Correct Wrong Path

Bhargav Reddy Godala[*] [†], Sankara Prasad Ramesh[*], Krishnam Tibrewala, Chrysanthos Pepi, Gino Chacon, Svilen Kanev, Gilles A. Pokam, Alberto Ros, *Senior Member, IEEE*, Daniel A. Jiménez, *Fellow, IEEE*, Paul V. Gratz, *Senior Member, IEEE*, David I. August, *Fellow, IEEE*

***Abstract*—Modern OOO CPUs have very deep pipelines with large branch misprediction recovery penalties. Speculatively executed instructions on the wrong path can significantly change cache state, depending on speculation levels. Architects often employ trace-driven simulation models in the design exploration stage, which sacrifice precision for speed. Trace-driven simulators are orders of magnitude faster than execution-driven models, reducing the often hundreds of thousands of simulation hours needed to explore new micro-architectural ideas. Despite the strong benefits of trace-driven simulation, it often fails to adequately model the consequences of wrong-path execution because obtaining such traces from real systems is nontrivial. Prior works exclusively consider either pollution or prefetching in the instruction stream/L1-I cache and often ignore the impact on the data stream. Here, we examine wrong path execution in simulation results and design a set of infrastructure for enabling wrong-path execution in a trace driven simulator. Our analysis shows the wrong path affects structures on both the instruction and data sides extensively, resulting in performance variations ranging from -3.05% to 20.9% versus ignoring wrong path. To benefit the research community and enhance the accuracy of simulators, we opened our traces and tracing utility in the hopes that industry can provide wrong-path traces generated by their internal simulators, enabling academic simulation without exposing industry IP.**

***Index Terms*—CPU Microarchitecture, Out of Order Execution**

I. Introduction

Accurate modeling of modern Out-of-Order (OOO) CPUs is very expensive in terms of simulator complexity and simulation times. Simulators facilitate prototyping of micro-architectures, but their complexity and simulation times can be significant bottlenecks in design space exploration. Consequently, architects often use simpler, and faster, though less accurate, trace-driven simulators, trading accuracy for speed. Trace driven simulators do not model wrong path (WP), as traces are typically generated from the execution of a workload on a real machine and thus can only contain correct path, executed instructions. Ignoring WP, however, can lead to incorrect estimation of performance.

Prior works [6], [9], [12] have proposed solutions to model WP, however those solutions are limited to the instruction stream, ignoring the data access effect. Instructions lines are fed to the instruction prefetcher pipeline which mimics instruction

Bhargav Reddy Godala and Gino Chacon from AheadComputing (bhargav.godala@aheadcomputing.com, ginoachacon@gmail.com). Sankara Prasad Ramesh from UCSD (spramesh@ucsd.edu). Krishnam Tibrewala, Chrysanthos Pepi, Daniel A. Jiménez, and Paul V. Gratz from Texas A&M University ({krishnamtibrewala, cpepis, djimenez, pgratz}@tamu.edu). Svilen Kanev from Google (skanev@google.com). Gilles A. Pokam from Intel (gilles.a.pokam@intel.com). Alberto Ros from University of Murcia (aros@ditec.um.es). David I. August from Princeton University (august@princeton.edu).

[*] Bhargav and Sankara Contributed Equally to this work.

[†]The work was done while the author was at Princeton University.

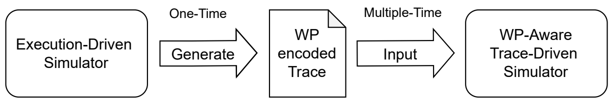

Fig. 1. WP traces from execution-driven to trace-driven simulator

cache pollution but overall impact of warming up data cache is not modeled. Further, while they often model either pollution effects of cache lines on the WP that go unused, or the prefetching effect of WP bringing in cache lines ahead of their eventual use; missing that both effects are possible even within a single workload. No existing approach encompasses *both* the pollution and prefetching impact on *both* the instruction stream and data stream. The alternative is to use more accurate execution-driven models, but they are orders of magnitude slower than trace-driven simulators [5], [7].

Instead of using actual execution to generate traces, we propose to use an execution-driven simulator to obtain traces which encode both WP and CP instructions. To facilitate this, we also propose a new trace format and a set of design changes to a trace-driven simulator to model speculative WP instructions and repair state of the CPU before executing in the correct path. Fig 1 shows our proposed simulation flow where a one-time cost of running the execution-driven model enables quick WP-aware trace-based design space exploration. Here we implemented our approach in the widely used ChampSim [7] simulator, though this approach could be implemented in any trace-driven simulator.

This paper makes the following contributions:

- Extensive analysis and metrics to measure the impact of WP instructions.
- A new tracing utility leveraging gem5 and a new WP trace format [3].
- Correct WP Configling in a trace-driven simulator [1].
- An open set of traces of widely used data center and SPEC workloads [4].

II. WP Config

Simulating instructions in the WP involves first, knowing what instructions are down the WP after a branch misprediction. Then letting those instructions flow from fetch till execute states of processor pipeline, repairing all pipeline stages by flushing instructions in the WP once the branch is resolved. Obtaining traces with WP instructions is very challenging as most tracing utilities only annotate committed instructions because they are obtained via binary instrumentation [10]. We propose using an execution-driven simulator to capture wrong-path instructions and encode them in the trace. Additionally, we suggest several modifications to a trace-driven simulator to support the execution and squashing of wrong-path instructions.

A. *WP Trace*

We propose a new trace format to encode a stream of instructions following a mis-speculation, either due to branch misprediction or load-store disambiguation failure. WP instructions executed before the redirection are discarded or squashed. These instructions are encoded in the trace following the mis-speculated instruction using additional fields to differentiate them from correct path instructions. Since it is not possible to obtain wrong-path instructions with existing tracing utilities on real CPUs, we use gem5 [5] to generate traces with wrong-path instructions. gem5 is an execution driven simulator so execution of the WP is modeled in an OOO CPU model.

In a modern core with a decoupled front-end, the Branch Predictor Unit (BPU) predicts lines ahead of fetch, populating the Fetch Target Queue (FTQ). This enables the Instruction Fetch Unit (IFU) to prefetch lines in advance. On a misprediction, multiple lines are fetched along the WP before branch resolution. If the trace lacks WP instructions, the IFU either stalls on mispredictions or prefetches only along the correct path, both misrepresenting Fetch-Directed Instruction Prefetching (FDIP).

B. *Implementing WP in a Trace Driven Simulator*

In typical trace driven simulators, such as ChampSim, when a branch instruction is encountered, the BPU is queried for the branch direction and target. The trace, which includes the correct branch target, is then compared with the predicted target to detect mispredictions. The front-end is stalled until the branch is resolved. Once the branch is resolved, a constant penalty is paid to account for pipeline repair cost and then fetch is resumed again to process stream of instructions in the correct path. Since wrong-path instructions are not seen in the pipeline, there is no need to squash or repair any structures in the pipeline, thus no pollution in the caches. To model WP instructions in a trace-driven simulator, all stages in the pipeline are modified as follows. The fetch stage is modified to continue streaming instructions from the trace after a misprediction. WP fetch halts either when a resteer signal is received, discarding the remaining WP trace and redirecting to the correct target, or when the WP trace ends, resulting in a fetch stall until a resteer signal arrives. To minimize stalls, long traces are collected along the WP. Resteers originate from the decode stage for unconditional branches or the execute stage for other mispredictions.

The decode stage identifies mispredicted unconditional direct branches. Once such a branch is found, all younger instructions following the mispredicted branch are squashed. The resteer signal is sent to the fetch stage, flushing the FTQ, Decode, and Fetch buffers. The core model tracks only the resolution of the parent mispredicted branch, while subsequent nested branches along the mispredicted path rely on trace information.

The execute stage handles all other mispredictions. At execute stage, instructions are executed out of order from the Reorder Buffer (ROB). The ROB contains instructions in the program order (in-order). Instructions in the ROB following a mis-speculating instruction are only in the WP. Once the mis-speculation is resolved all following instructions in the ROB are flushed. All pipeline structures leading to execute stage are flushed. Rename maps are repaired to remove stale dependencies introduced by the renamed WP instructions.

III. Methodology

A. *Simulation Infrastructure*

We have significantly enhanced the front-end and core model of ChampSim [7], with features like accurately modeled FDIP, post-fetch correction, and several bug fixes, including addressing the LSQ bug where loads occur prematurely and the OOO scheduling bug. Additionally, ChampSim has been expanded to support WP traces as outlined in Section II-B. The CPU core microarchitecture parameters (Table I) are modeled after Intel's Golden Cove core, (aka Alder Lake). Initially, the CPU is warmed up using 10 million instructions and, the simulation executes 100 million instructions in detailed mode (O3CPU).

We utilized the gem5 [5] simulator, enhanced with more accurate front-end models as detailed in [8], [11], to generate traces containing WP instructions. These traces were collected using the detailed O3CPU model with aggresive core parameters to ensure long WP traces. Since branch predictions can vary between simulators, we embed gem5's branch prediction as part of the trace used by ChampSim, mitigating the issue of inconsistent predictions and the lack of WP traces.

B. *Benchmarks*

We examined 14 widely-used server applications from various benchmark suites as depicted on Table II. These selected workloads contains traditional SPEC'17 CPU [2] and datacenter workloads with large code footprints from [8], [11].

Front-End	24 FTQ entries, 16K entries, 8-way BTB 64 KB TAGE [14]/ITTAGE [13] branch predictors, 32KB, 8-way L1I with latency of 2 cycles
Execution	512/194/144/112 ROB/Issue/Load/Store entries, 12 wide Decode and Retire, 448/400 Int/Vec. Registes
Caches	64KB, 16-way L1D with latency of 2 cycle, 1 MB, 16-way Private L2C with latency of 10 cycles, 2 MB, 16-way Shared LLC with latency of 20 cycles

TABLE I
Processor configurations for ARM ISA

Benchmark Suite	Benchmarks
SPEC [2]	401.bzip2, 429.mcf, 500.perlbench 514.leela, 520.omentpp, 531.deepsjeng, 557.xz
DaCapo	cassandra, kafka, tomcat
Renaissance	finagle-chirper, finagle-http
OLTB Bench	tpcc, wikipedia
Tailbench	specjbb, web-search, xapian
Cloudsuite V4	data-serving, media-streaming
Chipyard	verilator
Browser Bench	speedometer2.0

TABLE II
Benchmarks used to evaluate

IV. Results

The two simulated configurations are: 1) CP Config: Updated ChampSim model running traces with only Correct Path instructions, and 2) WP Config: The updated ChampSim model augmented with Wrong Path instructions. We analyze the impact of WP execution on cache performance across various cache subsystem levels. Since the core models of ChampSim and gem5 are not directly comparable, we focus on caches, which are common to both simulators. Table III compares L1I Misses and Hits Per Kilo Instructions between gem5, ChampSim WP, and ChampSim CP models, showing that the instruction cache performance of the gem5 model aligns closely with our ChampSim model.

Metric	Gem5	ChampSim WP	ChampSim CP
l1i_mpki	21.46	20.89	15.29
l1i_hpki	252.02	257.34	111.56

TABLE III
AVERAGE I-CACHE ACCESS STATS ACROSS ALL BENCHMARKS

A. *WP Impact on Instructions processed*

Fig 2 shows that on average, 83% more instructions are fetched in WP Config, with many benchmarks such as `429.mcf`, `514.leela`, and `finagle-chirper` showing greater than a 2.5-fold increase. This increase strongly correlates with branch misprediction rates, which control how often we proceed down the wrong path. ROB occupancy at the time of misprediction in WP is, on average, close to 2 times that seen on the CP. This also leads to considerably more ROB Full events in WP Config. Benchmarks with higher misprediction rates and lower ROB occupancy show the highest increases in WP instructions executed.

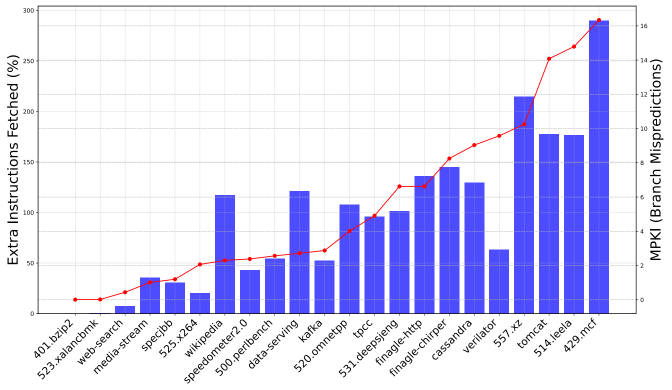

Fig. 2. Relative Increase in Instructions in WP vs CP.

B. *Cache State*

Modeling the execution of WP instructions has a direct impact on cache state. L1I cache is the most transformed by the excess WP instructions. Fig 3 shows that, on average, there is a 52% increase in misses in the L1I cache in WP Config compared to CP Config. Notably, three benchmarks more than double the number of misses. An even more pronounced effect is observed in the hit patterns of the L1I cache. In WP Config, there is an average increase of 140% in hits, with over half the benchmarks showing increases greater than 200%. The cache states of the other caches are also affected by WP. Enabling WP significantly alters cache access patterns. As shown in Fig. 4, L1D misses increase by 8.3%, L2C misses by 20%, and LLC misses by 30% on average. Conversely, Fig. 5 shows that L1D hits increase by 12%, L2C hits by 58.3%, and LLC hits by 7%. WP impacts data caches and cannot be ignored. L2C pressure correlates more strongly with performance than L1D. Higher hit rates may preserve high-reuse lines but can also pollute metadata updates, with effects that can be either positive or negative depending on the benchmark and program phase. Additionally, metadata used by cache policies, such as reuse percentage, stride, and other temporal and spatial patterns, differs with WP. Therefore, any work influenced by cache behavior is incomplete without accurately modeling WP.

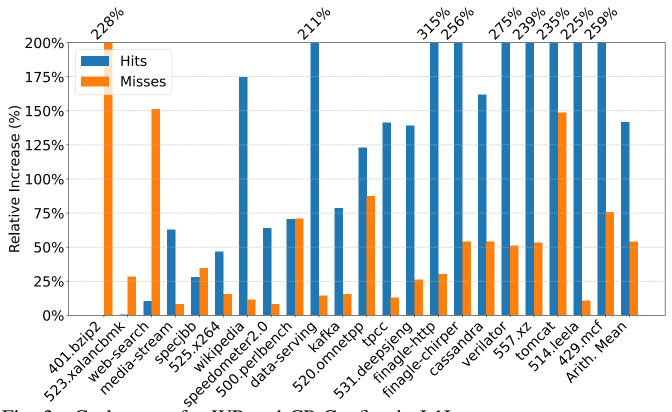

Fig. 3. Cache stats for WP and CP Configs in L1I

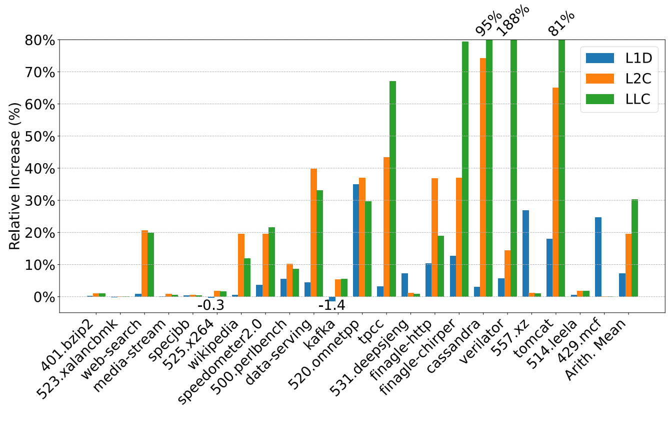

Fig. 4. Cache miss stats for WP and CP Configs in for all caches

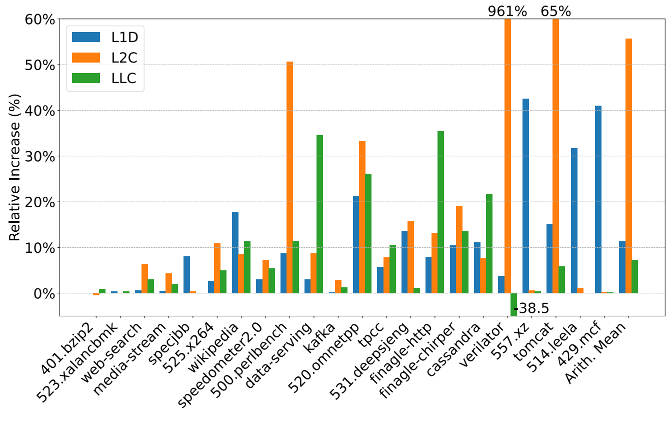

Fig. 5. Relative increase in cache hits of WP Config w.r.t CP Config

C. *WP Impact on Cache State*

WP-induced useless fills reduce effective occupancy of the caches and may also evict other critical useful lines, thus leading to lost performance. However, WP-induced useful fills help in prefetching lines accessed in later by the CP, resulting in lower cache misses. On average 67% of WP fills in L1I cache are useful. Thus, we see a strong prefetching effect. Further Fig 6

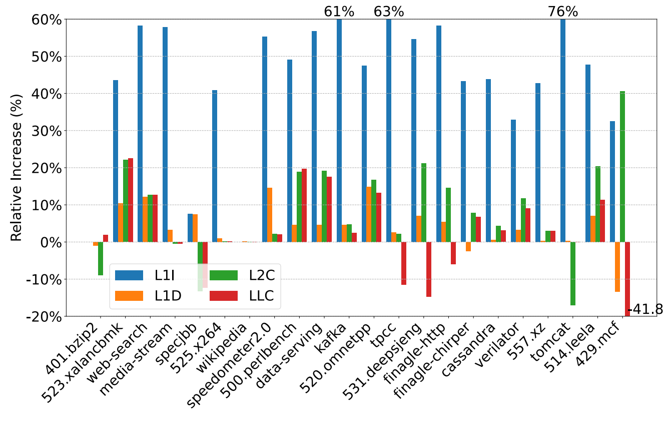

Fig. 6. Reduction of cache misses in the CP in WP Config

shows the reduction in misses along the CP when running in WP Config. L1I sees a reduction of 44% in the misses along the correct path, with L1D showing 3.85% and L2C showing 8.58% reduction. Although the overall misses increase, the prefetching effect of WP reduces the number of misses along CP across instruction and data caches.

D. Performance Impact

As discussed in the previous sections, the performance impact of WP cannot be approximated by considering cache behavior, branch mispredictions, or back-end impacts in isolation. WP influences these factors differently across various parts of the benchmarks, and only a comprehensive analysis of all these effects together can adequately capture the true impact of WP. Figure 7 shows the IPC speedup of WP Config over CP Config and the branch MPKI of all benchmarks. Benchmarks like `tomcat` and `cassandra` have high branch MPKI. In contrast, `523.xalancbmk`, and `525.x264` have low branch MPKI so it has negligible variance with WP. The expected trend is that the higher the branch misprediction rate is, the higher the number of WP instructions. However, in `verilator` the branch MPKI is 9.14 the WP instructions are only 55%(lower than other benchmarks with similar MPKI). This is because `verilator` has high number of unconditional branches which are resolved in the decode stage. Thus instructions fetched in the WP are squashed even before they reach decode stage.

The prefetching effect of WP dominates in majority of benchmarks showing positive impact with a speedup of up to 20.9% and a mean gain of 3.26%. WP can also have negative impact resulting in performance loss in seven benchmarks with `xapian` showing the minimum of -3.09% speeedup.

V. Conclusion

Complex modern OOO CPUs have speculative pipeline which have significant impact on the state of cache. The effect of WP execution is dependent on various characteristics of benchmarks. The impact of WP is often ignored by fast and less accurate trace-driven simulators. We provide a model to simulate WP in trace-drive simulator with a new trace format to encode WP instructions. As the traces are generated from a execution-driven simulator, the advantage of correct WP is achieved still at the quick speed of a trace-driven simulator. The WP enabled traces are robust enough to capture various characteristics of benchmarks and CPU parameters.

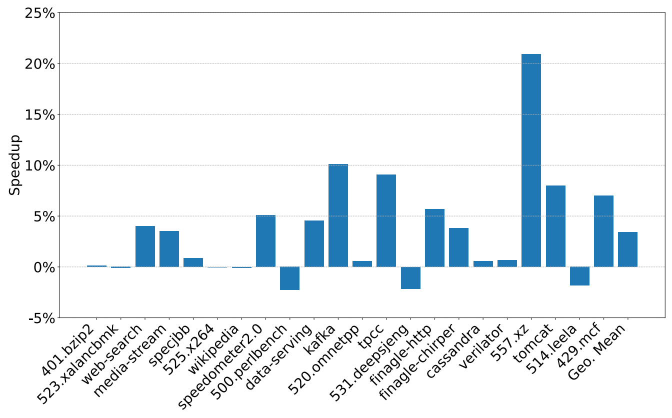

Fig. 7. IPC improvement with of WP Config w.r.t CP Config

VI. Acknowledgments

We thank the anonymous reviewers for their valuable comments and feedback to improve the content and quality of this paper. This research was conducted with advanced computing resources provided by Texas A&M High Performance Research Computing. The work was supported in part by NSF grants CCF-1912617, CCF-2107257, CCF-2107042, and generous gifts from Intel Corporation.

References

[1] "Champsim-wp," https://github.com/PrincetonUniversity/ChampSim.
[2] Spec 2017. https://www.spec.org/cpu2017/.
[3] "Tracer utility," https://github.com/PrincetonUniversity/gem5_FDIP.
[4] "Traces," https://tinyurl.com/4ak3kzhv.
[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718
[6] S. Eyerman, S. V. D. Steen, W. Heirman, and I. Hur, "Simulating wrong-path instructions in decoupled functional-first simulation," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 124–133.
[7] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The Championship Simulator: Architectural Simulation for Education and Competition," 2022.
[8] B. R. Godala, S. P. Ramesh, G. A. Pokam, J. Stark, A. Seznec, D. Tullsen, and D. I. August, "Pdip: Priority directed instruction prefetching," 2024.
[9] M. Hassan, C. H. Park, and D. Black-Schaffer, "Protean: Resource-efficient instruction prefetching," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '23. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3631882.3631904
[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN*, 2005.
[11] N. P. Nagendra, B. R. Godala, I. Chaturvedi, A. Patel, S. Kanev, T. Moseley, J. Stark, G. A. Pokam, S. Campanoni, and D. I. August, "EMISSARY: Enhanced Miss Awareness Replacement Policy for L2 Instruction Caching," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA*. ACM, 2023.
[12] A. Ros and A. Jimborean, "Wrong-path-aware entangling instruction prefetcher," *IEEE Transactions on Computers*, vol. 73, no. 2, pp. 548–559, 2024.
[13] A. Seznec, "A 64-kbytes ittage indirect branch predictor," in *JWAC-2: Championship Branch Prediction*, 2011.
[14] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *The Journal of Instruction-Level Parallelism*, vol. 8, p. 23, 2006.