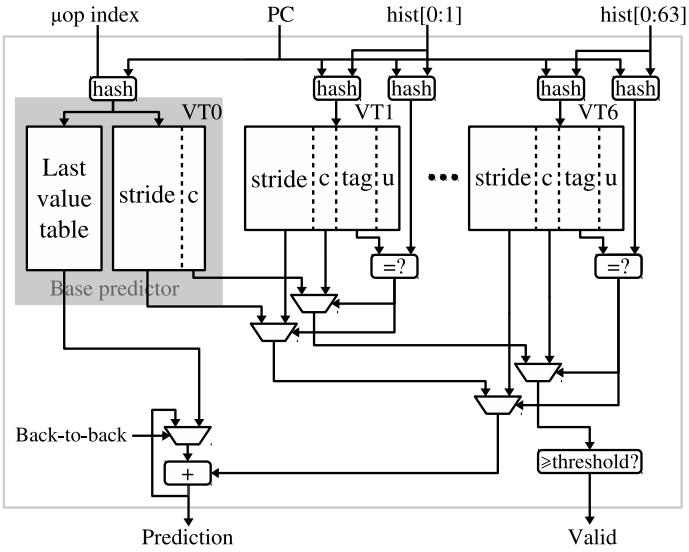


Fig. 2: $1 + 6$-component D-VTAGE predictor.

— Instruction scheduling passes should be modified for value prediction. Instructions consuming a likely value predicted result should better stay close to its producer, so that the prediction will be useful. This is in contrast to current compiler scheduling approaches (Finding #7);
— Although load instructions correspond to less than 30 % of value prediction hits, they contribute with around 60 % of the potential speedup provided by EOLE (Section 7), in accordance with similar results in previous work [Calder et al. 1999].

2. D-VTAGE AND EOLE BACKGROUND

This section briefly overviews the D-VTAGE value predictor and the EOLE microarchitecture. It also points out how compilers can exploit the interactions between software and the value prediction hardware.

Software-hardware interaction #1. Instruction scheduling policies may change in a VP-enabled processor. From Figure 1d and 1e, extra ILP is obtained by VP in an idealized machine. However, if instructions *I1* and *I2* were statically scheduled far away from *I0*, or if functional units were not available for them, it is likely that other instructions or pipeline stalls would have happened during cycles 1 and 2, and VP would have had no performance gain at all.

2.1. D-VTAGE

The Differential Value TAgged GEometric history length predictor (D-VTAGE) [Perais and Seznec 2015a] is a value predictor composed of two main modules: a base (stride) predictor and N tagged tables (components), forming an $1+N$ structure. Figure 2 shows D-VTAGE in an $1 + 6$-component configuration. Unlike branch predictors, D-VTAGE is not only placed in the fetch stage, but also writes the prediction in the register files at rename time and validates it in a pre-commit stage (Figure 1a).

The base predictor is indexed with the PC of the instruction hashed with its internal micro-op number. It is subdivided in a last value table (LVT), and a table holding the strides (str) with confidence estimation (c). Each one of the N tagged tables has entries with a stride (str), a confidence estimator (c), a tag and a useful bit (u). They are indexed

with varying bit lengths of the global branch history outcome ($hist$, sequence of bits, 0: branch mispred., 1: correct pred.) hashed with the PC. These variable lengths form a geometric series and permit to capture the dynamic instruction path followed by programs.

The prediction tables can provide as many as necessary predictions per cycle (i.e., fetch width). If one static instruction has to be predicted twice in two consecutive cycles, the second prediction uses the first prediction as last value, through the *Back-to-back* signal. If the same static instruction appears twice in the same fetched block, the second appearance is predicted through a 3-input adder (not shown in Figure 2).

Software-hardware interaction #2. From the compiler point of view, it is interesting to note that aliasing in D-VTAGE can happen in two ways: Two static instructions fall in the same entry in the LVT or VT0 (similarly to address aliasing in direct-mapped caches); or two dynamic instructions fall in the same entry in one of the tagged tables (VTN except VT0), because they have similar hashes (mix of instruction address and global branch outcome). In general, only the first form can be addressed by the compiler. In the second form, only pinpoint if-conversions may avoid some aliasing.

2.2. EOLE

{Early | Out-of-order | Late} Execution (EOLE) is a pipeline design that enables a practical implementation of VP [Perais and Seznec 2014a]. It leverages the fact that if only very high confidence predictions are used, validation can be delayed to the commit stage and performed in-order [Perais and Seznec 2014b]. Considering an out-of-order pipeline with VP/validation at commit, EOLE adds on top of it an early execution engine (EE) in parallel to the register rename stage, and a late execution engine (LE) together with the validation in a pre-commit stage (Figure 1a). Both engines are composed of single-stage ALUs with full bypassing.

The EE executes single-cycle ALU instructions that are ready at rename, using immediate, predicted values from the same rename group or early-executed/predicted results from the previous cycle. Briefly, early-executed instructions do not need to be validated, because the chains of speculative operands originate from previously (in program order) value-predicted instructions that are going to be validated anyway.

The LE executes not only single-cycle ALU instructions that had their results predicted, but also very high confidence branches. Given that the LE further reduces the instruction flow in the out-of-order engine, useless value predictions (i.e., which do not shorten the critical path) might allow to enhance EOLE performance in a narrower out-of-order core.

Software-hardware interaction #3. Because the EE unit reduces the pressure on processor resources by executing simple ALU instructions at rename time, a compiler could replace PC relative loads of constants by a few immediate moves (which are early executed), if reducing the data cache pressure tends to be more beneficial than increasing the instruction count.

3. VALUE PREDICTION IN AARCH64

This section describes the implementation details of VP that we assumed for AArch64. AArch64 is a 64-bit ISA [ARM 2015], therefore in this work we consider a 64-bit value predictor. Even though we simulate a very aggressive microarchitecture, the AArch64 ISA has been chosen because the x86_64 implementation in gem5 has known issues as unnecessary register dependencies [Nowatzki et al. 2015], lack of boosting structures for a highly micro-coded ISA (e.g., move elimination, micro-op fusion and stack engine [Perais and Seznec 2014a]), and hence not having representative performance of state-of-the-art processors.

We assume that instructions are decoded into micro-ops early in the fetch stage. Fetched micro-ops are classified into three distinct categories:

— Non value-predictable;
— INT or FP free move immediate;
— INT or FP value-predictable.

Non value-predictable micro-ops such as branches do not access the value prediction tables and are executed as usual. Free move immediate regroup instructions that are ready at rename time, which can write an immediate value through register file (RF) ports that are reserved for VP. Value predictable micro-ops always check for predictions at fetch time. These two latter categories are further detailed in the following subsections.

3.1. INT/FP Register Bank Writes

The INT and FP classification distinguishes micro-ops that write into the INT or FP/SIMD register files, and it is not necessarily related to the ISA definition of INT or FP instruction. For example, there are FP and SIMD instructions that only write to the INT RF, such as move instructions and data type conversions. We consider all instructions that write at most 64 bits in the register files, including scalar FP instructions and SIMD instructions that only write the lower 64 bits (and always set the upper 64 bits to zero). In this study, SIMD instructions that write 128 bits are not considered value predictable since they would require to predict 128 bits for a single instruction.

3.2. Free Move Immediate

A free move immediate corresponds to a move micro-op that writes an immediate, zero-extended and unshifted value, to a register. Such micro-ops can be "freely" executed at rename, because extra write ports in the RFs are already implemented to write predicted values.

3.3. Value-Predictable Micro-Ops

A micro-op is value-predictable if the following conditions are met:

— It writes to one or more general-purpose (GP) register and/or to the condition code (CC) register;
— The destination register has 64 or fewer bits, if it has more than 64, the upper bits must always be zero-extended;
— It is not classified as free move immediate.

All value-predictable micro-ops access the value predictor to check for matching entries. If a match is found and the confidence is high enough, the prediction is written into the corresponding physical register at rename time. A value-predictable micro-op can either be classified as:

(1) Writing into one or more GP registers;
(2) Writing into one or more GP registers and into the CC register;
(3) Writing into the CC register only.

AArch64 defines four condition flags: negative (N), zero (Z), carry (C) and overflow (V).

A micro-op from group 1 writes into a GP register following the rules from Section 3.1. If it writes to more than one GP register, then only one of them can be predicted.

In group 2, besides writing the prediction into a GP register, in our simulations the CC flags are derived from the predicted value and written to a physical CC register. The N and Z bits are easily derived from the predicted value. We considered that the

most frequent case is an operation between two positive values that do not overflow 64 bits. Briefly, the C bit is assumed to be 0, except when the opcode is a SUB[1]. The V flag is always assumed to be 0.

In group 3, the prediction of the CC flags is stored in one 64-bit entry of the predictor. At rename time, the prediction is used if the confidence is high enough.

4. EXPERIMENTAL SETUP

This section presents the experimental environment. We describe the simulation environment, the benchmarks used in the experiments and the compilers/flags employed.

4.1. The baseline configuration

In our experiments we employed the gem5 simulator[2] [Binkert et al. 2011].

Our baseline differs from the original gem5 version in the following main aspects:

— It features a TAGE branch predictor [Seznec and Michaud 2006];
— The BTB is set-associative;
— The fetch, decode and rename (frontend) stages are decoupled with real instruction queues;
— Store sets are only trained on the correct path;
— Store-to-load forwarding has the latency of L1 hits;
— The stride prefetcher has a parameter to set a prefetching distance[3];
— The DDR4 model has unlimited accesses per row[4] and no front/back-end (controller and PHY) latencies (best case DDR4 latencies were tuned to match commercial memories – around 40 ns).

The baseline parameters are shown in Table I. We will refer to this configuration as Base. Even though we simulate the AArch64 ISA, it closely models the Haswell microarchitecture [Intel 2015], with a 128 MB eDRAM L4 cache (Crystal Well family).

We considered a large L4 cache as recent trends regarding memory technology suggest that this will become widespread in the near future, particularly in processors featuring on-die accelerators (e.g., GPU). In any case, reducing the average latency of memory accesses can only diminish VP potential, thus this setup does not advantage VP in any fashion.

The core and uncore clocks were set to 4 GHz, which is approximately the peak frequency of Haswell in turbo mode.

4.2. EOLE configuration

We assume an EOLE configuration with the same parameters as the baseline. In addition, it features a D-VTAGE predictor [Perais and Seznec 2015a] plus the early and late execution engines.

D-VTAGE predicts the results of micro-ops as described in Section 3. Figure 2 shows the D-VTAGE $1 + 6$-component configuration used in the experiments. The base stride predictor has a last value table and a table containing the strides (VT0) with 8 k entries each. The 6 tagged components have 1 k entries each and tags with $12 + rank$ bits, $rank$ going from 1 to 6. The branch history lengths used to index the tagged components form a geometric series going from 2 to 64. All confidence estimators are 3-bit probabilistic counters mimicking 7 bits [Riley and Zilles 2006], incremented on correct prediction and reset on mispredictions.

[1]The subtraction of two positive numbers produces a carry.
[2]http://repo.gem5.org/gem5/rev/20bbfe5b2b86
[3]The next address to prefetch equals the current address plus the stride multiplied by the distance.
[4]Micron MT40A512M8 datasheet.

Table I: Main parameters of the simulated cores.

Front-end	8-wide and 15-deep; L1I: 8-way, 32 kB, 64 MSHR, 1 cycle; 128-entry fully-assoc. I-TLB; TAGE 1+12 components 15 k entries total (≈ 32 kB) [Seznec and Michaud 2006], 19 cycles of min. branch mis. penalty; 2-way 8 k-entry BTB; 32-entry RAS.
Execution	8-wide back-end; 192-entry ROB, 60-entry unified IQ, 72/48-entry LQ/SQ, 235/88 INT/FP registers; 2 k/1 k SSIT/LFST store sets [Chrysos and Emer 1998]; 10 exec. ports: 2 ALU (1c), 1 ALU/Mult (1c/3c), 1 ALU/Div (1c/12c[a]), 1 SimdALU-Mult-FloatMult (3c/3c/4c), 1 SimdALU-FloatAdd-FloatMult (3c/4c/4c), 1 SimdALU (3c), 2 MemRead-Write, 1 MemWrite.
Caches	L1D: 8-way, 32 kB, 64 MSHR, 32 Write buffers (WB), 4 cycles, 256/256 B peak load/store, 64-entry fully-assoc. D-TLB; L2: 512 kB, 11 cycles; L3: 8 MB, 34 cycles; L4: 128 MB, 120 cycles. L2, L3 and L4: 16-way and 64/64 MSHR/WB.
Prefetchers	L1: stride, degree=1, distance=16 [Michaud 2016], L2, L3 and L4: stream, degree=1.
Memory	Dual channel DDR4-2400 17-17-17, 2 ranks, 16 banks/rank, 8 kB row buffer, t_{REFI}=7.8 μs, min. read latency: 36 ns, average: 75 ns (SPEC CPU 2006).

[a] Unpipelined functional unit.

Table II: Benchmark regions and IPC of the baseline gem5 in function of optimization levels.

Benchmark	Function	WC	RC	O3	O2	O1	O0
400.perlbench	S_regmatch	161	818	1.6	1.6	1.5	1.7
401.bzip2	mainGtU	324 822	1 498 540	1.8	1.8	1.7	2.0
403.gcc	single_set_2	116	808	1.4	1.4	1.4	1.3
410.bwaves	mat_times_vec	1	1	1.4	1.3	1.6	3.5
416.gamess	DIRFCK	3264	20 893	1.9	1.9	2.0	2.3
429.mcf	primal_bea_mpp	3400	25 350	0.5	0.5	0.5	0.8
433.milc	uncompress_anti_hermitian	28 430	236 893	1.9	2.4	2.6	2.9
434.zeusmp	lorentz[a]	22	22	1.4	1.4	2.1	2.7
435.gromacs	inl1130	11	95	1.2	1.3	1.3	1.6
436.cactusADM	Bench_StaggeredLeapfrog2[b]	4	44	1.7	1.8	1.5	1.8
437.leslie3d	FLUXJ[c]	1693	1693	1.5	1.9	2.7	3.5
444.namd	calc_self_energy	4	34	1.7	1.7	1.5	2.0
445.gobmk	do_play_move	8279	150 233	1.9	1.9	1.9	1.7
447.dealII	compute_fill	56	80	1.9	2.0	2.2	1.5
450.soplex	vSolveUrightNoNZ	39	314	1.6	1.7	1.7	2.0
453.povray	All_Plane_Intersections	88 392	765 377	1.7	1.7	1.8	1.6
454.calculix	e_c3d	22	22	1.7	4.1	4.2	3.8
456.hmmer	P7Viterbi	5	70	4.3	4.4	3.6	2.5
458.sjeng	std_eval	7339	58 159	1.9	1.8	1.9	2.1
459.GemsFDTD	updateE_homo[d]	16 200	16 200	0.9	1.7	1.8	3.1
462.libquantum	quantum_toffoli	2	9	2.2	2.2	2.3	2.7
464.h264ref	FastPelY_14	885 992	2 838 742	3.2	3.3	3.2	2.7
465.tonto	make_esfs	242	967	1.7	1.7	1.7	1.8
470.lbm	LBM_performStreamCollide	1	1	1.0	1.0	0.9	1.7
471.omnetpp	cMessageHeap::shiftup	33 215	31 027	0.7	0.7	0.7	1.1
473.astar	wayobj::makebound2	1069	7559	0.7	0.7	0.7	1.0
481.wrf	advect_scalar	1	2	1.8	2.3	2.6	2.8
482.sphinx3	mgau_eval	11 204	112 799	1.8	1.8	1.7	2.0
483.xalancbmk	ValueStore::contains	395	2590	1.4	1.3	1.4	1.2
Average				1.7	1.8	1.9	2.1

Acronyms: Warm calls (WC), Run calls (RC).
[a] Beginning of 4th `do/continue`.
[b] Beginning of 5th `do/end do`.
[c] Beginning of 2nd `do/end do`.
[d] Beginning of 8th `do/end do`.

The early, late and validation units are 8-wide and not constrained by RF ports, as previous work have shown negligible impact on performance [Perais and Seznec 2014a]. The EE is implemented as a single-cycle stage in parallel to rename, while the late execution and validation is implemented as a single-cycle pre-commit stage.

4.3. Enforcing comparable simulations

Existing methodologies (for instance, SimPoint [Perelman et al. 2003]) to select representative sections of benchmarks are dedicated to simulate a single binary on a single input data set. These methodologies are not adapted to our study where we have to simulate several distinct binary versions of the same application. Since the simulation of the complete application would take months of CPU time, which is out of reach, we introduce below a methodology guaranteeing that the same slice of work is simulated in each benchmark.

For each benchmark we carefully chose one function that both represents a significative part of the execution time when compiled with gcc `-O3` (on a NVIDIA Jetson TX1), and being called near the beginning of the benchmark. The minimum, average and maximum percentage of the time that the chosen functions represent are 5.1 (444.namd), 33 and 99 % (470.lbm), respectively. Then we compiled the benchmark with the `-O3` option first, and selected a slice of the application that covers N calls to the function (approximately 50 million contiguous instructions) for simulator warming, and M calls (representing approximately 500 million contiguous instructions) for effective simulation. The whole program and kernel code executed during the N + M calls are simulated, including several nested function calls. The number of dynamic instructions simulated represents a very small fraction of the total number in each benchmark, but this setup is analogous to published work in the literature using similar simulators [Calder et al. 1999; Burtscher and Zorn 1999; Perais and Seznec 2014a]. For each of the compiler options, we simulated the exact same slice represented by the M+N calls. The calls to the selected functions are only used as reference points in the code, to guarantee that the same benchmark slices are simulated with varying compiler optimizations.

Adapting this methodology to select a set of representative slices of the applications is out of the scope of this study.

4.4. Benchmarks

We simulated the SPEC CPU 2006 benchmark suite. GCC 4.9.3 (Linaro GCC 4.9-2015.01-3) was used, except 416.gamess that could only be compiled with GCC 4.7.3 (linaro-1.13.1-4.7-2013.01-20130125). The baseline flags were: `-static -march=armv8-a -fno-strict-aliasing`[5]. The simulations run in gem5 with the Full-System mode under a Linux 3.16.0-rc6. This mode is needed to simulate complex system calls from the benchmarks. In a small experiment, simulating 500 million instructions from 10 random benchmarks and 10 runs each one called after varying initial delays, we estimated the OS noise as being around 0.08 % on average (min.: 0.03, max: 0.25 %).

Table II presents the simulated benchmark regions and IPC of Base. Figure 3 shows the increase in dynamic instructions and runtime when optimization level decreases. Because with decreasing optimization level the dynamic instructions count often increases proportionally more than runtime, IPC follows the inverse trend as Table II shows.

[5] 464.h264ref and 482.sphinx3 required the option `-fsigned-char`.

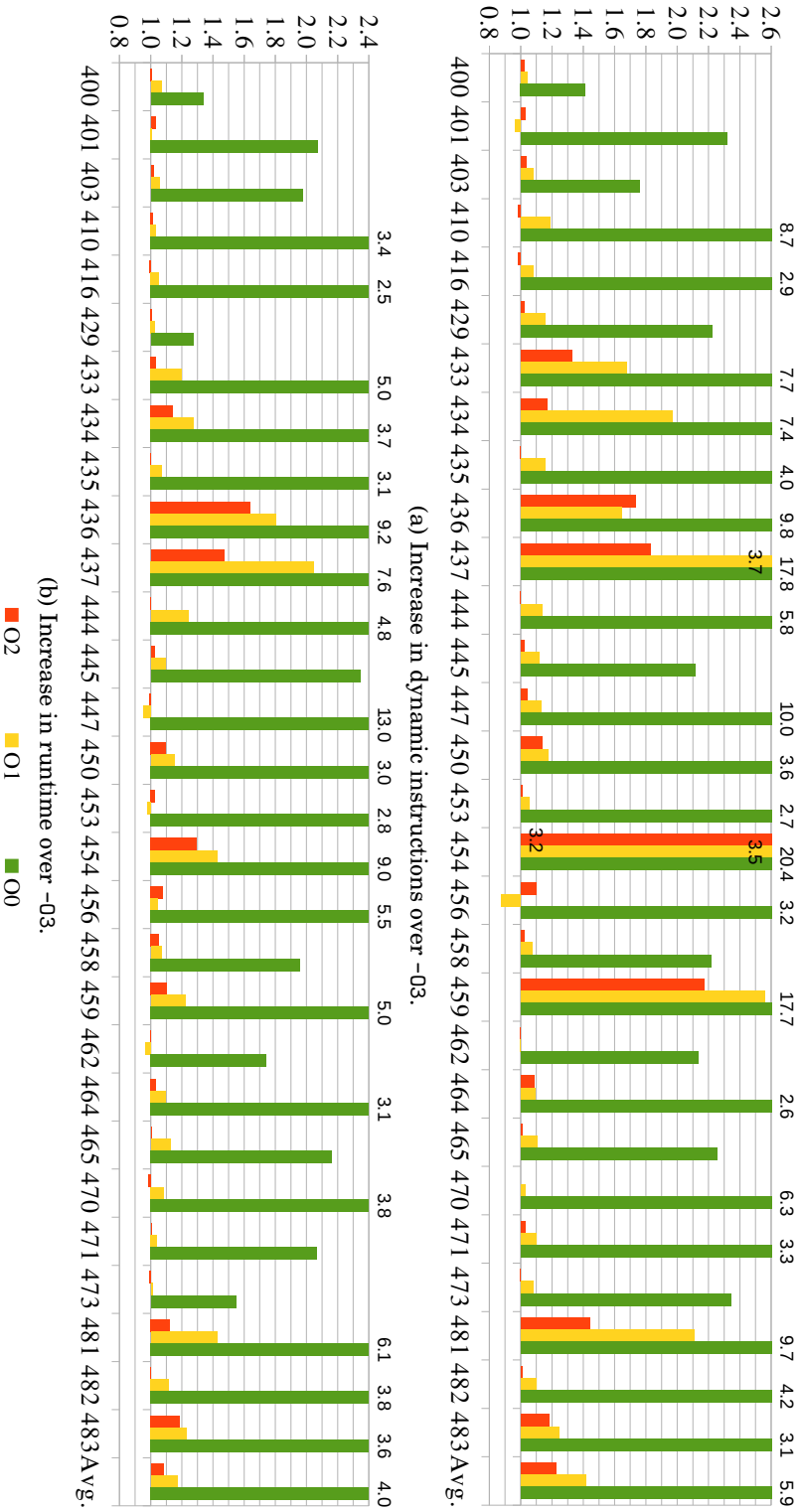


Fig. 3: Dynamic instruction and runtime increase with −02, −01 and −00 in Base, normalized to −03.

5. VARYING OPTIMIZATION LEVELS

In this section, we analyze the performance of EOLE with varying compiler optimization levels. Not only classic figures as predictability and coverage are used to analyze the results, but we also introduce a new metric called *usefulness*.

Figure 5 shows the speedup of EOLE over Base. Globally, EOLE rarely slows down the execution and can speed up by almost 2× in the best case. The speedup trends with `-O3`, `-O2` and `-O1` are similar in most benchmarks. On average, with these three optimization levels EOLE provides a speedup around 6.5 %. In contrast, with `-O0`, EOLE can speed up more in most benchmarks, increasing the geometric mean to 11.5 %. We note that measured speedups do not follow any global trend when the optimization level changes.

The three figures of merit are described in the following.

Predictability. It represents the percentage of fetched micro-ops that suits the conditions for VP presented in Section 3.3.

Coverage. It is defined as the ratio of correctly committed predictions over the number of committed value-predictable micro-ops. The coverage closely represents the percentage of predictions issued per predictable micro-ops being committed, because the accuracy is around 99.9 %.

Usefulness. It is defined as the proportion of committed value-predictable micro-ops that were classified as useful. By **useful** predictions, we denote a prediction that is used before the actual result has been computed.

5.1. Predictability, coverage and usefulness

Figure 4 shows the predictability, coverage and usefulness of D-VTAGE per micro-op type (with examples of instructions in the legend).

From Figure 4a, the overall predictability does not change considerably with varying optimization level. Three exceptions are 436.cactusADM, 437.leslie3d and 459.GemsFDTD, which were vectorized with `-O3`. Some benchmarks have a lot of predictable FloatMisc micro-ops. Particularly, in GCC 4.9.3 the local register allocator (LRA) is used by default in AArch64, which spills variables from INT to FP/SIMD registers. This explains the significant proportion of FloatMisc micro-ops, which encompass such MOVs.

In Figure 4b, the coverage varies significantly between benchmarks. Surprisingly, the benchmarks 410.bwaves, 434.zeusmp, 436.cactusADM and 459.GemsFDTD have almost 100 % of coverage in most cases, including non-negligible proportions of FP multiply (and accumulate), which should not be predicted so often. This happens because checkpoints were taken near the benchmarks beginning with sometimes many instruction producing zero or very predictable results in the first iterations.[6]

Finding #1. The aggregated usefulness follows approximately the same trend as the aggregated coverage (Figures 4b and 4c). Now, if we compare the proportion of micro-op types, we can note that on average these proportions stay roughly the same between `-O1` and `-O3`, but with `-O0` the percentage of IntAlu drops from 43 % in the coverage to 21 % in the usefulness, while the percentage of memory accesses rises from 38 % to 54 %, respectively. Given that loads have higher latencies, this observation globally explain the higher speedups with `-O0` compared with the other optimization levels. In summary, loads become relatively more useful with `-O0`, in opposition to `-O1` and higher.

[6] If the benchmarks were fast-forwarded to 3 % of the retired instructions in all experiments, it would unaffordably take one month in a 40-core cluster. gem5 already supports hardware acceleration for fast-forward, but our ARM platform does not.

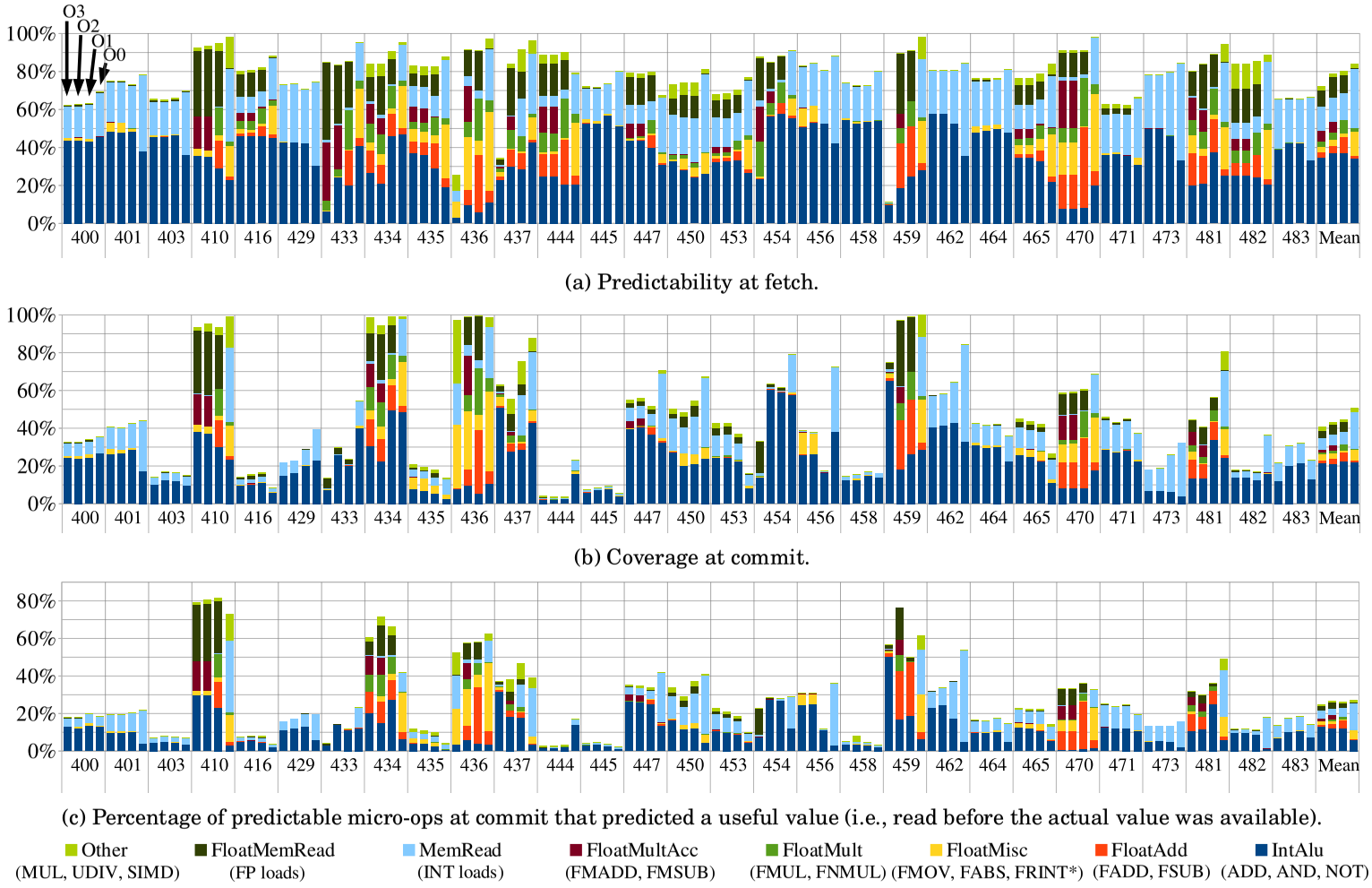


Fig. 4: Percentage of micro-ops suitable for value prediction (predictability) at fetch time, coverage of D-VTAGE at commit time, and usefulness at commit, with varying compiler optimization levels.

ACM Transactions on Architecture and Code Optimization, Vol. 0, No. 0, Article 0, Publication date: 0.

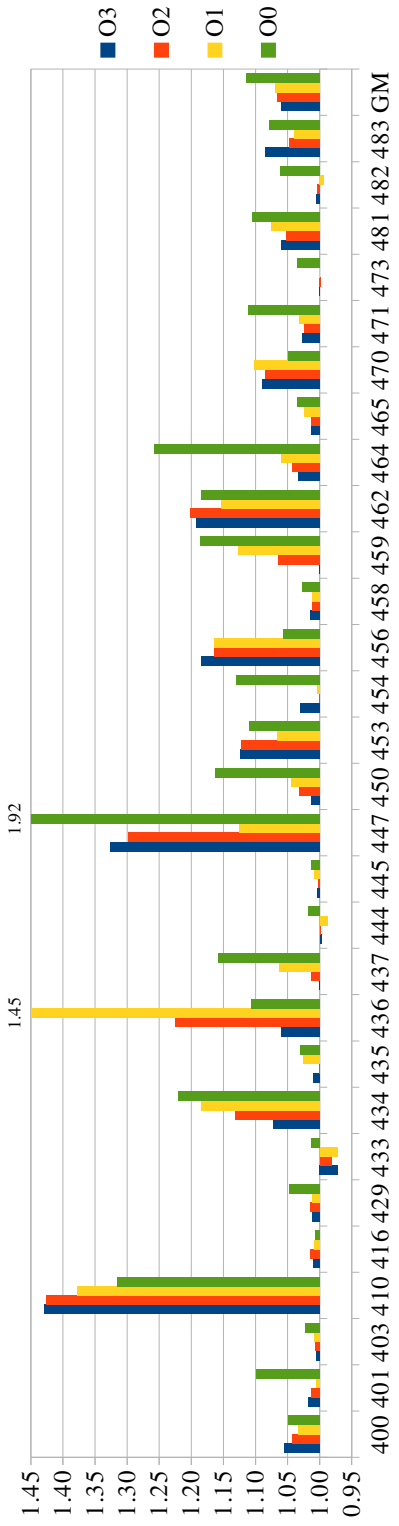


Fig. 5: Speedup of EOLE over the baseline with different optimization levels.

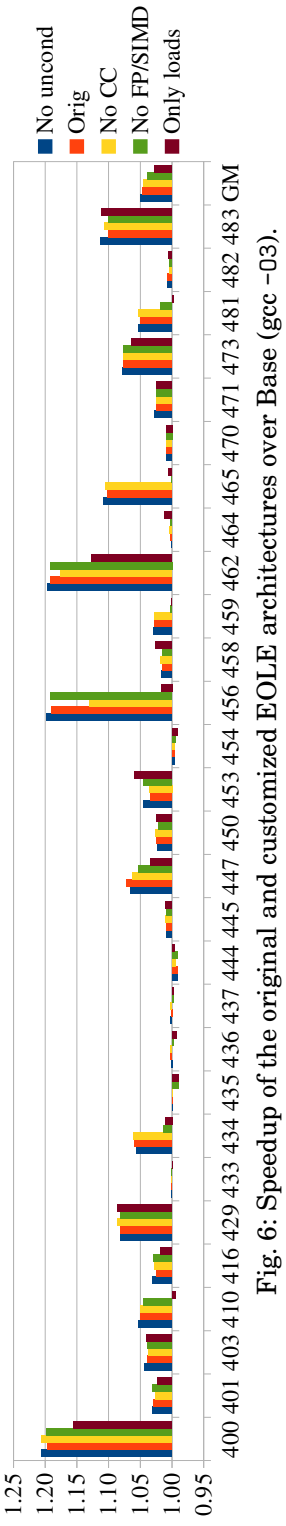


Fig. 6: Speedup of the original and customized EOLE architectures over Base (gcc -O3).

Table III: Pearson correlation coefficient between coverage/usefulness and speedup, over 29 benchmarks.

Optimization flag	Coverage	Usefulness	Difference
-O3	0.49	0.60	23 %
-O2	0.64	0.62	−3.3 %
-O1	0.73	0.81	11 %
-O0	0.41	0.51	23 %
Average			14 %

5.2. Correlation between speedup and coverage/usefulness

Intuitively, usefulness should better correlate with speedup than coverage. Table III consolidates this intuition by showing the correlation between coverage/usefulness and speedup, computed with the data from Figures 5 and 4.

Finding #2. Except for -O2 where the correlation using the usefulness is lowered by 3.3 %, on average the usefulness correlates with the speedup 14 % better than the coverage.

5.3. Benchmarks with high coverage

Four benchmarks, 410.bwaves, 434.zeusmp, 436.cactusADM and 459.GemsFDTD, have several similarities: They are coded in Fortran with several long or deep loops; several runtime constants inside their functions were predicted (input arguments and derived variables), resulting in high coverages and sometimes speedups.

Finding #3. High coverage comes from runtime constants found in various code patterns: In 410.bwaves, one matrix is multiplied several times with different vectors. In 434.zeusmp, 16 temporary arrays are repeatedly computed in loops and subsequently used as constants. In 436.cactusADM, the chosen function has more than one hundred arguments, used as constants in several loops. Finally, 459.GemsFDTD has a function with three constant arrays as input that are repeatedly read to compute the components of three output arrays. All these runtime constants provide high coverages independently from compiler optimizations, often more than 80 % of the dynamic instructions were predicted.

410.bwaves has higher speedups than the other three benchmarks because of its higher usefulnesses.

In 436.cactusADM and 459.GemsFDTD, with -O3 the inner loops were vectorized with full-width SIMD instructions and hence not predicted. The low predictability, even if the coverage is high, resulted in lower speedups compared to the other optimization levels.

Interestingly, in 436.cactusADM -O1 is 7.5 % faster than -O2 with EOLE, while with Base there is a slowdown of 9 %. Disabling instruction scheduling produced this speedup, and its impact on VP is explained in Section 6.3.

5.4. 433.milc: Small, but non-negligible slowdown

433.milc is the only that encounters some slowdowns with EOLE. With -O3, 65 % of predictions came from two simple kernel functions that multiply/add matrices. Even if some matrix elements were predicted, the total number of predicted loads represents less than 5 % of the committed ones from -O1 to -O3, justifying their very low coverages/usefulnesses and slowdowns. With -O0, about 12 % of committed loads were predicted. Among them, 96 % are popped values from the stack, which were optimized with higher optimization levels. This increase in coverage was only enough to compensate the EOLE overhead (misprediction recovery and extra cycle for validation).

Finding #4. Studying 433.milc, we observed that a significant part of the coverage and speedup with `-O0` come from popped value from the stack. Further analysis indicates that:

(1) Constants are computed and pushed at the beginning of the functions and then repeatedly popped (48 % of predicted loads);
(2) Input parameters are repeatedly popped from the stack instead of being held in registers (42 % of predicted loads);
(3) Simple predictable computations as incrementing indexes are popped just after being pushed (3.8 % of predicted loads);
(4) Only very few other data (matrix elements) are predicted (2.3 % of predicted loads).

Item 1 can be avoided with strength reduction (i.e., replace constant loads by immediate operands), while 2 and 3 can be effectively reduced by liveness analysis and register allocation passes. These optimizations are indeed—always enabled—starting from `-O1` in GCC and cannot be enabled/disabled by command line options. Indeed, the predicted loads of items 1, 2 and 3 disappear with `-O1`.

5.5. 447.dealII: Highest speedup variation between `-O1` and `-O0`

This benchmark obtained not only the highest speedup with `-O0`, the second highest with `-O2` and `-O3`, but also the highest speedup difference between `-O1` and `-O0`.

The most predicted instructions from `-O1` to `-O3` came from a highly predicted loop that multiply matrix elements: The loop index, two address offsets, one load and the `FMADD` or `FADD` represent around half of the predicted instructions. However, because this sequence cannot be further optimized compared to the rest of the code, it became progressively more critical as the optimization level increased from `-O1` to `-O3`, justifying the respective increasing speedups.

Conversely, `-O0` produced the worst code among all benchmarks, with an almost nine fold increase in dynamic instruction count compared to `-O1` (Figure 3a). This lack of optimization further increased the criticality of the matrix multiplication loop. For example, the address of matrix elements is computed with a base plus offset starting at `-O1`. However, without any optimization, it is computed through two nested function calls performing redundant push-pop sequences. As a consequence, the speedup of 447.dealII with `-O0` peaks at 1.9 with EOLE.

5.6. VP can harvest ILP even when compiler optimizations cannot

Only looking at speedups can sometimes be misleading. In 453.povray, Base was 5 % faster with `-O1` than `-O2` (Fig. 3b), while EOLE had the same performance with both levels. A similar behavior happened with 462.libquantum.

Given that these benchmarks had an equal or greater number of instructions with `-O1` than `-O2` (Fig. 3a), therefore `-O1` surely produced more ILP than `-O2`. It is likely that this extra ILP was somehow redundant to EOLE, which could already extract ILP from VP, explaining the different behavior with Base.

6. INFLUENCE OF COMPILER OPTIMIZATIONS ON VALUE PREDICTION

This section investigates the influence of compiler optimizations on VP. First, we state two hypotheses that may explain the higher speedups with lower optimization levels. Then, we detail our methodology and discuss the results.

Hypotheses on why EOLE performs better with lower optimization levels. The first hypothesis ($H1$) assumes that under-optimized code may be easier to predict, due to redundancy (e.g., pushes and pops that could be avoided by better allocating registers), or simple computations that could have been removed at compile time (e.g., always recompute the addition of two constant values). The second hypothesis ($H2$) considers

the fact that current compiler optimizations were established based on the behavior of existing processors, therefore an EOLE-like processor could benefit from not yet established optimizations, and some standard optimization passes could eventually hinder VP.

6.1. Methodology

To clarify the influence of compiler optimizations on VP, we analyzed the speedup of EOLE with varying compiler flags. Given that testing all benchmarks with all possible flags results in an intractable number of possibilities, we studied three classes of benchmarks:

— Group O3: Regroups 5 benchmarks for which absolute speedup ratios between `-O3` and `-O2` were greater then 3 %;
— Group O2: The same between `-O2` and `-O1` with 8 benchmarks;
— Group O1: The same between `-O1` and `-O0` with 17 benchmarks.

This speedup criteria was chosen to limit the number of cases, and because low speedup variations with neighbor optimization levels means that the added/removed flags unlikely affect the VP performance.

Benchmarks from group O3 were compiled with `-O3` and with one of the togglable flags added by `-O3` over `-O2` disabled at a time (8 flags, except `-fstrict-aliasing`). An analogous approach was employed in the benchmarks of groups O2 and O1, respectively with 35 and 32 flags.

The combination of benchmark groups and flags resulted in a space with 864 different programs.

Metrics. Our study employed three metrics: The speedup of EOLE with a given flag disabled over the case with the flag enabled, S_{EOLE} (Eq. 1); the same for Base, S_{Base} (Eq. 2); and the *speedup factor* (SF), defined in Eq. 3.

A positive speedup factor tells us that a given flag, when disabled, is more beneficial to EOLE than to Base. If negative, the opposite is true. In other words, the *speedup factor* is also equivalent, in percentage, to the speedup of EOLE vs Base when the flag is off divided by the speedup of EOLE vs Base when flag is on.

$$S_{EOLE} = \frac{num\_cycles_{flag\_on}(EOLE)}{num\_cycles_{flag\_off}(EOLE)} \tag{1}$$

$$S_{Base} = \frac{num\_cycles_{flag\_on}(Base)}{num\_cycles_{flag\_off}(Base)} \tag{2}$$

$$SF = \frac{S_{EOLE}}{S_{Base}} - 1 \tag{3}$$

Hypotheses in equations. In this experiment, we are seeking for patterns in the above metrics that can give us evidences of hypotheses $H1$ and/or $H2$. For example, whenever $SF > 0$ for a given standard flag, it means that a less optimized code benefits more or impairs less EOLE than Base, because the increased redundancy helps VP, attesting $H1$. A special subset of $H1$ happens when a standard flag also produces $S_{EOLE} > 1$ and $S_{Base} \leq 1$, meaning that disabling the flag only results in more speedup for EOLE, eventually degrading the performance of Base. In consequence, the standard flag does not fit the EOLE architecture, and $H2$ is plausible in this case. To filter negligible performance variations, we considered a margin of ± 1.5 %, with positive speedups being values greater than 1.015 (negative: lower than 0.985) and therefore meaningful SFs having absolute values greater than 3 %. Eq. 4 and 5 summarizes the

patterns of $H1$, $H2$. Eq. 6 and 7 present the previous equations, but with Base and EOLE interchanged; we call them alternative hypotheses $\overline{H1}$ and $\overline{H2}$.

$$SF > 3\,\% \implies H1 \tag{4}$$

$$H1 \wedge S_{EOLE} > 1.015 \wedge S_{Base} \leq 1.015 \implies H2 \tag{5}$$

$$SF < -3\,\% \implies \overline{H1} \tag{6}$$

$$\overline{H1} \wedge S_{EOLE} \leq 1.015 \wedge S_{Base} > 1.015 \implies \overline{H2} \tag{7}$$

6.2. Overall results

Compiler optimizations not always have a performance impact on a program phase. In order to find the space of benchmark simulations in which a performance impact was observed in our experiment, Table V shows the number of simulations per optimization level in which the observed speedups were beyond the ± 1.5 % margin (3 % for the SF). By doing so, the initial simulation space with 864 points was reduced to only 182 points. In other words, this reduced space, called Space of remarkable simulations (SRS), contains all the simulated benchmark configurations in which at least one of our three metrics, S_{Base}, S_{EOLE} and SF, had a value greater than the negligible performance threshold.

Table IV lists the flags that scored at least in one hypothesis. Surprisingly, $H1$ and its alternative hypothesis had almost the same score, representing about 18 % of remarkable simulations each.

Finding #5. The fact that hypothesis H1 and its alternative hypothesis had virtually the same score means that disabling individual optimizations actually does not mechanically increase or decrease VP performance.

On the other hand, hypothesis $H2$ overwhelmingly beats its alternative hypothesis, scoring in 13 % against less than 2 % of remarkable simulations.

Finding #6. The fact that $H2$ had a greater score than its alternative hypothesis implies that some standard compiler optimizations in some cases **diminish VP potential**—not impairing performance in current processors.

We do not claim that such standard flags must always be disabled when VP is present. Rather, understanding the impact of such flags on a value predictor can bring new software and hardware insights to further improve performance.

As Table IV shows, only 12 flags scored at least 2 benchmarks in the same hypothesis. In the following, because of space limitations, we discuss some of these flags, whose interaction with value prediction can be explained with simple reasoning.

6.3. Flags Supporting $H1$ and $H2$

Three flags strongly corroborated $H1$ and $H2$.

Instruction scheduling, `-fschedule-insns` (Group O2), actually scored three times for $H1/H2$, but also twice for $\overline{H1}$. In the best case, disabling it produced $SF = 27$ % in 436.cactusADM and in the worst case -3.5 % in 447.dealII. What happened in 436.cactusADM is that the usefulness of `FMADD` instructions increased by 68 % with `-fschedule-insns` disabled (with almost no increase in coverage).

Finding #7. Disabling instruction scheduling can sometimes result in more VP performance. This fact reveals a very interesting effect: If instructions were very well scheduled, eventually removing all stall cycles from RAW dependencies, indeed VP would have no benefit at all, because the actual results would always get computed before the predicted value became useful. However, perfect scheduling is not possible

Table IV: Score all hypotheses per benchmark group.

Group	Flag	$H1$	$\overline{H1}$	$H2$	$\overline{H2}$
O1	-ftree-dominator-opts	0	1	0	0
	-ftree-sra	1	0	1	0
	-ftree-copyrename	1	0	1	0
	-ftree-dce	1	0	1	0
	-ftree-dse	1	0	1	0
	-fdce	1	2	0	0
	-ftree-pta	2	1	1	0
	-ftree-ccp	1	0	1	0
	-fguess-branch-probability	2	1	2	0
	-ftree-slsr	1	0	1	0
	-fcprop-registers	1	0	1	0
	-ftree-fre	1	2	1	0
	-ftree-ch	0	1	0	0
	-ftree-ter	1	1	1	1
	-fif-conversion	2	1	1	0
	-fsplit-wide-types	0	1	0	0
	-ftree-forwprop	1	0	1	0
	-ftree-phiprop	1	0	1	0
O2	-foptimize-sibling-calls	0	2	0	0
	-fcrossjumping	0	1	0	0
	-fdevirtualize	0	1	0	0
	-ftree-tail-merge	0	1	0	0
	-freorder-functions	1	0	1	0
	-fsched-spec	0	1	0	0
	-fstrict-overflow	3	1	2	1
	-frerun-cse-after-loop	1	1	1	0
	-ftree-switch-conversion	0	1	0	0
	-ftree-pre	0	2	0	0
	-fschedule-insns2	0	1	0	0
	-fexpensive-optimizations	0	3	0	1
	-fgcse-lm	1	1	1	0
	-fsched-interblock	0	2	0	0
	-fschedule-insns	3	2	3	0
	-fgcse	1	1	1	0
O3	-fpredictive-commoning	1	1	0	0
	-ftree-loop-vectorize	2	0	0	0
	-ftree-partial-pre	0	1	0	0
	-finline-functions	0	1	0	0
	Total	31	35	24	3
	Percentage of the SRS	17 %	19 %	13 %	1.6 %

Table V: Metric statistics per benchmark group, over 75 flags and 864 different configurations.

	Group O1	Group O2	Group O3	Sum
$C1 = S_{Base} > 1.015 \vee S_{Base} \leq 0.985$	41	17	14	72
$C2 = S_{EOLE} > 1.015 \vee S_{EOLE} \leq 0.985$	106	51	15	172
$C3 = SF > 3\,\% \vee SF < -3\,\%$	29	31	6	66
$SRS = C1 \vee C2 \vee C3$	113	54	15	182

Abbreviation: Space of remarkable simulations (SRS).

in current processors, because of their complex pipelines and memory hierarchies. A helpful compiler design would take into account this observation, that is, it may be better to schedule a consumer instruction close to its producer, if the producer is known or likely to be value predicted and if there may be a free functional unit to execute the consumer. By doing this, future resource contention might be avoided.

With `-fguess-branch-probability` (Group O1), the proportion of committed unconditional branches increased in 453.povray (2.5 %) and 483.xalancbmk (24 %), because compilers may need to insert this kind of instruction to change the preferred direction of if-else blocks. Unconditional branches can increase micro-op aliasing in the tagged tables, since they cannot inform about changes in instruction values. Disabling the flag produced SFs of 12 and 4.3 %, respectively. A simple mechanism to avoid this kind of aliasing would simply remove unconditional branches from the branch history used to index the predictor. By doing so, the speedups over `-O1` increased by only 0.4 and 0.9 %, instead of the expected 11 and 1.8 % increases, respectively, suggesting that an indirect effect of disabling `-fguess-branch-probability` leads to ever higher SFs.

6.4. Other Flags Supporting $H1$

The if-conversion (`-fif-conversion`, Group O1) and vectorization (`-ftree-loop-vectorize`, Group O3) flags supported $H1$.

The if-conversion optimization tries to replace if-else structures by conditionally executed code. Intuitively, instructions dependent on and coming after a non-dead if-else can have distinct entries in the tagged tables of the predictor. With the if-conversion optimization, such a chain of instructions should have conflicts in the predictor. In 453.povray and 456.hmmer, the coverage increased by 14 and 9 %, resulting in SFs of 6.5 and 3.8 %, respectively, with `-fif-conversion` disabled.

Vectorization has a more obvious impact on EOLE: Full-width vector instructions cannot be predicted. Then, disabling vectorization automatically increases the number of predictable and predicted instructions, and also the speedup of EOLE over Base. In 459.GemsFDTD, disabling the vectorization slows down Base by 8 %, but EOLE by only 1.5 %.

6.5. Flags Supporting $\overline{H1}$

Finding #8. Several standard flags corroborated $\overline{H1}$ (Table IV), which means that **value prediction rather needs these standard flags more than ordinary processors**.

In general these flags remove not worth predicting or not predictable instructions. The result is an approximation of VP producers to consumers, increasing the VP effectiveness. In the following, we illustrate this finding.

The flag `-fdce` (dead code elimination) removes instructions that could never be useful if predicted. For example, in the inner loop of 436.cactusADM, 65 MOVs and FMOVs out of 1511 instructions (4.3 %) were eliminated by this flag. Such instructions write to registers that are subsequently overwritten without being read before. Then, disabling this flag resulted in a negative impact to EOLE with a SF of -5.1 %.

The option `-ftree-fre` (full redundancy elimination) for instance removes redundant computations out of loops. Predicting such recomputations, especially simple ALU operations, may not be worth, because their operands are likely always ready. In 436.cactusADM, this optimization reduced the percentage of ALU instructions in the innermost loop of `Bench_StaggeredLeapFrog2` from 9 to 4 %. Disabling this flag produced in a SF of -5.7 %.

The flag `-fexpensive-optimizations` for instance combines `FADD` and `FMUL` into a `FMADD`, among several other optimizations. In the hot function of 410.bwaves, this optimization compressed the innermost loop of `mat_times_vec` by 22 % (from 39 to 32

instructions). This increased the usefulness of loads and ALU micro-ops of 8.3 %. With this flag disabled, the SF was -6.3 %.

6.6. Flags with complex pipeline interaction

Some flags interacted in a non straightforward way with EOLE. For example, `-foptimize-sibling-calls` tries to replace a function call at the end of another function by a jump, avoiding one RET (function return). In 462.libquantum, more than 99 % of the issued predictions fell in three functions, which have the exact same instructions (with different addresses though) no matter the optimization was on or off. Similarly, in 447.dealII, more than 98 % of predicted instructions came from functions with the same exact instructions. Disabling this flag would increase the number of RET instructions, which in turn may increase the miss rate in the return address stack (RAS). Indeed, the RAS miss rate of Base and EOLE had an increase of 5 % in 462.libquantum, but this over only a thousand RETs, representing an insignificant number of events over half a billion instructions. These results cannot explain the fact that EOLE slowed down almost 5 %, while Base did not at all. In 447.dealII, `-foptimize-sibling-calls` did not change the RAS miss rate, and again EOLE was slowed down by 4 %. This flag must have had a very subtle interaction with the pipeline.

Just like `-foptimize-sibling-calls`, the flags `-fstrict-overflow` and `-fsched-interblock` also showed a similar behavior.

The option `-ftree-fre` also had a very subtle interaction with VP in 456.hmmer. With this optimization, its longest innermost loop had a considerable structural change: 6 extra IFs got if-converted, thanks to `-ftree-fre`, which allowed more if-conversion patterns. With `-ftree-fre` disabled, more branches were executed in the inner loop, and the branch predictor miss rate had an increase of 6.9 %. Even if EOLE could predict CMP instructions, the branches that use them had already been predicted by the branch predictor. By enabling `-ftree-fre`, IFs got converted to conditional execution, which could be predicted by EOLE and also benefit from predicted CMP instructions. As a result, VP became more effective with this flag enabled, speeding up EOLE 10 % more than Base, as an extra 12 % of the predictions became worthy.

7. PERFORMANCE OF CUSTOM EOLE IMPLEMENTATIONS

This section explores various EOLE implementations. We leverage the knowledge obtained in Section 6.3 to propose one customization in EOLE. We also evaluate EOLE implemented with restricted types of VP, in order to approximately illustrate the breakdown of performance contributions.

Figure 6 shows the speedups of all EOLE versions, described next. In this experiment, we averaged the simulation of 31 uniform checkpoints per benchmark, compiled with `-O3`. Each simulation is warmed up and measured during 50 million instructions each. The reference EOLE implementation, as described in Section 4.2, is labeled "**Orig**".

In Section 6.3, we concluded that unconditional branches may increase aliasing in tagged components of D-VTAGE (and also in TAGE). Then, we call "**No uncond**" a version that removes unconditional branches from the global branch history of both TAGE and D-VTAGE. As Figure 6 shows, this optimization only slows down 434.zeusmp and 447.dealII compared to "Orig", but the average speedup increases from 4.7 % to 4.9 %. Although the benefit is small, this customization is worth, as the hardware is actually simplified, with fewer micro-ops accessing prediction tables, and less checkpoint data.

CC predictions use only 4 out of 64 bits, so we may suspect that entries storing CC flags are not space-efficient and only increase conflicts. Hence, the "**No CC**" bars represent the speedups when CC values are not predicted (CC flags may still be derived from the predicted values). On average, not predicting CC values slows down

the execution by a negligible amount of 0.1 %, which is somehow expected. Indeed, should only branches use the CC flags, the value and branch predictors would perform redundant tasks. However, in 456.hmmer the degradation goes up to 5 %, because its main function `P7Viterbi` has 17 conditional moves, some of which in the inner loop. Consequently, D-VTAGE can predict them, while the branch predictor cannot.

Predicting FP/SIMD results may not actually bring considerable speedup, because they are rarely present in critical control flow decisions. The bars "**No FP/SIMD**" show the speedup when writes to the FP/SIMD RF are not predicted. In this case, the average slowdown is 0.7 %, going up to 9.2 % in 465.tonto, which actually is only sped up if FP/SIMD values are predicted.

Still in Figure 6, the "**Only loads**" bars represent the speedup of EOLE when only INT and FP results from loads are predicted. Even if on average load micro-ops only provide 23 % of the total coverage (Figure 4b, O3 bar), they account for almost 60 % of the total speedup achieved by EOLE. This result is in accordance with previous work [Calder et al. 1999] and highlights that the criticality of prediction is more relevant than having a high number of hits.

8. RELATED WORK

Introduction of value prediction. Lipasti et al. introduced the terms *value locality*, *load value prediction* and extensively argued why programs always have predictable values [Lipasti et al. 1996]. They proposed a mechanism to predict the last value of load instructions with confidence estimation and a fully-associative cache coherent with memory to store highly-predictable or constant loads. Gabbay and Mendelson independently proposed VP to collapse true data dependency and broadly characterized value predictability in function of data and instruction types [Gabbay 1996; Gabbay and Mendelson 1998]. Sazeides et al. and, independently, Gonzalez and Gonzalez studied and proposed mechanisms for a special case of VP, load address speculation, in order to anticipate memory accesses [Sazeides et al. 1996; González and González 1996].

Types of value predictors. The **Last Value** predictor [Lipasti et al. 1996; Gabbay 1996; Lipasti and Shen 1996] stores the previous results of instructions in tables and uses them to predict the result of subsequent executions of the same instruction. The **Stride** predictor stores in separate tables the last value of an instruction and a stride, which summed to the previous value produces the prediction [Gabbay 1996]. The **Per-path Stride** predictor uses global branch outcome history to improve accuracy [Nakra et al. 1999]. While the last value table is only indexed with bits of the instruction address, entries in the table holding the strides are accessed using both bits of the address and bits of branch history. **Context-based** predictors predict the next value by identifying patterns in previous sequences of values (contexts). A **Finite Context Method** (FCM) predictor of order k uses k preceding values [Sazeides and Smith 1997]. FCM predictors suffer from a very tight critical path on back-to-back predictions of tight loops. This critical path has been addressed by the **Value TAgged GEometric history length** (VTAGE) [Perais and Seznec 2014a] and the **Differential VTAGE** (D-VTAGE) [Perais and Seznec 2015a] predictors. The main idea is to have a base predictor (last value in VTAGE and stride predictor in D-VTAGE) backed by several tagged tables (components) indexed with a different number of bits from the global branch outcome history hashed with the instruction address.

Practical issues of value prediction. Previous work considered **validation at execute**, occupying execution resources until the prediction is validated and considering a yet to be implemented selective replay mechanism [Kim and Lipasti 2004; Lipasti et al. 1996; Lipasti and Shen 1996; Burtscher and Zorn 1999]. In this scenario, VP without a complex selective replay mechanism can not fully take advantage of large

instruction windows of current out-of-order processors (e.g., 192 in the Intel Haswell microarchitecture [Intel 2015]). Furthermore, bypassing and validating the prediction at execute would complexify even more the already complex out-of-order engine. An alternative is **validation at commit**: the predicted values are written in the RF at rename and any chain of dependent instructions can speculatively execute/writeback. This alternative has been shown to be cost-effective, provided that only very accurate predictions are used [Perais and Seznec 2014b]. If the predicted value is correct, the results can be committed, otherwise all instructions after the oldest misprediction must be squashed and re-fetched.

EOLE. With validation at commit, the main source of complexity is the extra access ports on the physical register file. EOLE mitigates this complexity by reducing the issue width and hence the overall complexity of the out-of-order execution stage [Perais and Seznec 2014a; Perais and Seznec 2015b].

This work differs from previous ones. We approach VP from the compiler point of view rather than the microarchitect's: We extensively characterize VP with various compiler optimizations.

9. CONCLUSION

This paper extensively characterized the interaction of value prediction (VP) and compiler optimizations. Not only this study opened new possibilities of value prediction mechanisms, but it also brings new insights to compiler developers.

Not surprisingly, less optimized programs benefit more from VP. Very basic and compulsory compiler passes enabled by gcc `-O1` effectively eliminate repetitive and very predictable stack loads. However, while most standard compiler optimizations are also needed by EOLE, some may actually either have a negative impact or be more useful to EOLE than ordinary processors, suggesting that new compiler optimizations and strategies are needed. Although this study used GCC, our findings can be generalized to any compiler, because we described the interactions of code patterns with VP.

In one hand, classic metrics such as accuracy and coverage have been historically employed to asses prediction performance. On the other hand, these may not be enough to model VP in a compiler. This paper proposed the usefulness metric, which better correlates with speedup than the coverage does. However, there may be more room for improvement, considering that the average correlation factors are between 0.5 and 0.8 depending on the optimization level.

When (and if) VP effectively appears in real hardware implementations, the definition and development of new compiler optimizations might enhance its potentiality.

ACKNOWLEDGMENTS

The authors would like to thank Kleovoulos Kalaitzidis, Biswabandan Panda, and anonymous reviewers for their comments and suggestions. This work has been partially supported by an Intel research gift.

REFERENCES

ARM. 2015. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile* (a.h ed.). ARM. ARM DDI 0487A.h (ID092915).

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (May 2011), 1–7. DOI: http://dx.doi.org/10.1145/2024716.2024718

Martin Burtscher and Benjamin G. Zorn. 1999. Exploring Last *n* Value Prediction. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on (PACT)*. 66–76. DOI: http://dx.doi.org/10.1109/PACT.1999.807407

Brad Calder, Glenn Reinman, and Dean M. Tullsen. 1999. Selective Value Prediction. In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on (ISCA)*. 64–74. DOI:http://dx.doi.org/10.1109/ISCA.1999.765940

George Z. Chrysos and Joel S. Emer. 1998. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Washington, DC, USA, 142–153. DOI:http://dx.doi.org/10.1145/279358.279378

Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 365–376. DOI:http://dx.doi.org/10.1145/2000064.2000108

Freddy Gabbay. 1996. *Speculative Execution based on Value Prediction*. Ph.D. Dissertation.

Freddy Gabbay and Avi Mendelson. 1998. Using Value Prediction to Increase the Power of Speculative Execution Hardware. *ACM Transactions on Computer Systems* 16, 3 (Aug. 1998), 234–270. DOI:http://dx.doi.org/10.1145/290409.290411

José González and Antonio González. 1996. *Memory Address Prediction for Data Speculation*. Technical Report. Universitat Politecnica de Catalunya. UPC-DAC-1996-50.

Intel. 2015. *Intel 64 and IA-32 Architectures Optimization Reference Manual*.

Ilhyun Kim and Mikko H. Lipasti. 2004. Understanding Scheduling Replay Schemes. In *Software, IEE Proceedings- (HPCA)*. 198–209. DOI:http://dx.doi.org/10.1109/HPCA.2004.10011

Mikko H. Lipasti and John Paul Shen. 1996. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Washington, DC, USA, 226–237. http://dl.acm.org/citation.cfm?id=243846.243889

Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. *SIGPLAN Not.* 31, 9 (Sept. 1996), 138–147. DOI:http://dx.doi.org/10.1145/248209.237173

Rick Merritt. 2009. ARM CTO: power surge could create 'dark silicon'. (October 2009). http://www.eetimes.com/news/design/showArticle.jhtml?articleID=220900080 [Accessed: 17 June 2015].

Pierre Michaud. 2016. Best-Offset Hardware Prefetching. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

Tarun Nakra, Rajiv Gupta, and Mary Lou Soffa. 1999. Global Context-Based Value Prediction. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On (HPCA)*. 4–12. DOI:http://dx.doi.org/10.1109/HPCA.1999.744311

Tony Nowatzki, Jaikrishnan Menon, Chen-Han Ho, and Karthikeyan Sankaralingam. 2015. Architectural Simulators Considered Harmful. *IEEE Micro* 35, 6 (Nov 2015), 4–12. DOI:http://dx.doi.org/10.1109/MM.2015.74

Arthur Perais and André Seznec. 2014a. EOLE: Paving the Way for an Effective Implementation of Value Prediction. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. IEEE Press, Piscataway, NJ, USA, 481–492. http://dl.acm.org/citation.cfm?id=2665671.2665742

Arthur Perais and André Seznec. 2014b. Practical Data Value Speculation for Future High-end Processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on (HPCA)*. 428–439. DOI:http://dx.doi.org/10.1109/HPCA.2014.6835952

Arthur Perais and André Seznec. 2015a. BeBoP: A Cost Effective Predictor Infrastructure for Superscalar Value Prediction. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on (HPCA)*. 13–25. DOI:http://dx.doi.org/10.1109/HPCA.2015.7056018

Arthur Perais and André Seznec. 2015b. EOLE: Toward a Practical Implementation of Value Prediction. *IEEE Micro* 35, 3 (May 2015), 114–124. DOI:http://dx.doi.org/10.1109/MM.2015.45

Arthur Perais and André Seznec. 2016. EOLE: Combining Static and Dynamic Scheduling through Value Prediction to Reduce Complexity and Increase Performance. *TACM Transactions on Computer Systems* (Feb. 2016), 34. https://hal.inria.fr/hal-01259139

Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. *SIGMETRICS Perform. Eval. Rev.* 31, 1 (June 2003), 318–319. DOI:http://dx.doi.org/10.1145/885651.781076

Nicholas Riley and Craig Zilles. 2006. Probabilistic Counter Updates for Predictor Hysteresis and Stratification. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on (HPCA)*. 110–120. DOI:http://dx.doi.org/10.1109/HPCA.2006.1598118

Y. Sazeides and J.E. Smith. 1997. The Predictability of Data Values. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on (MICRO)*. 248–258. DOI:http://dx.doi.org/10.1109/MICRO.1997.645815

Yiannakis Sazeides, Stamatis Vassiliadis, and James E. Smith. 1996. The Performance Potential of Data Dependence Speculation & Collapsing. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. IEEE Computer Society, Washington, DC, USA, 238–247. http://dl.acm.org/citation.cfm?id=243846.243890

André Seznec and Pierre Michaud. 2006. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction Level Parallelism* 8 (2006), 1–23.

Received ; revised ; accepted