Charon: Expediting Secure Cache State Changes

Felipe Golitz Jofre
felipejofre2001@utexas.edu
University of Texas at Austin
Austin, Texas, USA

Edgar Turcotte
eturcotte@utexas.edu
University of Texas at Austin
Austin, Texas, USA

Harshadeep Kambhampati
harshadeepk@utexas.edu
University of Texas at Austin
Austin, Texas, USA

Abstract

Recently, side-channel attacks like Spectre and SpectreRewind have been used to exploit the speculative execution necessary for the increase in microarchitectural performance to leak sensitive data. While secure cache systems, like GhostMinion, allow for the minimization of the overhead required to section off speculative data, there is still performance to be gained before being able to fully adopt such systems. We propose Charon, a secure cache system building off of GhostMinion that commits speculative information to the traditional cache hierarchy at branch resolution rather than instruction retirement. With this early commitment alongside minimally conservative strictness ordering, we are able to see an average IPC improvement of 10% towards the baseline system with no such secure cache system.

Keywords: Security, Spectre, Secure Caches, Computer Architecture

ACM Reference Format:
Felipe Golitz Jofre, Edgar Turcotte, and Harshadeep Kambhampati. 2025. Charon: Expediting Secure Cache State Changes. In *Proceedings of May 01–05, 2025 (CS395T '25).* CS395T, Austin, TX, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Out-of-order execution[1] is an integral technique used to improve instruction-level parallelism and performance on modern systems. Alongside it, speculative execution (predicting control flow direction and running those instructions before control flow is resolved) is also necessary for performance.

However, misspeculated out-of-order execution can leak information through insecure side-channels, pushing state changes to the system that never should have happened. Exploiting this behavior, speculative side-channel attacks, such as Spectre[2] or SpectreRewind [3], can manipulate predictors to acquire sensitive information.

As such, an effective balance between preserving the performance benefits of speculative execution while also comprehensively eliminating any side channel information leakage must be found. Initial recommendations were to stop speculative execution all together, but the many decades of research into prediction mechanisms in computer architecture have proven that speculation is necessary for performance.

Secure cache solutions designed around "speculation-hiding" techniques attempt to address side-channel leakage through post-misspeculation rollback of those state changes. Speculative execution is permitted as normal, but upon misspeculation, corresponding state changes are properly reset. However, attacks such as SpectreRewind demonstrate it is insufficient to cleanup state changes post-misspeculation. Proper coverage of the entire system and concurrent execution of malicious instructions pose issues to rollback solutions.

Directly restricting speculative execution causes performance drawbacks; whenever side channel leakage is possible, the pipeline must be stalled until control flow is resolved.

GhostMinion [4] presents a cache and core modification that achieves a secure cache system protected against all such side-channel attacks. GhostMinion allows for speculative execution, but restricts access to other instructions observing speculative state in a "GhostMinion" cache through their model of "strictness ordering".

Speculative cache state changes are held off in the GhostMinion until they can safely be committed to the L1 cache. This happens upon the triggering instruction retiring from the ROB (at which point the instruction must be non-speculative and correctly predicted). Under strictness ordering, speculative information in the GhostMinion can be observed by safe instructions without hazardous side-channel leakage. GhostMinion aims to be a minimal-overhead secure cache implementation, providing a 2.5% overhead over a baseline system without ordering.

However, one key issue with the GhostMinion secure cache system is that it delays the timeliness of the L1 access stream. Because it stages state changes in the GhostMinion cache until the triggering instruction is retired, the exposed L1 access stream is less timely. Microarchitectural predictors suffer by training off this less timely L1 access stream [5], but training off speculative data compromises side-channel

CS395T '25,

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

security. This delay in timeliness may shape much of the overhead incurred by GhostMinion.

We present Charon: a lightweight modification to the GhostMinion secure cache that safely expedites the committing of secure cache state changes from the speculative "GhostMinion" cache.

- Through this "early-commit" modification, Charon enhances GhostMinion to minimally hinder the timeliness of other predictors (primarily prefetchers and cache replacement policies).
- Charon implements true "strictness ordering" rather than GhostMinion's "temporal ordering", an overconservative restriction of access to speculative state.
- We evaluate Charon to reclaim roughly 10% of the overhead incurred by Ghostminon, with an additional performance increase for scenarios with more complicated prefetchers in the L2 cache.

2 Background

2.1 Speculative Execution

Speculative execution is the execution of any instructions ahead of verified control flow by predicting branch directions (thus making them speculative and potentially incorrect).

Misspeculated instructions (instructions down an incorrectly predicted branch direction) are normally squashed before exposing protected data by verifying control flow before they can retire. But in an out-of-order core, misspeculated instructions may be executed before their control flow is resolved.

In such cases, this execution can result in some "state change" to the machine. For instance, speculative load instructions can bring data into the cache before resolving whether the speculation was correct. These state changes are not properly rolled back under normal microarchitectural behavior. Aware of this behavior, malicious attackers can manipulate predictors and derive sensitive information.

Spectre [2] is one such side-channel attack designed to access secret information using speculative execution. As can be gleamed from Figure 1, it mistrains predictors to speculate and execute some instructions that leak information into side-channels such as the cache. This misspeculated state change occurs using some kind of secret value. Then, the secret value can be retrieved by measuring side-channel leakage effects such as timing data. For example, speculatively loaded data in the cache can be accessed much faster.

SpectreRewind[3], as opposed to Spectre's use of speculative execution to perform the side channel attack, utilizes the very nature of the Out of Order (OoO) processor architecture to leak data. Since instructions that occur later down the pipeline can execute and finish before others within an OoO core, SpectreRewind exploits this by stalling functional units while accessing sensitive transient values, like registers. These instructions can then reaccess previous sensitive state

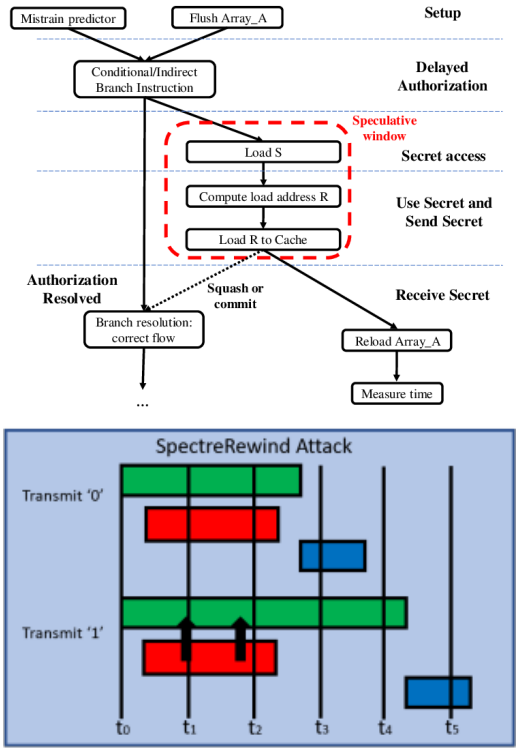

Figure 1. Spectre (Top) and SpectreRewind (Bottom) Attack Flow

and transmit this to instructions that are waiting to enter one of these stalled functional units, leaking data.

2.2 GhostMinion

GhostMinion is a cache and core modification proposed to design a comprehensively secure system. To enforce such a system, all speculative cache state changes are staged in a "GhostMinion" compartment of the L1 cache until they are later resolved to be non-speculative. Upon being resolved, speculative state is *committed* to the main cache hierarchy. Within the GhostMinion, access to speculative state is managed through "temporal ordering". This way, speculative cache state denies any information leakage.

For concurrently running instructions, GhostMinion defines a notion called "strictness ordering". This controls how instructions in the pipeline can access the GhostMinion and any speculative state. As they define it, consider two executing instructions x and y. For instruction y to securely observe any speculative state changes caused by instruction x in the GhostMinion:

$$commit(y) \dashrightarrow commit(x).$$

Where *commit(i)* is true if and only if instruction i is guaranteed to make it through the pipeline and retire without

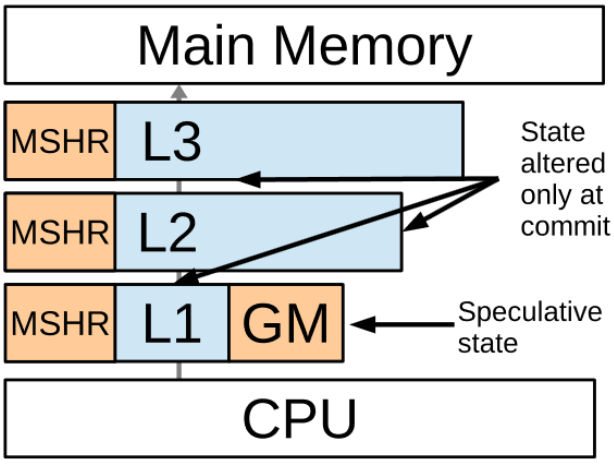

Figure 2. A GhostMinion is placed next to the L1 cache and serves as a staging area for speculative state changes until resolved. Using strictness ordering, instructions can only observe speculative state changes guaranteed to commit if the instruction commits (avoiding side channel leakage).

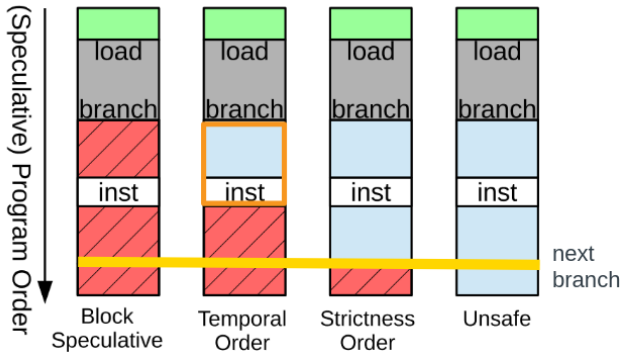

Figure 3. Side by Side Comparison of Several Orderings

being squashed. Under this model, no instruction y can observe and leak speculative state caused by a misspeculated instruction x.

As a simple example, let instruction x speculatively load data into the GhostMinion cache. Then, a concurrently running instruction y can only access that data in the GhostMinion if x is guaranteed to commit if y commits. Otherwise, it will find that memory access to be a cache miss (speculative state is hidden).

GhostMinion actually implements an over-approximation of strictness ordering known as "temporal ordering" for simplicity, as seen in Fig 3. Essentially, all instructions are "timestamped" upon entering the pipeline in their logical ordering. Instructions with later timestamps are deemed more speculative. Thus instructions can only observe speculative state caused by earlier timestamped (less speculative) instructions, the timestamp window restricted up to the last change in control flow.

While temporal ordering preserves the correctness of a secure cache, it is an overconservative approximation that restricts more access to speculative state than necessary. As seen in Fig 3, strictness ordering is the minimum restriction needed to ensure a secure cache. All of the state changes caused by the blue instructions after "inst" and before the next branch can safely be observed by "inst". However, temporal ordering does not permit this observation since these instruction are logically timestamped later.

This can hinder instruction parallelism when speculative state is safe to share. Consider that instruction y is safe to observe a cache load speculatively brought by instruction x under strictness ordering. However, if y is timestamped earlier (simply entered the pipeline earlier), then y is unable to access this speculative cache data and must wait for main memory access latency.

Additionally, we note that GhostMinion commits speculative state changes from the GhostMinion into the L1 upon instruction retirement (discussed in Section 4.1).

2.3 Secure Prefetching

Because speculative state can be potentially be leaked by hardware prefetchers, GhostMinion proposes "on-commit" data prefetching. Prefetchers are only trained off non-speculative commits and prefetches trigger on-commit. This way, prefetchers cannot be exploited for any information leakage, keeping the system secure.

"Secure Prefetching for Secure Cache Systems" analyzed the interactions between GhostMinion's secure cache system and hardware prefetchers [5]. They make several observations attributed to the worse performance of "secure prefetching". One key issue identified is that prefetcher timeliness is hindered by "on-commit" secure prefetching.

Prefetcher timeliness is key for prefetched data to be useful for servicing memory accesses. To best support prefetcher timeliness, prefetches should be triggered on-access for the next target prefetch. This ensures the prefetch occurs as soon as possible after getting access stream information. However, because of the delays introduced by GhostMinion in securely committing speculative state changes, an on-commit prefetch trigger worsens timeliness [5].

Additionally, while not discussed in their work, secure prefetchers must be trained "on-commit" as well. Thus, delays in committing speculative state changes to the L1 access stream results in prefetchers training off less timely data. This can also hurt the prediction accuracy of prefetchers, limiting their performance benefits. This timeliness issue with training extends to other hardware predictors such as cache replacement policies.

Thus, accelerating how quickly a secure cache is able to commit state changes is crucial to improving predictor timeliness, and any extraneous overhead should be eliminated

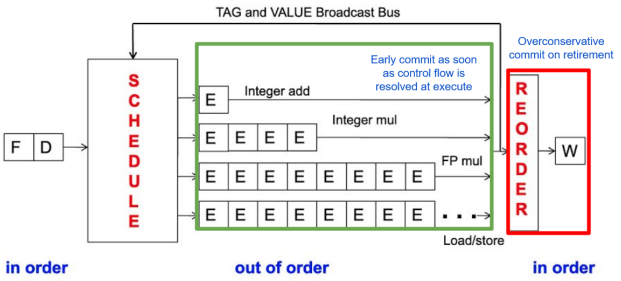

Figure 4. As shown in this out-of-order core model, GhostMinion commits speculative state changes at the responsible instruction's retirement. However, that instruction's control flow is earliest resolved once all branch instructions it is dependent on finish execution.

to ensure the performance of any prefetchers is as close to a baseline system as possible.

3 Charon: Early Commit GhostMinion

Here we propose Charon, a core modification to the GhostMinion secure system addressing its predictor timeliness issues and overconservative speculative state restriction.

In designing Charon, we hope to expect better results predictor performance since Charon aims to better interact with predictors than GhostMinion. We also expect to see better performance with Chiron from strictness ordering being more transparent with speculative state vs. GhostMinion.

3.1 Early Commit

First, we identify that there exists opportunity to safely commit speculative state changes to the L1 earlier than instruction retirement, which is where GhostMinion's implementation performs the commit. While state change commits are guaranteed to be safe at instruction retirement, control flow for a speculative instruction can be resolved as soon as its previous branch instructions finish executing.

Under secure cache constraints, speculative state changes are safe to commit once the responsible instruction is guaranteed to commit from the pipeline.

We perform early-commits by changing the trigger for speculative state change commitment from the responsible instruction retiring to the resolution of a branch. Once a branch is resolved, the instructions contained within its "control-flow block" are no longer speculative and can be released into the traditional memory hierarchy. To borrow a term from the world of compilers, an instructions' state changes may now be committed if all of the conditional instructions that *dominate* it have also been resolved.

As discussed, by early-committing speculative state changes to the L1, the L1 access stream that predictors train off of will remain more timely, which should benefit their accuracy. For cache replacement policies which only operate on non-speculative cache state, early commits help them operate more closely to how a non-secure transparent cache operates.

Additionally, under an "on-commit" prefetch trigger policy for a secure prefetcher, expediting these commits to occur earlier can directly help improve prefetcher timeliness since predicted requests are sent out sooner.

3.2 Strictness Ordering

In the original GhostMinion design, the authors utilize a much more constrictive ordering named *temporal ordering*, which they themselves admit is an oversimplification of the proposed strictness ordering. Charon implements full strictness ordering through allowing instructions to commit earlier. This early commit allows the influence of said instruction to propagate throughout the pipeline, which in turn may alleviate dependent branches or other such dependent instructions.

As discussed before, by implementing true strictness ordering, the transmission of speculative state is minimally restricted while maintaining a secure cache. This way, Charon does not incur penalties for hiding speculative state unnecessarily as temporal ordering does. Charon offers the additional degree of freedom for earlier timestamped instructions within a "control block" to access safe speculative state later in that control block (see Fig 4).

3.3 Implementation

Microarchitecturally implementing our design for Charon would require minimal changes to be built upon GhostMinion for gaining early-commit and strictness ordering benefits.

To identify upon branch direction resolution, which instructions' speculative state changes can now be committed (what instructions exist in the same "control flow block"), a Markov table of branch instructions and their corresponding control block instructions is maintained (similar to a secondary ROB). Upon instructions entering the pipeline, branch instructions (or other control flow instructions) create a new entry in this table while speculative state-changing instructions are added to the last branch entry's collection of dependent instructions.

Upon a branch being resolved after execution, the associated speculative state change instructions that can now be committed are stored in that branch's table entry. Thus, we can easily track which instructions can be early-committed as soon as their most speculative dominant instruction is resolved.

While just a general outline, the idea remains that the metadata overhead needed to track and perform early-commits is minimal. Thus, Charon exists as a lightweight modification easily built on top of GhostMinion.

Core	
Core	4 cores, 8-wide, out-of-order, 2.0 GHz
Pipeline	192-entry ROB, 64-entry IQ, 32-entry LQ/SQ, 256 Int / 256 FP regs, 6 Int ALUs, 4 FP ALUs, 2 Mult/Div ALUs
Caches with GhostMinions	
L1 I-cache	32 KiB, 2-way, 2-cycle latency, 4 MSHRs
L1 D-cache	64 KiB, 2-way, 2-cycle latency, 4 MSHRs
D/I GhostMinions	2 KiB, 2-way, accessed with I/D cache
Rest of System	
L2 cache	2 MiB, shared, 8-way, 20-cycle latency, 20 MSHRs
Memory	DDR3-1600 11-11-11
OS	Ubuntu 14.04 LTS

Table 1. gem5 System Configuration

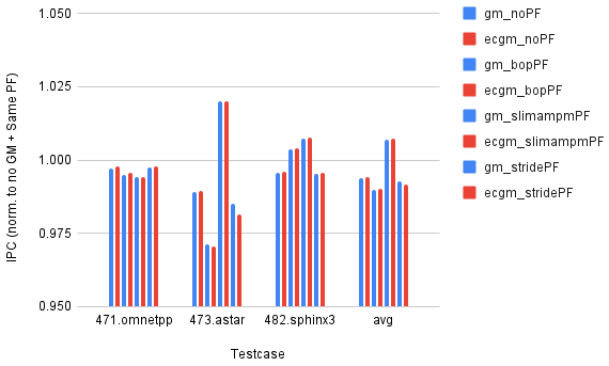

Figure 5. IPCs of GhostMinion and Charon normalized to the IPC of a No-GhostMinion system.

4 Methodology

For our experiments, we use the gem5 simulator with the system specifications shown in Table 1. We simulate SPEC CPU2006 benchmarks in syscall emulation mode, fast-forwarding and running for 100 million instructions each. The authors of GhostMinion also provided an image to run Parsec commands in full system mode, but as will be discussed in subsequent sections, there were issues in attaining this test data. All SPEC06 benchmarks were run at 100M fast-forward and then another 100M of subsequent instructions.

Benchmarks were ran testing with several natively implemented predictors in gem5. In the L1 cache we ran tests with Stride, SlimAMPM, and Best-Offset prefetchers. In the L2 cache, we tested using Stride, AMPM, BOP, ISB, SBBOE, SPP, STMS, and Tagged. Unless otherwise stated, the L1 cache defaults to no prefetcher and the L2 defaults to the Stride prefetcher.

5 Evaluation

In evaluating the performance of Charon, we try to test its interactions with various different hardware prefetchers at different cache hierarchy levels. Since Charon directly impacts the timeliness of the L1 access stream by early-committing speculative state changes, any predictors that engage with the L1 access stream are impacted. Both L1 and L2 prefetchers may train off the L1 access stream and thus are evaluated. No prefetcher test cases are also evaluated to attribute potential impacts on the cache replacement policy (further discussed later).

As can be gleamed from Figure 5, directly comparing GhostMinion to Charon results in very difficult to read graphs (note the zoomed in scale) since we are working within an already small GhostMinion overhead of 2.5%.

Since we are working with relatively small changes in IPC, we utilize a different method of data visualization: IPC overhead normalized to GhostMinion. The goal of Charon and other such secure caches is to reach as small of an overhead as possible, so showing a normalized IPC and MPKI as a percentage of the overhead as compared to GhostMinion allows for better visualization of the improvements Charon yields. Normalized overhead is calculated with the formula:

$$X_{normOH} = \frac{X_{ecgm} - X_{nogm}}{X_{gm} - X_{nogm}} \quad (1)$$

, where X is the value to represent, ECGM (early-commit GhostMinion) is Charon, GM is GhostMinion, and NOGM is No-GhostMinion. A nice quality of this representation is that **lower is always better** as it implies a reduction in the overhead which GhostMinion introduced. However, negatives are considered neutral. A negative normalized overhead implies that either Charon or GhostMinion performed better than a No-GhostMinion system, which should not be the case. This suggests that an element of luck was at play, so any performance wins or losses should be ignored.

5.1 Cache Replacement

The effects of Charon on the system's cache replacement performance can be seen when comparing Charon to GhostMinion with no prefetching involved.

As can be seen from Figure 6, early commits result in up to a 30% reduction in IPC Overhead as seen with *471.omnettp*. Overall, an average of 10% of the overhead has been shaved off by the inclusion of Charon. Since, the only prefetcher present in the system is the standard L2 Stride Prefetcher, most of this boost in performance is due to the improvement in the system's cache replacement schemes.

Going deeper into the caches shows that this performance is due equally from the L1D and L2 Cache. Figures 8 and 9 show that Charon results in a 2% decrease in MPKI Overhead for both the L1D and L2 Cache, whereas Figure 7 shows that the L1I shows little benefit from Charon's Early Commits.

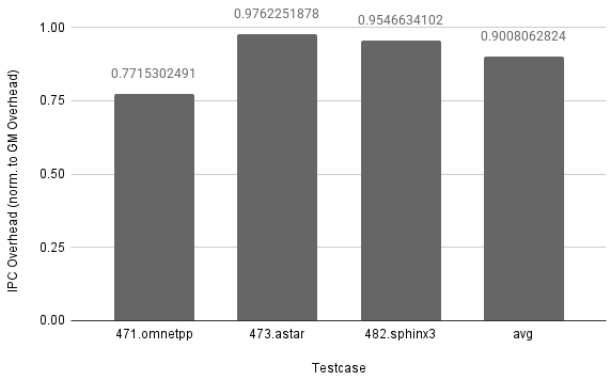

Figure 6. IPC Overhead of Charon normalized to the IPC Overhead of Ghostminion (No PF).

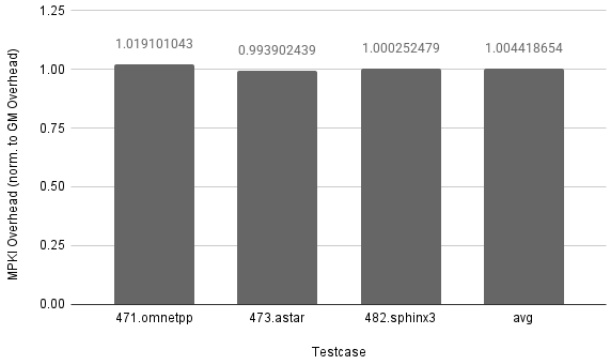

Figure 7. L1I MPKI Overhead of Charon normalized to the L1I MPKI Overhead of Ghostminion (No PF).

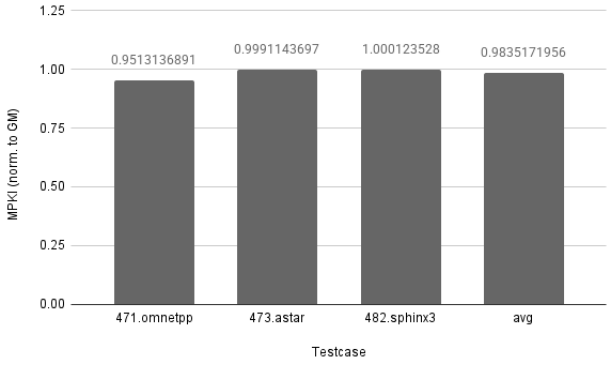

Figure 8. L1D MPKI Overhead of Charon normalized to the L1D MPKI Overhead of Ghostminion (No PF).

This shows that there is most likely very little headroom for performance increases in the IMinion.

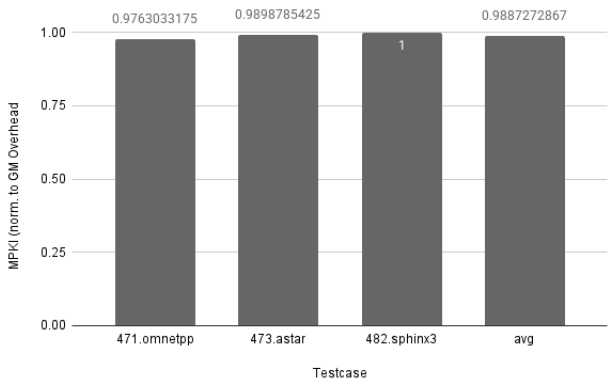

Figure 9. L2 MPKI Overhead of Charon normalized to the L2 MPKI Overhead of Ghostminion (No PF).

5.2 L1D Prefetching

As shown in prior work [5], the delay to cache state changes as a result of GhostMinion waiting to Commit instructions results in poor timeliness for prefetching applications. Charon's Early Commit system allows for the cache state to be more present, and results in more informed prefetching.

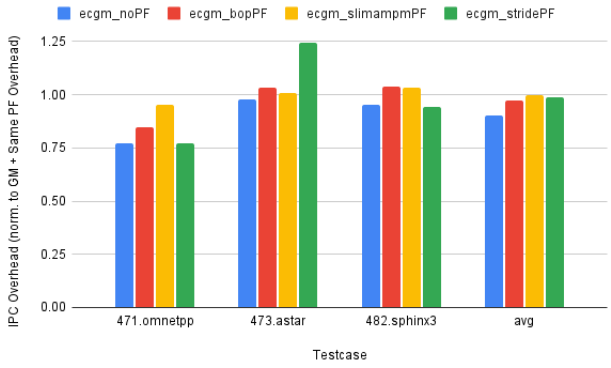

Figure 10. IPC Overhead of Charon normalized to the IPC Overhead of Ghostminion across various L1D Prefetchers.

Some prefetchers see improvement due to this more present state, such as BOP, which observes a 10% reduction in MPKI Overhead, as seen in Figure 11. However, we see in Figure 10 that most of these prefetchers actually result in a gain of Normalized IPC Overhead when added. This suggests that either the prefetching is too aggressive and may result in a performance hit due to increased DRAM traffic, or perhaps prefetchers perform better with GhostMinion than with Charon.

As is the case with the others graphs shown in this section, the coverage and accuracy figures are representative of the reclamation of the performance lost when utilizing a secure cache system over a traditional memory hierarchy.

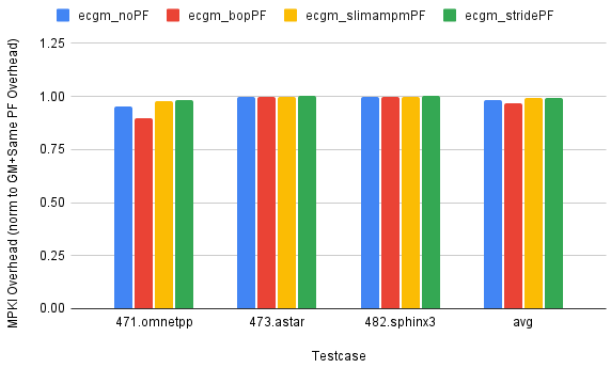

Figure 11. L1D MPKI Overhead of Charon normalized to the L1D MPKI Overhead of Ghostminion across various L1D Prefetchers.

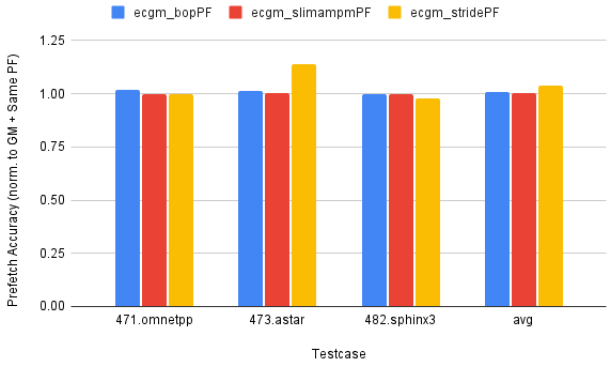

Figure 12. L1D Prefetch Accuracy Overhead of Charon normalized to the L1D Prefetch Accuracy Overhead of Ghostminion across various L1D Prefetchers.

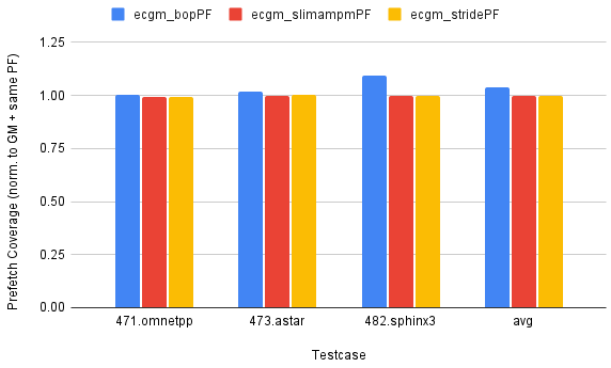

Figure 13. L1D Prefetch Coverage Overhead of Charon normalized to the L1D Prefetch Coverage Overhead of Ghostminion across various L1D Prefetchers.

As seen in figures 12 and 13, the data suggests that L1D prefetchers perform nearly the same within a GhostMinion as well as in Charon, which is curious. Overall, it seems that L1D prefetching is not aided by the improved timeliness of Charon.

5.3 L2 Prefetching

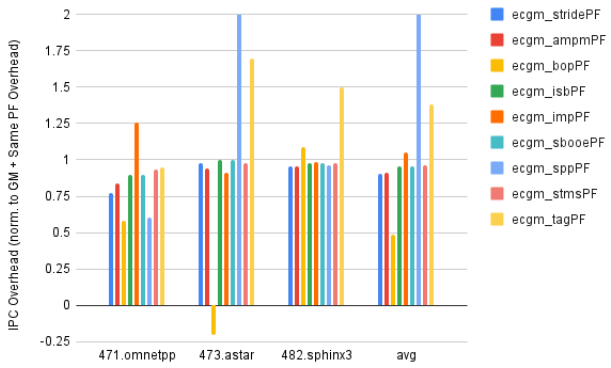

Figure 14. IPC Overhead of Charon normalized to the IPC Overhead of Ghostminion across various L2 Prefetchers. **Due to constraints, some data was cut off, namely spp.astar being 5.974 and spp.average being 2.512**

As can be immediately seen by Figure 14, the performances of L2 prefetchers are much more wild and varying. Looking at IPC alone, it is very clear that there are some prefetchers which benefit greatly from the improved timeliness of Charon, such as BOP, similarly to L1D prefetching. Additionally, it is apparent that some prefetchers, such as SPP and tagged, do not agree with the new system. Once again, MPKI seems relatively unaffected by most prefetchers, as can be seen with Figure 15.

Although there is a degree of performance improvement with these prefetchers, the inconsistency seen in the L1 prefetcher data rears its head here once again. The MPKI for the L2 cache has decreased across the board, with the exception being the tagged prefetcher once again.

In terms of accuracy, it was a coin flip whether a prefetcher would have an increase in accuracy or not. Certain prefetchers like ISB had absolutely no change in accuracy when compared to a GhostMinion system, exemplified by the overhead difference being zero in figure 16. Other predictors like AMPM performed significantly worse in comparison to the GhostMinion baseline. Overall, accuracy was a mixed bag.

Coverage tells a similar story, with some noted differences, The aforementioned ISB had a huge loss in coverage, whereas its accuracy stayed relatively consistent. The clear winner, discounting the negative outliers, seems to be AMPM.

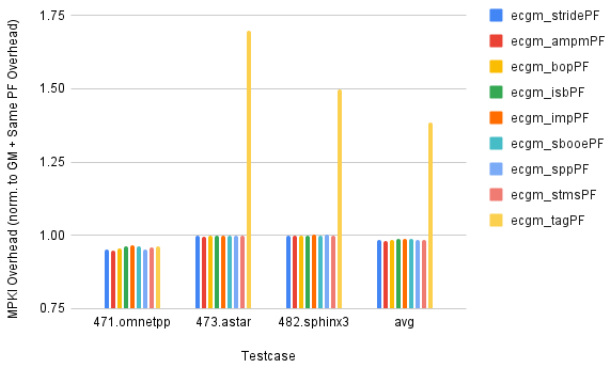

Figure 15. L2 MPKI Overhead of Charon normalized to the L2 MPKI Overhead of Ghostminion across various L2 Prefetchers. **Due to constraints, some data was cut off, namely spp.astar being 5.25 and bop.sphinx being 4.734**

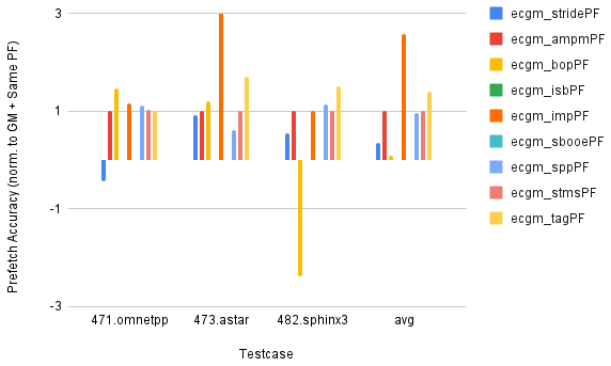

Figure 16. L2 Prefetch Accuracy Overhead of Charon normalized to the L2 Prefetch Accuracy Overhead of Ghostminion across various L2 Prefetchers. **Due to constraints, some data was cut off, namely imp.astar being 5.579**

5.4 Security

As will be expanded upon in section 8.1, there were numerous issues with running Parsec, but to the best of our knowledge Charon maintains the security of its predecessor GhostMinion.

6 Discussion

As we can see from our evaluations, Charon is able to reclaim around 10% of the overhead that its predecessor GhostMinion incurs, yielding a total overhead over a baseline system of roughly 2.25%, all through relatively simple microarchitectural solutions atop GhostMinion. The initial thought behind the development of Charon was to improve the accuracy of L1 prefetchers by early-committing to the L1 access stream.

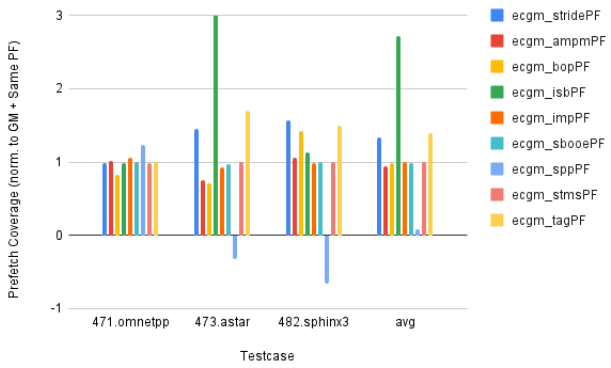

Figure 17. L2 Prefetch Coverage Overhead of Charon normalized to the L2 Prefetch Coverage Overhead of Ghostminion across various L2 Prefetchers. **Due to constraints, some data was cut off, namely isb.astar being 6.037**

However, as our methodology proves, while Charon does improve the timeliness of some prefetchers, it does not improve them significantly enough for overarching improvements.

In fact, the largest gains seem to stem from the cache replacement policies (no prefetcher test cases) within the memory hierarchy. Having state affect the caches closer to the time of their resolution allowed LRU to perform better, and because of this the L2 prefetchers gain performance when compared to the GhostMinion baseline. This may explain why our no prefetcher baseline performs so well - it isolates the LRU and prefetcher interplay and exemplifies how improving timeliness can aid prefetchers. While orthogonal to the scope of this project, the prefetch filter [5] may yield significant improvements through Charon.

7 Future Work

We rank our future works by level of viability and urgency, beginning with the fundamental problems we encountered when working with GhostMinion and expand upon future expansions to Charon.

7.1 Roadblocks

The most fundamental future work is the continued modification of the GhostMinion repository to enable the effective running of benchmarks. As we have alluded to throughout this paper, there were many issues we ran into while implementing Charon:

To begin with, half of the Spec06 benchmarks given to us were not functional with GhostMinion's modification of gem5. While soplex and xalancbmk were able to run at much lower thresholds, around 100K instructions for both fast forward and data collection, mcf would refuse to terminate even when running a couple hundred instructions. Becasue of this, we were only able to retrieve data from 3 of the given benchmarks.

GAP test cases were unable to run in an achievable time frame either, with both **bfs** and **cc** not terminating at the baseline captures of 100M within 48 hours. While they may have terminated given more time, since they are much more reliant on cache state changes and thus GhostMinion's effects are exacerbated. Being able to run these in a timely manner would allow for a wider variety of data to be extracted.

Parsec test cases worked well for the baseline implementation of GhostMinion, but the behavior of the tests was non-deterministic. Tests would run for indeterminate amounts of instructions, and for the one instance where both baseline and GhostMinion ran for the same amount of instructions, the test case terminated after running for more than 48 hours. In addition to this, there were several instances where Parsec would run for 24 hours or more successfully, but would fail at a given point and unceremoniously delete any data that had been tracked. Despite these issues, we were able to capture the fact that Charon conforms to the Spectre securities that GhostMinion does, as Parsec throws runtime errors if non-secure behavior is able to be executed, but were unfortunately unable to retrieve run data for these test cases

7.2 Further Expansion

We propose two further expansions for Charon: microarchitectural support for execution protection, and enforcing in-order early commits.

First, our early commit system only takes into account time steps that are based on branches: a control flow block is allowed to modify state if all the branches it depends upon have resolved. This does not take into account whether instructions will trigger an exception, as in a Spectre scenario a user program could attempt to load memory from kernel space. This leads to a scenario where loads that would trigger an exception are allowed to modified space. While there does exist operating system[6] support to ensure this type of attack will not be possible, future work would be directed into creating a purely microarchitectural solution.

Secondly, there should be a microarchitectural system that enforces in-order commits from the GhostMinion. The current implementation could lead to a scenario where a load instruction is still executing but the branch it is dependent on is resolved, marking the entire control flow block as committed. Since the load has not been committed, its timestep will not be set to 0, and thus will only be committed at the point of retire, which is when GhostMinion would have commited the instruction. This does not compromise security, but there may be a performance gain to ensure these commits happen in-order.

Finally, and most simply, due to many of the performance gains coming from the more accurate LRU state of the L1 cache, using different L2 replacement policies may yield greater results. In addition to this, improvising the cache state of the L2 cache could affect LLC prefetchers, which are much heavier and require more up-to-date data, and thus may benefit more from timeliness.

8 Conclusion

There is still a lot of work that needs to be done in the security landscape to truly mitigate the performance loss that having such security entails. Our work in Charon allows for a reclamation of 10% of the overhead that GhostMinion incurs while still maintaining side-channel protection. We have also proven that we are able to improve the performance of some predictors in both the L1 and L2 caches, but that a large chunk of the performance increase comes from LRU more accurately representing the current state of the machine.

9 Github Link

Our repository can be found at the following link: https://github.com/FelipeGJofre/CS395T_Term_Project

References

[1] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of Research and Development, vol. 11, 1967, pp 25 - 33. Principles and Examples, McGraw-Hill, 1982.

[2] P. Kocher et al., "Spectre Attacks: Exploiting Speculative Execution," 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2019, pp. 1-19, doi: 10.1109/SP.2019.00002.

[3] Jacob Fustos, Michael Bechtel, and Heechul Yun. 2020. SpectreRewind: Leaking Secrets to Past Instructions. In Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security (ASHES'20). Association for Computing Machinery, New York, NY, USA, 117–126. https://doi.org/10.1145/3411504.3421216

[4] Sam Ainsworth. 2021. GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation. In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21). Association for Computing Machinery, New York, NY, USA, 592–606. https://doi.org/10.1145/3466752.3480074

[5] S. Nath, A. Navarro-Torres, A. Ros and B. Panda, "Secure Prefetching for Secure Cache Systems," 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO), Austin, TX, USA, 2024, pp. 92-104, doi: 10.1109/MICRO61859.2024.00017.

[6] Jack Tang. 2017. KAISER: Hiding the kernel from user space. https://lwn.net/ Articles/738975/

Accepted 5 May 2025