Review of Ignite: Front-end Restoration Mechanism for Microarchitectural-state Targeting Instructions

David Cai, Madhushree Manjunatha, Pierce Wiegerling

November 2023

Contents

Initial Thoughts

This collaboration between two Edinburgh researchers and an ARM employee[0], focuses on solving the constant disruption to temporal locality mechanisms on a server's processing front-end, due to the on-demand nature of serverless functions, which are favoured by modern web developers. Their remedy is an OS-operated, front-end *restoration* mechanism that, they claim, bests the prefetching solutions currently available on the market, like Jukebox, Boomerang, and even the application of both at the same time.

In my opinion, they have provided a comprehensive selection of criteria for evaluation, precise specifications of their test hardware and methodology for accurate reproducibility testing, and they've provided a variety of competitors and configurations for comparison. Since a lot of these results display a marked improvement over the posited current solutions, Ignite could be effectively utilised to reduce the impact of lukewarm serverless invocations at server centres that must service many such functions.

1 Problem Introduction

The paper discusses the consequences of using serverless computing and the proposed solution for it. Serverless computing is characterised by independent, stateless functions and is said to be favoured amongst developers for its scalability and cost-effectiveness. Studies show that they have short execution periods (as low as a millisecond) and small memory footprints (128–256 MiB), which makes interleaving them financially favourable to both sides; however, this can cause issues for microarchitectural front-ends.

The colocation of many functions with short running times has been shown to lead to frequent context-switching and lukewarm invocations. A lukewarm invocation occurs when a function is re-invoked on the server after multiple interleaved invocations of other functions. This interleaving of other functions leads to thrashing of the front-end (when a processor's resources are filled with information from other functions, later leading to constant cache and branch misses), leaving the microarchitectural CPU in a cold state, as a result. It has been shown that these lukewarm invocations degrade the performance of the microarchitecture by up to two times, as compared to consecutive invocations, i.e. a warm microarchitecture.

It has been identified that the core problem lies within the front-end, due to the cold microarchitectural state of the branch target buffer (BTB) and the conditional branch predictor (CBP), resulting in poor prefetching and frequent pipeline flushes. The instruction delivery, branch identification, and branch prediction problems have been addressed by the proposed mechanism, Ignite.

2 Contributions and Solutions

The primary contribution of this paper is the proposition of the restoration mechanism for the front-end microarchitectural state, Ignite, to improve the performance during lukewarm invocations. It records a function's working set, by monitoring BTB insertions, to create a unified control flow record for that function, which is stored in the main memory during an invocation; when the function is re-invoked, the metadata is replayed from memory to unconditionally restore instructions and the branch prediction unit's (BPU's) state.

The paper additionally evaluates results from the testing and comparison of existing prefetchers, Jukebox and Boomerang, with an ideal front-end: it is shown that their combination improves the performance by only 20%, in contrast to 61% with an ideal front-end. So even that fails to address the high miss rates in the level 1 instruction cache (L1-I)(26 MPKI), BTB (13 MPKI), and CBP (21 MPKI), therefore identifying the instruction fetch and the cold BPU state as the issues.

They demonstrate that Ignite sees an average performance improvement of 43% with a reduction in the number of misses/mispredictions per thousand instructions (MPKI) in the L1-I, BTB, and CBP, compared to the existing approaches. This, as a result, has helped in industry as it has a low logic complexity, is easy to integrate with existing prefetchers, and supports thousands of functions on a server.

3 Evidence Supporting their Claims

The paper provides extensive benchmarks of Ignite against contemporary prefetchers in a lukewarm environment. They used gem5 to model a last-generation server CPU (Intel Xeon Ice Lake), this was probably done to represent the architecture of the average server CPU in use today. To simulate the lukewarm state induced by function interleaving, the microarchitectural structures were flushed between each invocation in the benchmark. In my opinion, there does not seem to be any issues with their experimental method, the microarchitectural flushing should represent a worst-case scenario; ideally, I would have liked to see a server workload benchmark used.

The overall performance analysis of the benchmarks show a significant performance gain from using Ignite across the board, with a 43% speedup over the baseline and far exceeding the closest competitor (Boomerang + JB, with 20% above baseline). The paper goes on to explain the sources of the performance gains. Ignite outperforms the other prefetchers significantly (more than halving the MPKI) for L1-I, BTB, and CBP miss coverage. It is clear that the performance increases are coming from the increase in L1-I, BTB, and CBP accuracy over its competition.

An aspect emphasised in the paper, is that current prefetchers, such as Jukebox and Confluence, rely on storing metadata on-chip in the LLC, occupying valuable cache capacity and reducing scalability whereas Ignite metadata is stored directly to memory. This is further explored by the memory bandwidth results showing that Ignite, while utilising slightly more memory bandwidth than baseline and less than other prefetchers, performs much better leading to greater performance per KiB of bandwidth.

4 Do the Experiments Prove the Contributions?

I think Sections 1 to 3 do a fantastic job of introducing the problem, corroborating prior studies, and presenting the efficacy of competing solutions. Figures 1[1] and 2 [0] show strong support for the general idea that performance suffers when function calls are interleaved, i.e. lukewarmly invoked; Figures 3 and 4 [0] make the point that the key flaws of the other solutions lies in the BPU, which is key to the efficient operation of prefetching solutions in general, confirming and extending the results of previous studies. However, the security considerations are quite terse. And although they miss Intel's own mitigations [1], it would be informative to see how far a professional team could get, in hijacking Ignite to execute or branch to malicious code.

The introduction of Ignite comprehensively describes the hardware modifications, the metadata format, the compression methodology, as well as when, where, and how to call this solution. The Methodology and Evaluation [0, s5-6] sections then pit Ignite (and improved configurations thereof) against combinations of other solutions. Finally, as mentioned above, although the memory usage due to metadata is much higher than that of JB, the reduction in unused instructions fetched amortises that usage, affording Ignite a lower bandwidth compared to the presented alternatives, indicating that even its weakest area bests its professed competitors. Altogether, a solid 'usage manual' as well as justification for Ignite's introduction are presented, thus a novel and effective solution is contributed to the field.

Regarding the use of **stress-ng** and flushing as methods to simulate thrashing on hardware and software simulations [, s5.2-.3] respectively, instead of making use of the other functions in their benchmark suite: I would like to see the tests re-performed with more realistic thrashing. There's no indication of which **stress-ng** test was used, e.g. I doubt the matrix test adequately thrashes the BPU due to the lack of distinct branches; I also believe that flushing would provide too clean a work surface, completely emptying all components and instigating compulsory misses (immediate stalls) as opposed to misprediction-misses (discovered and delayed re-steering).

5 Conclusion

With more and more computation moving to the cloud, this paper clearly tackles an increasingly relevant problem. The technical details of the paper are separated into meaningful sections which help explain the problem, evaluate previous solutions and layout their proposal while being easy to understand throughout. It is my opinion that the experiments they have run are thorough and represent a worst-case scenario for the CPU while the results conclusively show a significant performance increase for ignite in almost every metric.

[1] I think there may be an error in the figure caption for Figure 1, where it's the shaded bars that actually show back-to-back execution and the solid bars that show the interleaved execution.

Distribution of Work

David

Evidence Supporting their Claims and Conclusion with no use of AI tools.

Madhushree

Problem Introduction and Contributions and Solutions with no use of AI tools.

Pierce

Initial Thoughts and "Do the Experiments Prove the Contributions?", and surprisingly, no use of AI tools, at all.

References

[0] Warming up a Cold Front-End with Ignite, David Schall, Andreas Sandberg, Boris Grot, University of Edinburgh, 17th June 2023, https://www.research.ed.ac.uk/en/publications/warming-up-a-cold-front-end-with-ignite

[1] Indirect Branch Control Mitigation, Intel, *Revision 2.0, originally published May 2018*, Accessed: 22.11.2023 at 13:40, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html#inpage-nav-1