UNIVERSIDAD DE MURCIA

Escuela de Doctorado

TESIS DOCTORAL

# *Predicción sensible al contexto precisa para técnicas especulativas en procesadores*

*Accurate context sensitive prediction for speculative techniques in processors*


AUTOR/A Sebastián Sumin Kim

DIRECTOR/ES Alberto Ros Bardisa


2025

UNIVERSIDAD DE MURCIA

Escuela de Doctorado

TESIS DOCTORAL

# *Predicción sensible al contexto precisa para técnicas especulativas en procesadores*

*Accurate context sensitive prediction for speculative techniques in processors*


AUTOR/A Sebastián Sumin Kim

DIRECTOR/ES Alberto Ros Bardisa


2025

UNIVERSIDAD DE MURCIA

# DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA EN MODALIDAD DE COMPENDIO O ARTÍCULOS PARA OBTENER EL TITULO DE DOCTOR/A

*Aprobado por la Comisión General de Doctorado el 19 de octubre de 2022.*

Yo, D. Sebastián Sumin Kim, habiendo cursado el Programa de Doctorado en informática de la Escuela Internacional de Doctorado de la Universidad de Murcia (EIDUM), como autor/a de la tesis presentada para la obtención del título de Doctor/a titulada:

| |
|---|
| Enhanced context sensitive prediction for speculative techniques in modern processors / Predicción sensible al contexto mejorada para técnicas especulativas en procesadores modernos |

y dirigida por:

| |
|---|
| D.: Alberto Ros Bardisa<br>D.:<br>D.: |

**DECLARO QUE:**

La tesis es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, de acuerdo con el ordenamiento jurídico vigente, en particular, la Ley de Propiedad Intelectual (R.D. legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, modificado por la Ley 2/2019, de 1 de marzo, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), en particular, las disposiciones referidas al derecho de cita, cuando se han utilizado sus resultados o publicaciones.

Además, al haber sido autorizada como prevé el artículo 29.8 del reglamento, cuenta con:

- *La aceptación por escrito de los coautores de las publicaciones de que el doctorando las presente como parte de la tesis.*
- *En su caso, la renuncia por escrito de los coautores no doctores de dichos trabajos a presentarlos como parte de otras tesis doctorales en la Universidad de Murcia o en cualquier otra universidad.*

Del mismo modo, asumo ante la Universidad cualquier responsabilidad que pudiera derivarse de la autoría o falta de originalidad del contenido de la tesis presentada, en caso de plagio, de conformidad con el ordenamiento jurídico vigente.

Murcia, a 03 de Septiembre de 2025

*Fdo.: Sebastián Sumin Kim*

| Información básica sobre protección de sus datos personales aportados: | |
|---|---|
| Responsable | Universidad de Murcia.<br>Avenida teniente Flomesta, 5. Edificio de la Convalecencia. 30003; Murcia.<br>Delegado de Protección de Datos: dpd@um.es |
| Legitimación | La Universidad de Murcia se encuentra legitimada para el tratamiento de sus datos por ser necesario para el cumplimiento de una obligación legal aplicable al responsable del tratamiento. art. 6.1.c) del Reglamento General de Protección de Datos |
| Finalidad | Gestionar su declaración de autoría y originalidad |
| Destinatarios | No se prevén comunicaciones de datos |
| Derechos | Los interesados pueden ejercer sus derechos de acceso, rectificación, cancelación, oposición, limitación del tratamiento, olvido y portabilidad a través del procedimiento establecido a tal efecto en el Registro Electrónico o mediante la presentación de la correspondiente solicitud en las Oficinas de Asistencia en Materia de Registro de la Universidad de Murcia |

*Esta DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD debe ser insertada en la quinta hoja, después de la portada de la tesis presentada para la obtención del título de Doctor/a.*

# Abstract

With each new generation, high-performance processors adopt increasingly aggressive techniques to extract more performance. Out-of-order execution, advanced memory communication mechanisms, and instruction fusion all contribute to higher performance, but they rely heavily on speculation. As designs grow wider and deeper, these speculative mechanisms become more vulnerable to misprediction, which can cost up to 15% of the processor's performance potential.

In this context, memory dependence prediction is essential to ensure that loads do not bypass older stores to the same address. Similarly, instruction fusion has been used to reduce pressure on the pipeline, though current industrial implementations focus almost exclusively on simple, consecutive patterns. Only a single prior proposal has explored speculative fusion of non-consecutive memory instructions using a predictor.

However, state-of-the-art approaches in both memory dependence prediction and instruction fusion prediction remain limited. Most are directly inspired by branch predictor designs, which, as we claim in this thesis, are not well suited for these domains and often result in inefficiencies.

The central claim of this thesis is that context information must be carefully

selected for accurate prediction. The use of fixed lengths or finding the best suited among a set through brute-force, like how it is done in branch prediction, can lead to an explosion of paths when the history is larger than necessary, or to aliasing when shorter.

Instead, we state that each prediction must be trained only with the information between the instructions involved in it. If the same path between those two instructions recur, then the outcome of the prediction will likely recur too.

Following this premise, the objective of this thesis is to revisit speculative hardware mechanisms from the perspective of context sensitivity. Our working hypothesis is that prediction accuracy and efficiency can be substantially improved if mechanisms adapt to the specific dynamic context of each case, rather than relying on static or generic lengths. To test this hypothesis, we design and evaluate new predictors that exploit the relevant execution context to expand the reach of speculation while minimizing its risks.

The first part of this thesis revisits the Store Sets predictor, still widely used as a reference mechanism for memory dependence prediction. Our analysis identifies five key limitations, most rooted in the unnecessary serialization of memory instructions.

Building on these insights, we introduce PHAST, a new memory dependence predictor that departs from fixed-distance and set-based approaches. PHAST learns from the actual execution path between a conflicting store and its dependent load, enabling it to predict precisely the store from which the load should receive its value. This path-sensitive training eliminates many false dependences and significantly reduces pipeline squashes. With only 14.5 KB of hardware budget, PHAST reduces misspredictions per kilo instruction (MPKI) by 62% on average and achieves a 1.29% average speedup over the state-of-the-art, with improvements of up to 22%, narrowing the gap to an ideal predictor to just 1.5%.

The final contribution addresses instruction fusion. Prior work on non-consecutive fusion relies on a Tournament branch predictor with long histories, a design that we found to be inappropriate for this task. To address these shortcomings, we present FLIP, a new fusion predictor that applies the same context-sensitive principle as PHAST. FLIP identifies profitable non-consecutive fusion opportunities among loads and stores and introduces a set of Non-consecutive Fusion Optimizations (NFOs) to relax unnecessary constraints and mitigate harmful cases. Experiments show that FLIP improves performance by 2.44% on SPEC CPU 2017 and 2.94% on MiBench, while reducing MPKI by 83%.

# Resumen

Con cada nueva generación, los procesadores de alto rendimiento se vuelven cada vez más agresivos en su búsqueda de un mayor rendimiento. Para ello, se han introducido varios mecanismos, tales como la ejecución fuera de orden (OoO), las técnicas avanzadas de comunicación de memoria y la fusión de instrucciones. Todos ellos comparten un mismo objetivo: ocultar la latencia de las instrucciones y de los accesos a memoria.

A pesar de estos esfuerzos, todavía se pierde hasta un 20 % del potencial total de rendimiento en el subsistema de memoria. Además, a medida que los procesadores adoptan diseños más agresivos, dependen en gran medida de técnicas especulativas que son inherentemente vulnerables a errores de predicción, lo que provoca una pérdida adicional del 15 % del potencial de rendimiento.

En este contexto, la predicción de dependencias de memoria juega un papel fundamental al garantizar que los *loads* no se ejecuten antes que los *stores* más antiguos que acceden a la misma dirección. De manera similar, la fusión de instrucciones se ha explorado como una vía para mejorar la eficiencia, aunque las implementaciones actuales suelen estar restringidas a patrones consecutivos y simples. Solo una propuesta previa ha considerado la fusión especulativa de instrucciones no consecutivas, con la ayuda de un predictor.

Sin embargo, el estado del arte tanto en predicción de dependencias de memoria como en predicción de fusión de instrucciones siguen siendo limitados. La mayoría se inspiran directamente en los diseños de predictores de saltos que, tal y como se declara en esta tesis, no se adaptan correctamente a estos dominios y, en consecuencia, suelen resultar ineficientes.

La principal afirmación de esta tesis es que la información de contexto debe seleccionarse cuidadosamente para obtener predicciones precisas. El uso de longitudes de historia fijas o el intentar encontrar la mejor longitud de entre un conjunto a base de fuerza bruta, tal y como se realiza en predicción de saltos, puede llevar a una explosión de caminos cuando la historia es más larga de lo necesario, o a aliasing cuando es más corta.

En su lugar, proponemos que cada predicción debe entrenarse sólo con la información de contexto delimitada por las instrucciones involucradas en ella. Si el mismo camino de ejecución se repite entre dichas instrucciones, entonces es altamente probable que el resultado de la predicción sea correcto.

El objetivo central de esta tesis es revisar las técnicas especulativas desde la perspectiva de la sensibilidad al contexto. La hipótesis de trabajo es que la precisión y eficiencia de la predicción pueden mejorar sustancialmente si los mecanismos son capaces de adaptarse al contexto dinámico específico de cada caso, en lugar de basarse únicamente en longitudes estáticas o genéricas. En consonancia con ello, este trabajo diseña, implementa y evalúa predictores que aprovechan el contexto de ejecución para ampliar el alcance de la especulación minimizando al mismo tiempo sus riesgos y costes.

Para la evaluación de las propuestas de esta tesis se ha empleado principalmente la simulación ciclo a ciclo de un procesador. No obstante, dado que los simuladores de código abierto presentan limitaciones en el modelado de la desambiguación de memoria y en la ejecución especulativa de *loads*, se optó

por utilizar un simulador interno desarrollado por el grupo de investigación de este departamento. En cada contribución, el procesador modelado se basó en la generación más reciente de Intel disponible en ese momento. Respecto a las aplicaciones empleadas en la evaluación, se consideraron los conjuntos de referencia SPEC CPU 2017 y MiBench.

Como primer paso, analizamos las limitaciones del predictor de dependencias de memoria *Store Sets* [18], aún ampliamente utilizado como referencia. Propuesto en 1998 por Chrysos y Emer, este mecanismo asocia a cada *load* el conjunto de *stores* con los que ha presentado conflictos. Cada *store* solo puede pertenecer a un único conjunto, de modo que, cuando dos *loads* comparten un mismo *store*, sus conjuntos se fusionan.

El predictor se compone de dos tablas. La primera, denominada *Store Sets Identification Table (SSIT)*, almacena los identificadores de conjunto o *SSID*, a los que acceden tanto *loads* como *stores* para determinar su pertenencia. La segunda, llamada *Last Fetched Store Table (LFST)*, guarda el identificador del último *store* de ese conjunto que ha sido decodificado, junto con un bit de validez que indica si la instrucción aún no se ha ejecutado. Tanto *loads* como *stores* consultan esta tabla mediante el *SSID* obtenido de la SSIT. Con esta información, los *loads* y *stores* establecen dependencias si el bit de validez está activo, mientras que los *stores* actualizan la tabla con su propio identificador.

Nuestro análisis identificó cinco deficiencias principales, en su mayoría relacionadas con una excesiva serialización de las instrucciones de memoria. En primer lugar, el *aliasing*, que surge cuando múltiples instrucciones de memoria comparten la misma entrada de la tabla. En segundo lugar, el predictor fuerza siempre al *load* a depender del *store* más joven, garantizando así el orden correcto de ejecución, pero a costa de rendimiento, en especial cuando ese *store* no accede a la misma dirección que el *load*.

En tercer lugar, este predictor induce la ejecución en orden de los *stores*, retrasando innecesariamente aquellos que no tienen un conflicto real con el *load*. Para mitigar este problema, se propuso ignorar conflictos en los casos en que el *load* ya hubiera recibido el dato de un *store* más joven. En cuarto lugar, aparece la serialización en bucles: cuando un *load* entra en conflicto con un *store* de iteraciones previas, acaba dependiendo de la instancia más cercana de dicho *store*, a la par que obliga a ejecutar en orden todas las instancias de dicho *store*.

Por último, identificamos las dependencias ocasionales: situaciones en que un *load* y un *store* acceden a la misma dirección solo esporádicamente. En estos casos, tras el primer conflicto el *store* pasa a formar parte permanente del conjunto del *load*, forzando dependencias innecesarias en el futuro hasta que el predictor se reinicie.

Para evaluar estas limitaciones, comparamos *Store Sets* con un oráculo, eliminando progresivamente cada una de las restricciones señaladas hasta dejar las dependencias ocasionales. Nuestros resultados muestran que eliminar la serialización de los *stores*, sin un mecanismo adicional que informe al *load* de todas sus dependencias, redujo el rendimiento debido al reordenamiento incorrecto frente a sus *stores* asociados. Asimismo, el *aliasing* explicó aproximadamente una séptima parte de la diferencia con respecto a la predicción ideal, mientras que corregir la gestión de dependencias en bucles permitió reducir la distancia con el oráculo en un 20 %. En conjunto, la aplicación de todas las mejoras situó el rendimiento de *Store Sets* a un 4 % del oráculo, diferencia atribuida principalmente a los fallos en frío tras cada reinicio del predictor y a las dependencias ocasionales.

A partir de estas observaciones, desarrollamos *PHAST*, un predictor de dependencias de memoria que se aparta de los enfoques convencionales basados en longitudes de historias fijas o en conjuntos de *stores*. PHAST se basa en dos elementos claves. Primero, esperar a un conjunto de *stores* no es lo adecuado, puesto

que en la mayoría de casos, el *load* obtendrá el dato de uno solo, típicamente el más reciente, por lo que esperar a otros stores más antiguos no tiene sentido.
Segundo, la información de contexto a utilizar para cada predicción ha de ser únicamente la que se encuentra comprendida entre el store y el load involucrados en el conflicto. Utilizar historias de tamaño fijo o seleccionar por fuerza bruta la que mejor se adapte no es óptimo, como ya se indicó anteriormente.
Basándose en las dos claves anteriores, PHAST se entrena utilizando el camino de ejecución real entre un *store* conflictivo y el *load* dependiente, lo que le permite predecir con gran precisión el *store* concreto que adelantará el dato. Este nuevo enfoque de selección de la información de contexto, elimina numerosas falsas dependencias y reduce el número de invalidaciones respecto al estado del arte. Nuestros experimentos de simulación demuestran que, comparando PHAST con un presupuesto de hardware de solo 14.5 KB frente al mejor predictor del estado del arte con 19 KB de presupuesto, PHAST obtiene mejoras promedio en el rendimiento del 1.29 % (con picos de hasta el 22 %) al reducir en un 62 % los fallos por cada mil instrucciones y acorta la brecha en el rendimiento con un predictor ideal a tan solo un 1.5 %.
La contribución final se centra en la predicción de fusión de instrucciones. Tal y como se mencionó anteriormente, solo hay una propuesta previa que haya tratado la fusión de instrucciones de memoria no consecutivas, Helios [86]. En Helios, se denomina *cabeza del núcleo* a la instrucción más antigua involucrada en la fusión y *cola del núcleo* a la más joven. El conjunto de instrucciones que yacen entre ambos núcleos se denomina *catalizador*.
Tras decodificar las instrucciones, las operaciones de memoria consultan un *predictor de fusión* que indica la distancia, en instrucciones, donde se encuentra la cabeza del núcleo. Si dicha instrucción aún no se ha insertada en la estación de reserva, entonces se sustituye por la fusión de ambos núcleos y debe esperar

a que la cola valide la fusión. Para ello, la cola del núcleo se mantiene en el cauce de instrucciones hasta que se envía a la estación de reserva. Si durante el renombramiento de registros la cola detectó dependencias con alguna instrucción dentro del catalizador, debe subsanar la instrucción fusionada para que utilice los registros correctos. Asimismo, si detectó algún caso de *deadlock*, en donde se produce una situación imposible de solucionar debido a que la instrucción fusionada debe esperar a haberse ejecutado para poder ejecutarse, entonces la fusión se deshace.

Para entrenar el predictor, las operaciones de memoria no fusionadas acceden a una estructura conocida como *Unfused Committed History* (UCH), la cual contiene las direcciones de memoria a las que han accedido los *loads* y *stores* previos no fusionados, así como un identificador que se asigna en esta etapa. Así, las operaciones de memoria que no se fusionaron buscan en esta estructura una coincidencia en la dirección de memoria accedida. En caso de encontrarla, calculan la distancia con la diferencia de los identificadores y notifican al predictor.

Si el predictor poseía una entrada válida para esta instrucción, entonces incrementará un contador asociado. Sino, creará una entrada nueva y establecerá el contador en uno. A la hora de realizar una predicción, sólo tendrá se tendrá en cuenta aquellas que tengan el contador al valor máximo.

Finalmente, cuando una instrucción fusionada no contigua se ejecuta, si las direcciones de memoria de los núcleos se encuentran demasiado alejados, entonces debe invalidar todo el cauce desde la cola del núcleo. En este caso, también se notifica al predictor para que invalide la entrada relacionada.

Aunque Helios ofrece resultados prometedores, hemos identificado algunas decisiones de diseño que dan lugar a un rendimiento subóptimo. Por ejemplo, su predictor está basado en un predictor de saltos por concurso (Tournament en inglés) con longitudes de historia demasiado largas para fusión de instrucciones.

Para abordar este problema, presentamos *FLIP*, un predictor de fusión de instrucciones diseñado para superar las limitaciones de Helios. Mientras que Helios basa su esquema de predicción en un predictor por concurso, que debe seleccionar entre un predictor con historia local (utiliza únicamente el contador de programa de la instrucción) y otro con historia global (añade 9 saltos previos), FLIP utiliza el mismo principio que PHAST—aprender del contexto de ejecución adecuado—para identificar pares no consecutivos de *loads* o *stores* que pueden fusionarse en una sola operación. Para ello, FLIP utiliza la historia comprendida entre los núcleos de una fusión no consecutiva.

Además, FLIP incorpora un conjunto de Optimizaciones de Fusión No Consecutiva (NFOs) que relajan las restricciones de Helios—como forzar a la cabeza del núcleo a esperar su validación—o mitigan el impacto de casos de fusión perjudiciales. Nuestros resultados experimentales muestran que FLIP mejora el rendimiento de Helios en un 2.44 % en SPEC CPU 2017 y en un 2.94 % en MiBench, al tiempo que reduce los fallos por cada mil instrucciones en un 83 %. En conjunto, PHAST y FLIP muestran el potencial de la predicción sensible al contexto como principio unificador para mejorar las técnicas especulativas en procesadores modernos. Ambas contribuciones demuestran que la especulación puede ser más precisa y eficiente cuando se aprende del contexto real de ejecución de las instrucciones, en lugar de depender únicamente de información estática o de heurísticas simplistas.

*To my family*

# Agradecimientos

El desarrollo de esta tesis doctoral ha representado, sin duda, el mayor reto al que me he enfrentado. El haber pasado más de la mitad de esta tesis recibiendo comentarios como "¿Por qué trabajas en un tema muerto como es la predicción de dependencias de memoria?" y enfrentando las duras críticas de algunos revisores de PHAST fue desgastante, e hicieron que quisiera retirarme en muchas ocasiones. Por ello, son muchas las personas a las que debo expresar mi gratitud, y sin las cuales habría sido imposible llegar a este punto.

En primer lugar, empezaré por mi familia, y en concreto con María, mi pareja, por su apoyo incondicional durante los 17 años de relación que llevamos. Han sido muchas las ocasiones en las que este trabajo me sobrepasó o en que yo perdía la fe en lo que hacía, y su cariño y confianza me animaron a seguir adelante.

Quiero agradecer también a Sofía, mi hija, nacida en medio de esta aventura académica. Aunque compaginar las responsabilidades de padre primerizo con el trabajo resultó agotador, tenerla en mi vida es de lo que más orgulloso me siento.

Extiendo mi agradecimiento a toda mi familia, tanto biológica como política, por haber estado siempre ahí, por confiar en que lograría terminar y por las horas de niñeras cerca de los plazos de entrega y durante la redacción de esta tesis.

En segundo lugar, y no por ello menos importante, quiero expresar mi más

sincero agradecimiento a mi director de tesis, Alberto, sin el cual no habría seguido el camino de la arquitectura de computadores. Su gran orientación, comprensión y apoyo, no siempre presentes en los superiores, marcaron una diferencia fundamental. Todavía recuerdo cuando, finalizando el primer año de doctorado, lo contacté para comunicarle que no me sentía a la altura de este desafío y que estaba pensando en abandonar, y acto seguido me llamó para darme ánimos y ayudarme a replantear el enfoque de esta tesis.

En tercer lugar, quiero agradecer a mis amigos y compañeros de trabajo por las pausas de café, las risas y los planes. En especial, quiero darle las gracias a Víctor (*Nikitin*), Sawan, Eduardo, Ashkan y Agustín, con quienes compartí prácticamente la totalidad de mi tiempo en el doctorado. Voy a echar de menos las comidas y cenas de navidad o los asados que compartimos.

También quiero agradecer especialmente a Juanma y Ricardo, quienes siempre se prestaron a resolver cualquier duda o problema que pudiera surgir con respecto a las herramientas de trabajo o a la docencia a impartir. Finalmente, quiero destacar la labor de Eduardo en el mantenimiento de nuestro servidor, cuya dedicación nos salvó de muchos problemas a lo largo de estos años.

De igual manera, quiero agradecer a la Universidad de Murcia y a la Escuela Internacional de Doctorado por la oportunidad de llevar a cabo esta investigación y de la experiencia que he vivido.

Finalmente, quiero agradecer a la fundación Séneca (Agencia de Ciencia y Tecnología de la Región de Murcia) por su apoyo económico mediante una Beca-Contrato Predoctoral de Formación de Personal Investigador (FPI). En especial, quiero darle las gracias a Viviane Barelli por su paciencia y por toda la ayuda en la gestión, documentación y dudas varias que me surgieron a lo largo de estos años.

*A todos ustedes, muchísimas gracias por hacer posible esta tesis.*

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

Processors have become an essential part of everyday life. From smartphones and laptops to large-scale datacenter infrastructures, billions of users depend on them to execute programs efficiently and reliably. To meet these demands, computer architects continually seek new ways to improve both performance and energy efficiency.

Historically, performance improvements were achieved by increasing processor frequency or shrinking transistor sizes to fit more on a chip. However, frequency scaling has reached physical limits, and increasing transistor counts requires more sophisticated ways to organize and use these resources effectively.

Beyond raw hardware scaling, performance can also be improved by tailoring processor design to meet diverse computational needs more efficiently. Two widely adopted techniques are *out-of-order* (OoO) execution and *speculate* on the outcomes of some events, which exploit instruction-level parallelism and hide latency.

OoO execution allows instructions to execute as soon as their operands are available, rather than waiting for the previous instruction to complete. This

improves resource utilization and overlaps long-latency operations. To preserve correctness, however, instructions must still retire in order. This is ensured by the reorder buffer (ROB). Memory instructions require additional support: loads allocate entries in the load queue (LQ), while stores allocate entries in the store queue (SQ). Stores only update memory after retirement, which requires an auxiliary structure called the store buffer (SB). In modern implementations, SQ and SB are typically combined into a single, logically partitioned structure.

Speculation allows the processor to make educated guesses about the outcomes of unresolved events and continue execution without waiting. By doing so, it keeps pipelines full and avoids costly stalls. When the guess is correct, speculation hides latency and boosts performance. When it is wrong, however, the processor must discard all dependent work and restart from the last known correct state—a process known as a squash. This wastes time and energy, and frequent mis-speculation can reduce performance rather than improve it. Speculation also adds hardware complexity and has been shown to expose security vulnerabilities [46, 55]. Thus, speculation is a powerful but delicate tool that must be carefully designed to balance benefits against risks.

Speculation is guided by predictors, hardware structures that provide guesses based on program behavior. Predictors use inputs such as the instruction's program counter (PC) and the global history of past outcomes (e.g., whether earlier branches were taken). Accurate predictors allow speculation to be highly effective, while inaccurate ones lead to costly mis-speculation.

This thesis focuses on two speculative mechanisms: memory dependence prediction (MDP) and instruction fusion prediction.

## 1.1 Memory Dependence Prediction

As noted earlier, OoO execution enables instructions to bypass each other once their inputs are ready. Loads have one input (the address) and one output (the retrieved data), whereas stores have two inputs (address and value) but no outputs. Unlike register dependencies, which are explicit in the instruction encoding, memory dependencies are more complex because addresses are not known until runtime [69]. This challenge is known as the *unknown address problem* [66].

To execute correctly, the processor must ensure that each load retrieves data from the most recent store to the same address. If that store's address is unresolved or its data is not yet available, the load cannot safely proceed [66]. The act of verifying whether two memory operations overlap is called *memory disambiguation*, and the younger operation is said to be memory dependent on the older. A memory dependence arises whenever two memory instructions access the same address and at least one is a store. Some cases, such as Write-after-Read (a load followed by a store) or Write-after-Write (two stores), do not create hazards because stores update memory in program order after commit.

The critical case is Read-after-Write (RAW), where a load risks reading stale data if it bypasses an older store before the store has resolved its address. Figure 1.1 illustrates this scenario: in-order execution retrieves the correct value using store-to-load forwarding, but in OoO execution the load may speculatively read from memory before the store's address is known, leading to a violation. Once the store resolves, it detects the error and triggers a squash.

Two naive strategies exist. A pessimistic approach delays all loads until every older store has resolved, but this severely reduces the instruction-level parallelism (ILP). An optimistic approach issues loads immediately, recovering from

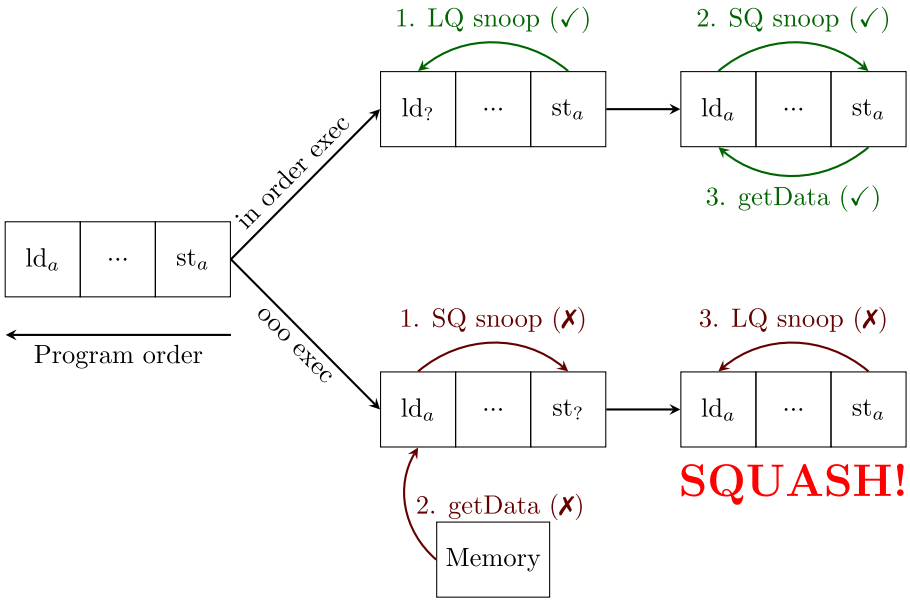



Figure 1.1: Example of memory violation

violations when they occur, but this can lead to frequent squashes and wasted work [12]. Both extremes hurt performance, with the pessimistic approach reducing IPC to as little as one-fifth of blind speculation [69].

A more effective strategy is memory dependence prediction, introduced by Moshovos et al. [60]. MDP predicts whether a load is likely to be dependent on unresolved stores, enabling early execution of safe loads while avoiding costly violations.

Most MDP designs were proposed in the early 2000s, with few recent advances. A notable exception is the work of Perais and Seznec [69,71], who argued that MDP could be absorbed into branch prediction. This gave the impression that the problem was largely solved. However, recent trends suggest otherwise. As shown in Figure 1.2, processor aggressiveness has grown rapidly since Intel Skylake (2015), with exponential increases in queue sizes and execution ports. If this trend continues, inefficient load scheduling may become a critical bottleneck,

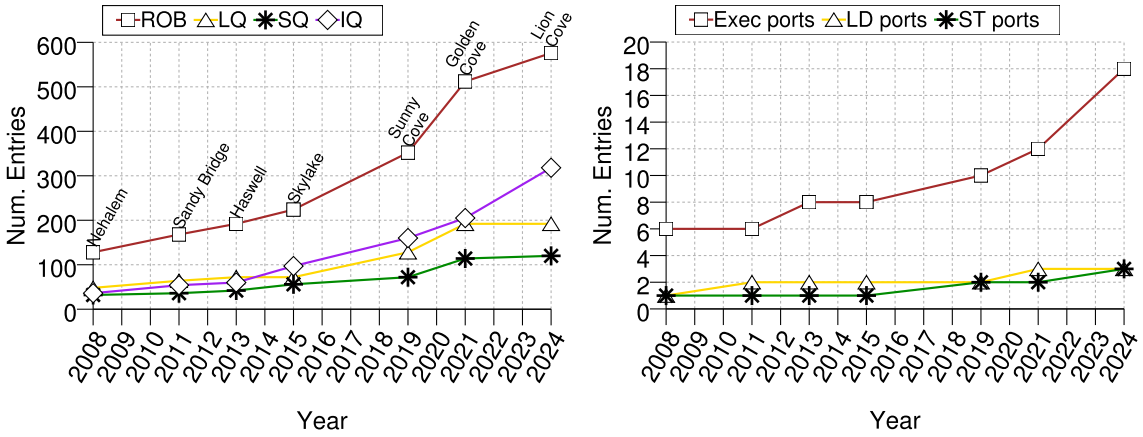



Figure 1.2: Number of entries in several queues (left) and number of execution ports (right) in several Intel processors from 2008 to 2024

as the growing number of in-flight instructions and wider pipeline stages leave loads with more stores whose effective addresses are still unresolved, forcing the loads—and their dependent instructions—to stall longer when a false dependence is predicted.

## 1.2 Instruction Fusion Prediction

Improving processor efficiency is not limited to running instructions faster; it also involves reducing the amount of work performed internally. Instruction fusion is one such optimization. By combining multiple simpler instructions into a single, more powerful internal operation, fusion reduces pipeline pressure while preserving program correctness.

There are two common forms of instruction fusion in x86 processors. In *macro-fusion*, two adjacent instructions are combined at the frontend—before or during decode—so that the core treats them as a single macro-instruction. Only a limited set of instruction patterns can be macro-fused, most commonly a flag-setting instruction, such as a comparison or other arithmetic operation, immediately

followed by a conditional branch. In contrast, *micro-fusion* combines multiple logical operations within a single macro-instruction into a single fused $\mu$-op through most of the pipeline, saving bandwidth. Micro-fusion was added in Intel's Pentium M microarchitecture [25], to cope with limited rename and retirement bandwidth by allowing certain instruction components to share entries in backend structures such as the reorder buffer. Although represented as a single fused $\mu$-op for most purposes, it was treated as two $\mu$-ops by the scheduler and sent to two different execution units. This was applied to read-modify—such as add EAX, [MEM]—and memory write operations.

Traditionally, fusion has focused on consecutive instructions in program order. Recently, Singh et al. [86] demonstrated that non-consecutive instruction fusion is possible with the help of speculation, and that focusing on memory instructions (e.g., load-load or store-store pairs) is enough to exploit most of the benefits. Prediction is required because (1) memory addresses are unknown at decode, (2) to expand the scope beyond instructions with identical base registers or contiguous memory accesses, and (3) to know with which instruction to fuse in non-consecutive cases.

## 1.3 Overview of Proposed Solutions

Existing work on both memory dependence and instruction fusion prediction has often borrowed designs from branch prediction. These approaches rely on predetermined history lengths or sets of history lengths where the best option is usually selected in a brute-force manner. While effective in branch prediction, we argue that such methods are not well suited for MDP and instruction fusion prediction.

The key claim of this thesis is that, unlike branch prediction, in memory de-

pendence and instruction fusion the relevant history length can be precisely delimited. In MDP, the required context information is the one between the conflicting store and load. In instruction fusion prediction, it is delimited by the pair of memory operations to be fused.
If the same execution path recurs, the same conflicts or fusion opportunities are likely to recur as well. By training the predictor using solely the proper history length, accuracy improves significantly. Longer histories dilute accuracy by creating redundant paths, while shorter histories suffer from aliasing. The solutions proposed in this thesis exploit this property to design predictors that are both efficient and accurate.

## Thesis Contributions

This thesis investigates speculative techniques for improving memory operations in high-performance OoO processors.

**Research problem 1:** Store Sets [18], introduced in 1998, remain a widely used memory dependence predictor. In Store Sets, each load is associated with a set of stores whose addresses overlap. Stores may belong to only one set, and when two loads share a common dependence their sets are merged.
**Proposed analysis:** We first analyze the limitations of Store Sets, identifying five major challenges: inter-loop dependencies, load-store pair linking, serialization of memory operations, intermittent dependencies, and aliasing. Our results show that intermittent dependencies are the dominant issue, while forcing loads to wait for the youngest store accounts for a smaller fraction of the performance gap.

**Research problem 2:** Memory dependence prediction has seen little progress in the past fifteen years, largely because earlier processors had relatively few memory operations in flight. For instance, Intel Nehalem (2008) executed one load per cycle with a small instruction window, whereas Golden Cove (2021) executes three times as many loads per cycle and holds four times as many in flight. Under these more aggressive conditions, existing predictors are inadequate, and most designs only consider mis-speculations that flush the pipeline.
**Proposed solution:** We propose PHAST, a novel predictor that associates each load with a single store while selecting the appropriate history length for each conflict. This adaptive approach significantly improves accuracy compared to Store Sets (tracks multiple stores per load), NoSQ (tournament-like predictor), and MDP-TAGE (TAGE-based design). PHAST reduces misprediction rate (MPKI) by up to 62% compared to the best existing alternatives, while requiring less storage.

**Research problem 3:** After decoding, many processors crack instructions into multiple micro-operations ($\mu$ops). Instruction fusion performs the inverse optimization by merging multiple $\mu$ops into a single, slightly more complex one. Fusion reduces critical-path latency and resource usage (ROB entries, LQ/SQ slots). While recent proposals support speculative non-consecutive fusion [86], its predictor relies on tournament mechanisms that are suboptimal, typically using nine or more branches, even though most fusion opportunities involve fewer.
**Proposed solution:** We present FLIP, which, like PHAST, adaptively selects the appropriate history length for each fusion pair. FLIP is further enhanced with optimizations for speculative non-consecutive fusion, relaxing constraints and mitigating harmful fusion cases. With only two-thirds the storage of the current state of the art, FLIP achieves average speedups of 2.44% and 2.94% on SPEC CPU 2017 and MiBench, respectively.

This thesis is based on the following publications, where each work addresses one of the research problems outlined above:

1. S. S. Kim and A. Ros, "Caracterizacón de los límites del predictor de dependencias de memoria store sets". Jornadas Sarteco 2022, Alicante, España, 2022, pp. 275–280.
2. S. S. Kim and A. Ros, "Effective context-sensitive memory dependence prediction". International Symposium on High-Performance Computer Architecture (HPCA), Best paper award honorable mention, Edinburgh, Scotland, 2024, pp. 515–527.
3. FLIP is under submission.

In addition, our memory dependence predictor, PHAST, has fostered collaborations with the University of Cambridge, resulting in a second publication at HPCA 2025 [57]—which has not been included as a contribution of this thesis. Furthermore, with Imperial College London, PHAST has been integrated and released into the gem5 simulator [61].

Figure 1.3 summarizes the impact of each contribution across the processor pipeline. Encircled numbers indicate which contribution affects each component. Structures directly optimized are shown in green, optimized pipeline stages in blue, and processor queues that are not directly improved but benefit from better utilization through our predictors in orange.

## Thesis Organization

The remainder of this thesis is structured as follows:

- Chapter 2 introduces the concept of memory disambiguation, describing how it is implemented in widely used open-source simulators in contrast to industrial approaches. It then reviews memory dependence predictors

developed in both academia and industry, and concludes with the current state of the art in instruction fusion.

- Chapter 3 outlines the methodology employed to evaluate the experiments presented in this thesis.
- Chapter 4 addresses the first research problem, analyzing the limitations of the Store Sets memory dependence predictor.
- Chapter 5 presents PHAST, a novel memory dependence predictor that selects the appropriate history length for each conflict and directly associates loads with the specific store expected to forward the data.
- Chapter 6 introduces FLIP, an instruction fusion predictor that extends the principle behind PHAST to enable non-consecutive load and store fusion. This chapter also proposes a set of Non-consecutive Fusion Optimizations designed to relax existing constraints and mitigate the negative effects of harmful fusion cases.
- Chapter 7 summarizes the main contributions of this work and outlines promising directions for future research in the area.

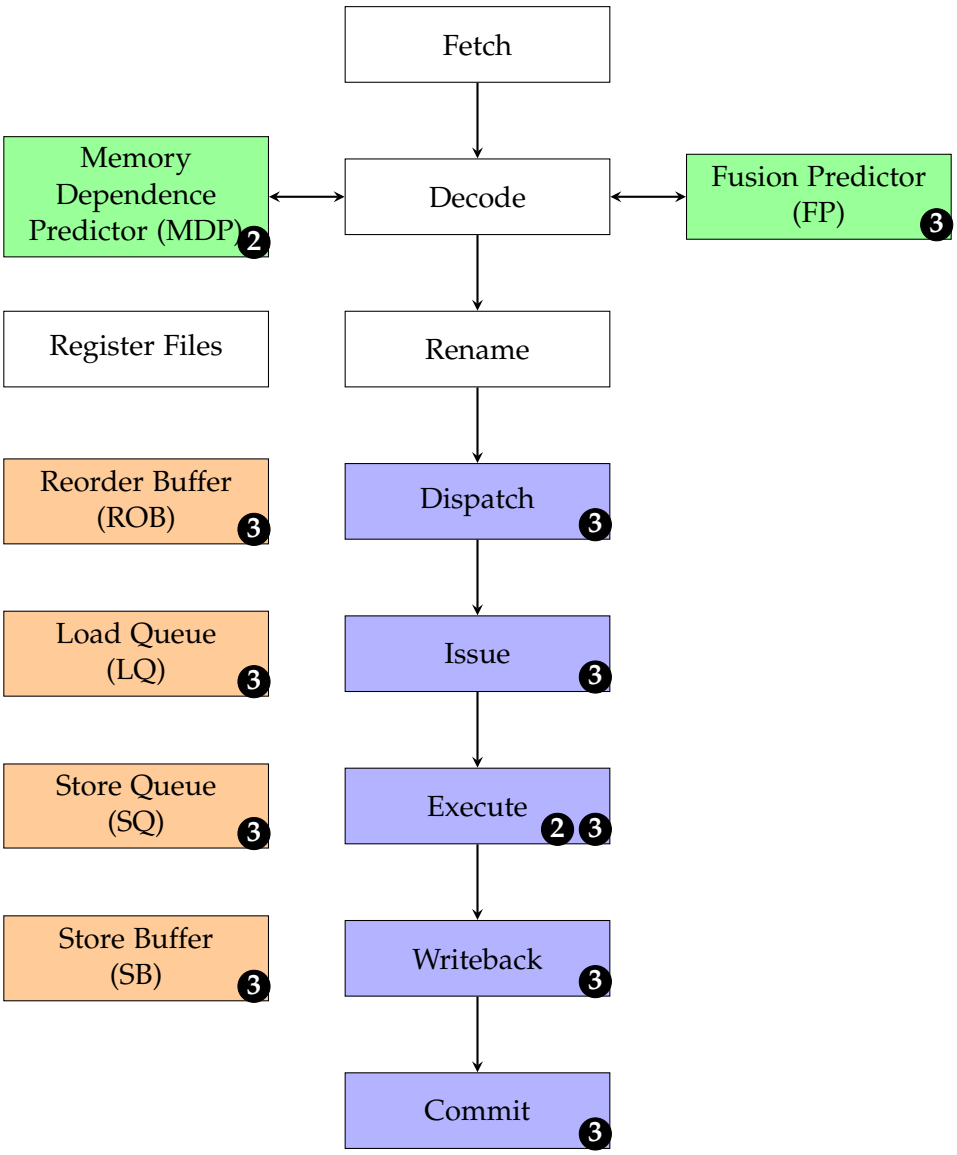



Figure 1.3: Overview of an OoO processor pipeline. Structures optimized in this thesis are shown in green, optimized pipeline stages in blue, and queues that benefit indirectly from improved utilization in orange

Chapter 2

# Background

This chapter introduces the main techniques employed to improve memory communication and mitigate the challenges posed by the unknown address problem. It also provides an overview of how several widely known processor families incorporate these mechanisms.

## 2.1 Memory Disambiguation

Memory disambiguation refers to the process of determining whether two memory operations access the same address, thereby deciding if a load instruction can safely execute before an earlier store. It is fundamental to enabling speculative execution and enhancing performance while maintaining program correctness. The following subsections review several approaches to memory disambiguation, beginning with basic mechanisms and progressing to more advanced implementations.

### 2.1.1 Store-to-Load Forwarding

In modern out-of-order microarchitectures, store-to-load forwarding is a critical mechanism to ensure high performance. It allows a load instruction to receive data directly from an in-flight, older store targeting the same address, without waiting for the store to commit its value to the memory hierarchy [34]. This significantly reduces memory access latency and improves instruction-level parallelism (ILP). Consider the sequence:

```
store [A] ← value
load  R  ← [A]
```

Without forwarding, the load must wait until the store commits and writes back, introducing unnecessary serialization. Store-to-load forwarding eliminates this stall by directly forwarding the data from the store queue once both the store's address and value are available.

Forwarding is typically permitted under the following conditions:

1. The store must be older than the load in program order.
2. The store and the load must access identical byte addresses (same effective address and size).
3. The store's effective address must be fully resolved.
4. The store's data must be ready for use.

When these requirements are satisfied, the load retrieves the value directly from the store queue. Some implementations relax the address-matching requirement to allow inclusions of the load's address within the store's, or even partial overlap. In hardware, the store queue tracks all speculative stores in program order. When a load's address is known, it searches the queue for the youngest matching older store. If the store's data is available, it is forwarded immediately; otherwise, the

load stalls or retries. If the store address has not yet been computed, the store is ignored. If no match is found, the load proceeds to access the cache or main memory.
Store-to-load forwarding is therefore essential to reducing memory stalls and enabling efficient speculative execution. Without it, loads dependent on prior stores would serialize, causing performance bottlenecks in loops and memory-intensive code.

### 2.1.2 Memory disambiguation in processors

Ensuring correctness in store-to-load forwarding is complex. A load must never forward from a younger store or from a store to a different address, which significantly constrains implementations. In byte-addressable ISAs such as x86, forwarding must also handle partial overlaps, requiring fine-grained per-byte tracking in the store queue.
Unlike simulators, which often abstract or idealize behavior, real processors must satisfy strict constraints on timing, power, and area. The following subsections describe specific implementations in commercial processors.
Although academia considered memory disambiguation largely solved by the early 2000s, industry continued to refine predictors throughout the 2010s.

#### 2.1.2.1 Intel

Since the Pentium Pro, Intel has supported out-of-order memory execution. Early designs were conservative: loads could not bypass unresolved stores [22]. To reduce false dependencies, the Intel Core (2006) introduced a memory dependence predictor under the name *smart memory disambiguation* [22]. The Alder Lake whitepaper [76] reports improvements, though details remain undisclosed.

| Store alignment | Store size | Load alignment | Load size | Outcome |
|---|---|---|---|---|
| Natural size | 16 | word | 8,16 | not stall |
| Natural size | 16 | not word | 8 | stall |
| Natural size | 32 | dword | 8,32 | not stall |
| Natural size | 32 | not dword | 8 | stall |
| Natural size | 32 | word | 16 | not stall |
| Natural size | 32 | not word | 16 | stall |
| Natural size | 64 | qword | 8,16,64 | not stall |
| Natural size | 64 | not qword | 8,16 | stall |
| Natural size | 64 | dword | 32 | not stall |
| Natural size | 64 | not dword | 32 | stall |
| Natural size | 128 | dqword | 8,16,128 | not stall |
| Natural size | 128 | not dqword | 8,16 | stall |
| Natural size | 128 | dword | 32 | not stall |
| Natural size | 128 | not dword | 32 | stall |
| Natural size | 128 | qword | 64 | not stall |
| Natural size | 128 | not qword | 64 | stall |
| Unaligned, byte 1 | 32 | Byte 0 of Store | 8,16,32 | not stall |
| Unaligned, byte 1 | 32 | Not byte 0 of Store | 8,16 | stall |
| Unaligned, byte 1 | 64 | Byte 0 of Store | 8,16,32 | not stall |
| Unaligned, byte 1 | 64 | Not byte 0 of Store | 8,16,32 | stall |
| Unaligned, byte 1 | 64 | Byte 0 of Store | 64 | stall |
| Unaligned, byte 7 | 32 | Byte 0 of Store | 8 | not stall |
| Unaligned, byte 7 | 32 | Not byte 0 of Store | 8 | not stall |
| Unaligned, byte 7 | 32,64 | Don't matter | 8,16,32,64 | stall |

Table 2.1: Intel's store-forwarding restrictions

Intel processors implement store-to-load forwarding with two main requirements. Violating such requirements cause stalls proportional to pipeline depth, known as store-forwarding stalls [35]. The first requirement concerns size and alignment. Table 2.1 summarizes these restrictions. Until Golden Cove, partial forwarding was disallowed; since then, it has been permitted under strict offset-matching rules, though multiple partial forwardings remain unsupported [35].

The second requirement is data availability. Intel processors split stores into

two micro-operations: store data (STD) and store address (STA). Addresses are usually computed earlier than data, so sending them to the store queue quickly reduces order violations. If loads are scheduled too aggressively before dependent stores complete, or if STD is delayed, penalties occur [35]. While unpredictable, these penalties are less severe than alignment violations.
Since Ice Lake (Sunny Cove), Intel processors also support speculative memory bypass, marketed as the *fast store forwarding predictor* [35], a process by which the physical register of the STD is copied to the load's output physical register and any consumer of the load.

#### 2.1.2.2 AMD

AMD employs techniques comparable to Intel. The optimization guide for the 15th processor family [20] lists restrictions similar to Intel's, including exact address matching, avoiding narrow-to-wide or wide-to-narrow cases, and restrictions on alignment.
In Zen4, many of these constraints were relaxed. Forwarding still requires the load to be fully contained within the store, but alignment and 64B boundary restrictions have been removed [21].
Since Zen3, AMD has supported speculative memory bypass under the name *Predictive Store Forwarding*. In Zen4, this was further enhanced to allow bypassing from committed stores, branded as *Enhanced Predictive Store Forwarding* [5].
In addition, AMD filled a patent in 2016 describing a mechanism where loads predicted to wait for a store are blocked from accessing the D-cache, while loads predicted as non-conflicting bypass the store buffer [63].

#### 2.1.2.3 Qualcomm

In 2018, Qualcomm proposed a scheme where the branch predictor annotates instruction blocks with disambiguation indicators, specifying whether their loads and stores may issue speculatively. The patent describes seven disambiguation levels [47].

## 2.2 Memory Dependence Prediction

As discussed previously, memory dependence prediction is a key mechanism for enabling efficient out-of-order execution of loads with respect to older stores. The concept was first introduced by Andreas Moshovos in 1997 [58–60]. Predictions can be fine-grained, explicitly linking a load to one or more specific stores, or coarse-grained, indicating that a load may conflict with *any* older store without identifying a precise source.

Like other speculative techniques, memory dependence prediction is subject to mispredictions. A false positive occurs when a load is unnecessarily stalled even though no conflict exists, a case referred to as *false dependence*. Although this scenario does not cause correctness issues, it reduces instruction-level parallelism by delaying the load unnecessarily [96]. Conversely, a false negative arises when the predictor fails to detect a true conflict. In this case, a memory order violation occurs, forcing a pipeline flush and incurring a higher performance penalty.

Yoaz et al. [96] classified prediction outcomes into the following categories:

- Non-speculative: the load has no unresolved older stores at issue time.
- AC-PC: Actually Conflicting - Predicted Conflicting. A true positive, meaning a conflict was correctly identified.
- ANC-PC: Actually Non-Conflicting - Predicted Conflicting. A false positive, or false dependence.

- AC-PNC: Actually Conflicting - Predicted Non-Conflicting. A false negative, which triggers a memory order violation.
- ANC-PNC: Actually Non-Conflicting - Predicted Non-Conflicting. A true negative, where the predictor correctly allows the load to proceed.

Most academic work has concentrated on reducing AC-PNC cases, as these represent costly pipeline flushes. The following subsections summarize the main proposals for memory dependence prediction (MDP) in academia and industry.

## 2.2.1 Memory dependence predictors in academy

### 2.2.1.1 Store-Load Pair

Proposed by Moshovos in 1997 [59], this predictor explicitly associates store-load pairs. It relies on two structures. The first, the *memory dependence prediction table (MDPT)*, records violating pairs by storing the program counters (PCs) of both instructions and the dependence distance, defined as the number of cycles between their fetch times. When a load matches an entry in the MDPT, synchronization with the corresponding store is required. This is enforced using the second structure, the *memory dependence synchronization table (MDST)*, which is either allocated by a store or probed by a load.

### 2.2.1.2 Store Sets

Chrysos and Emer introduced Store Sets in 1998 [18], a predictor that has since become the de facto baseline in the field. Its key idea is to associate each load with a set of stores, all identified by a unique set ID (SSID).

Section 4.1 of Chapter 4 gives a more detailed description of this predictor. At a higher level, the mechanism of Store Sets relies on two structures. The *Store Sets Identification Table (SSIT)* associates each load or store with an SSID. The *Last*

*Fetched Store Table (LFST)* records the most recently fetched store for each set. To simplify table lookups, each store belongs to at most one set. When a store issues, it checks its SSID and, if it was the last store fetched, its LFST entry is cleared. Loads depend on the last store in their set, and all stores in the same set are forced to issue in program order to maintain correctness. To prevent sets from growing excessively, entries are periodically reset.
Önder and Gupta later proposed Enhanced Store Sets [65], which relaxed the in-order restriction on stores within a set. Instead of relying on addresses, dependence detection was based on data values, with violations detected at retirement by comparing loaded values rather than effective addresses.

#### 2.2.1.3 Collision History Table

Proposed by Yoaz et al. in 1999 [96], the Collision History Table (CHT) represents a coarse-grained approach. Rather than linking specific store-load pairs, it predicts whether a load will conflict with *any* inflight store. An extended version also tracks the minimum distance between a load and a conflicting store, restricting waiting to only those stores within that distance.

#### 2.2.1.4 Color Sets

Önder and Gupta introduced Color Sets in 2002 [64]. This mechanism defines multiple speculation levels, or *colors*, which represent increasing aggressiveness in speculative load execution. Loads that never conflicted remain at the base color. Based on memory port utilization and the presence of potentially conflicting stores in the instruction window, the processor is assigned a color, and loads may speculatively issue if their color does not exceed this level. On a violation, the load's color is incremented.

#### 2.2.1.5 NoSQ predictor

Sha et al. [85] proposed a store-load bypassing predictor that maps each dynamic load to the store providing its value. The design consists of two load-indexed set-associative tables. Each entry stores a partial tag, a distance field encoding the store dependence, and an $n$-bit confidence counter.
One table captures path-insensitive loads and is indexed solely with the load PC. The other table captures path-sensitive loads using a hash of the load PC combined with a fixed-length branch history (one bit per conditional branch, two bits of PC per call). When a memory ordering violation is detected, entries are allocated in both tables. During prediction, both tables are accessed, and in case of a conflict, the path-sensitive entry takes precedence.

#### 2.2.1.6 Store Vector

Subramaniam and Loh proposed the Store Vector predictor in 2006 [89], aiming to simplify industrial adoption by avoiding CAM-based designs. Each load is assigned a vector of inflight stores it must wait for, enabling out-of-order issue of stores and allowing a store to belong to multiple sets without requiring merges. While promising in concept, this approach suffers from poor scalability. Vector sizes grow with the store queue. For example, the paper assumes a 32-entry store queue, requiring 5 bytes per load. Modern processors such as Intel's Lunar Lake feature 120-entry store queues, which would require 15 bytes per load, most of which would be unused.

#### 2.2.1.7 Counting Dependence

Roesner et al. [74] proposed Counting Dependence, targeting systems without centralized fetch mechanisms or global store tracking. The predictor dynamically

alternates between aggressive (predict no dependence), conservative (wait for all), and selective (wait for a single store) modes. Instead of predicting specific store-load pairs, it learns the number of events a load must wait for.
Each entry requires only two bits to encode the strategy, along with partial PC bits in some implementations. Loads begin in the aggressive mode and update their state on each execution. If multiple matching stores exist, the load waits until all have issued.

#### 2.2.1.8 MDP-TAGE

Perais and Seznec extended the TAGE branch predictor to also predict memory dependence in 2017 [69], later generalizing it into an omnipredictor in 2018 [71]. In this design, memory dependencies are predicted as store distances using the partially tagged tables. Precise store-load links are possible when the distance is below eight.
This limit arises from the 3-bit counters in the tagged tables, which encode store distances, with the value 111b representing *wait for all*. As with Store Sets, entries require periodic resets. Since TAGE was optimized for branch prediction, its reset interval of 512K cycles was too long for memory dependence. To mitigate this, false dependences trigger resets probabilistically, with a probability of $\frac{1}{256}$.

### 2.2.2 Memory dependence predictors in industry

#### 2.2.2.1 IBM

A 1997 patent by Hesson et al. [30] describes the *Store Barrier Cache*, which records stores responsible for violations by incrementing a saturating counter. When such a store is fetched, the cache forces younger loads to wait. If the store does not cause violations, the counter is decremented.

#### 2.2.2.2 Alpha 21264

The Alpha 21264 introduced the *Load-Wait Table* [41]. Each entry contains a single bit, which is set when a memory order violation occurs. A set bit forces the corresponding load to wait for all older stores. As with Store Sets, entries are reset periodically.

#### 2.2.2.3 Intel

Intel introduced a memory dependence predictor in the Core microarchitecture. It employs a hash table indexed by selected bits of the load's PC. Each entry contains a saturating counter [22]. At dispatch, the predictor is consulted; if the counter is saturated, the load is allowed to issue.
At retirement, loads update the predictor. If no violation occurred, the counter is incremented. If a violation did occur, the counter is reset. To guard against performance degradation, a watchdog mechanism disables memory disambiguation if predictor accuracy drops below a threshold.
The behavior outlined in the whitepaper closely resembles a 2010 Intel patent, later granted in 2013 [48]. Empirical evaluation by Downs [23] confirmed that the client version of Intel Skylake implements a predictor consistent with this design.

#### 2.2.2.4 Apple

Apple has also pursued this line of work. A 2019 patent introduced a load/store dependency predictor trained on whether violations caused flushes or replays. The dependency may be enforced if the Confidence counter is above a threshold, which varies depending on the status of a replay/flush indicator. If a load matches multiple entries, and at least one has the flush indicator, it is forced to wait for all prior stores [40].

More recently, in 2025, Apple patented a speculative memory bypass scheme in which a detected store-load pair directly forwards the physical register of the store to the load, enabling *zero-cycle loads* [10].

## 2.3 Instruction Fusion

The Complex Instruction Set Computers (CISC) paradigm introduced instruction cracking, a process in which a complex instruction is decomposed into several simpler micro-operations ($\mu$op) that can be handled by the processor. Instruction fusion can be viewed as the complementary optimization, in which multiple instructions are combined into a single operation. Fusion typically occurs at the processor front-end and aims to increase throughput while reducing pressure on critical resources such as instruction queues and execution units.

### 2.3.1 Instruction fusion in academy

Recent academic work on instruction fusion has largely concentrated on improving large language models [27,67,95].

Kim and Lipasti [45] proposed macro-op scheduling with the goal of simplifying the Instruction Queue (IQ) logic. Candidate pairs of $\mu$ops are scheduled as a unit, which reduces the frequency of wakeup and select operations, since the IQ schedules fused pairs half as often as individual $\mu$op. While macro-ops occupy only a single IQ slot, thereby saving capacity, they do not execute with reduced latency.

Celio et al. [17] argued that fusion can improve the amount of work performed per $\mu$op in RISC-V designs, rather than expanding the ISA with additional instructions. They identified a set of idioms suitable for fusion, summarized in

| Pattern | Result |
|---|---|
| add rd, rs1, rs2<br>ld rd, 0(rd) | Fused into an indexed load |
| slli rd, rs1, 1,2,3<br>add rd, rd, rs2 | Fused into a load effective address |
| slli rd, rs1, 1,2,3<br>add rd, rd, rs2<br>ld rd, 0(rd) | Three-instruction fused into<br>a load effective address |
| slli rd, rs1, 32<br>srli rd, rd, 32 | Clear upper word |
| lui rd, imm[31:12]<br>addi rd, rd, imm[11:0] | Load upper immediate |
| lui rd, imm[31:12]<br>ld rd, imm[11:0](rd) | Load upper immediate |
| auipc rd, symbol[31:12]<br>l[dw] rd, symbol[11:0](rd) | Load global immediate |
| auipc t, imm20<br>jalr ra, imm12(t) | Fused far jump and link<br>with calculated target address |
| addiw rd, rs1, imm12<br>slli rd, rs1, 32<br>SRLI rd, rs1, 32 | Fused into a single 32-bit<br>zero extending add operation |
| mulh[[S]U] rdh, rs1, rs2<br>mul rdl, rs1, rs2 | Fused into a wide multiply |
| div[U] rdq, rs1, rs2<br>rem[U] rdr, rs1, rs2 | Fused into a wide divide |
| ld rd1, imm(rs1)<br>ld rd2, imm+8(rs1) | Fused into a load-pair |
| ld rd, imm(rs1)<br>add rs1, rs1, 8 | Fused into a post-indexed load |

Table 2.2: Common operations suitable for instruction fusion. Obtained from [17]

Table2.2. Their approach is limited to consecutive instructions and contiguous memory operations.

Building on this work, Singh et al. [86] developed Helios, a mechanism that supports non-consecutive and non-contiguous memory instruction fusion through prediction, thereby broadening the scope of instruction pairs that can be fused.

### 2.3.2 Instruction fusion in industry

Instruction fusion has also been widely adopted in industry. Intel introduced macro-op fusion in the Core family [93], and it has been present in all subsequent designs. After macro-op boundaries are identified, candidate pairs are inserted

into the instruction queue waiting to be decoded, where fusion opportunities are exploited.

The most common Intel fusion case is compare followed by branch, which creates a combined operation-and-branch instruction. The conditions for this type of fusion are:

1. The two instructions must be consecutive.
2. The first instruction must be one of CMP, TEST, ADD, SUB, INC, DEC, or AND.
3. The second instruction must be a conditional jump (e.g., JA, JAE, JE, JNE).
4. The first instruction cannot end at byte 63 of a cache line, and the second cannot begin at byte 0 of the following line.
5. Only one macrofusion is allowed per cycle; additional opportunities are ignored.

Intel also employs $\mu$op-fusion, in which two operations of the same instruction are decoded as a single $\mu$-op while still being sent to two execution units, thereby improving bandwidth. The following cases are fused at the $\mu$op level [35]:

1. All stores to memory, including store immediate, which internally execute as store-address and store-data.
2. All read-modify instructions that combine a load with an operation between register and memory.
3. All load+jump instructions.
4. CMP and TEST instructions with an immediate operand and memory access.

AMD also introduced macro-op fusion for compare-and-branch idioms, beginning with its 15th family of processors [26]. With Zen 3 [25], support was extended to almost any arithmetic or logic instruction followed by a conditional jump. Zen 4 further enhanced fusion by enabling NOP fusion with a preced-

ing integer fast-path instruction (excluding branches) [6, 25]. This reduces the overhead of loop alignment in both the decoder and the retire control unit [6]. Finally, while no instruction fusion prediction scheme has been proposed in industry, Qualcomm has shown interest in non-contiguous memory access and non-consecutive instruction fusion [91]. Their patent describes a mechanism to fuse non-consecutive memory instructions if the following conditions are met:

- Same base register for the memory address.
- The addresses must be adjacent, or at least lie within one cache line and not exceed the maximum bus width.
- The instructions between them must not overwrite the base register, or perform conflicting memory access.

## 2.4 Instruction fusion prediction

With the exception of the patent of Qualcomm [91], prior fusion mechanisms have been highly conservative. They typically restrict fusion to instructions that are consecutive in program order and, for load-load or store-store pairs, to accesses that are contiguous in memory. To address these limitations, Singh et al. [86] introduced Helios in 2022, a scheme that enables fusion of non-consecutive instructions in program order, non-contiguous in their memory access, and pairs with different base registers. The rest of the background will provide a brief summary of the Helios architecture, while a more detailed description can be found in Chapter 6.

### 2.4.1 Helios

Helios focuses exclusively on memory pairs—load-load and store-store—since these were shown to provide most of the performance benefits of an idealized system that aggressively fuses all instruction pairs proposed by Celio et al. [17], plus the additional gains from non-consecutive and non-contiguous memory pairs. Drawing on terminology from nuclear fusion, they named *nuclei* to the two instructions involved in the fusion, where the *head nucleus* is the older instruction, and the *tail nucleus* is the younger. All the instructions in between the head and tail nuclei are called *catalyst*. Helios relies on three key mechanisms.

#### 2.4.1.1 Prediction for non-consecutive fusion

Because snooping the entire allocation queue would be prohibitively expensive, Helios employs prediction to identify candidate pairs. Prediction also enables recognition of pairs with different base registers or non-contiguous memory accesses. The fusion predictor, placed at decode, stores the distance—in number of instructions—between the *head* and *tail nuclei*, along with a saturating counter for that distance. At decode, each memory instruction is assigned a predicted distance to its potential *head nucleus*. Non-consecutive fusion occurs if:

- the saturating counter for the predicted distance has reached its maximum value.
- the *head* and *tail nuclei* are of the same type (both loads or both stores) and the *head* is not already fused.
- the *head nucleus* is still resident in the allocation queue or belongs to the same decode group as the tail nucleus.

The predictor is trained at commit using a small structure that records the effective addresses of non-fused loads and stores. Before committing, a memory

instruction checks this structure for a matching address. If a match exists, the entry is invalidated and the predictor is updated; otherwise, the new address is inserted.

#### 2.4.1.2 Dependency validation

Since non-consecutive fusion is speculative, Helios requires a mechanism to detect and resolve dependencies between the *tail nucleus* and any catalyst instructions, as well as other situations that prevent fusion. To simplify recovery, Helios delays issuing a non-consecutive fused instruction until the *tail nucleus* validates it. If a dependency is detected, the fused instruction is corrected to reference the appropriate source register. If a fusion-preventing condition arises—for example, a dependency between the *head* and *tail nuclei*—the fusion is undone and the *tail nucleus* is dispatched independently.

#### 2.4.1.3 Handling mispredictions and exceptions

Helios also handles cases where instructions are incorrectly fused since the memory addresses spans for more than a cacheline, or where mispredictions or exceptions occur within the catalyst region. In such scenarios, the fused pair is unfused, and a squash is triggered from the faulting instruction.

Chapter 3

# Methodology

This chapter summarizes the methodology used in this thesis. First, we give a general description of the simulation environment, followed by the benchmarks used, as well as relevant metrics. Lastly, we present the microarchitectural models used in each experiment.

## 3.1 Memory disambiguation in simulators

This section reviews how open-source simulators implement memory disambiguation.

### 3.1.1 gem5

gem5 [51] is a discrete event-driven, cycle-level, open-source computer architecture simulator that supports flexible modeling of modern processors. It enables both single- and multi-core simulation across a variety of ISAs and full-system modes.

In gem5, loads may issue before unresolved older stores. Its only built-in memory dependence predictor is Store Sets [18], and no straightforward interface is provided for adding alternatives. Loads that trigger violations are squashed and re-executed. Store-to-load forwarding is modeled in an almost idealized manner: fully covered loads forward in a single cycle, while partial matches are rescheduled with no penalty, as loads snoop the store queue and reissue in one cycle.
To date, gem5 appears to be the only open-source simulator that models both store forwarding and speculative out-of-order load execution.

### 3.1.2 Other simulators

Scarab [1] is an accurate multicore simulator designed to balance precision with ease of use. It does not implement memory dependence predictors; instead, disambiguation is resolved at fetch time (`icache_stage::icache_issue_ops`) using an oracle that tracks prior stores and their addresses.
ChampSim [3], a trace-driven simulator for microarchitecture studies, does not explicitly model memory disambiguation. Similarly, Sniper [16], based on the interval core model and Graphite infrastructure, does not model the load and store queues. Instead, it resolves dependencies by maintaining a queue of previous stores for each load to check.

## 3.2 Simulation environment

Originally, this thesis was thought to be done using gem5 [51], since it was the only open-source simulator to have memory disambiguation. However, several limitations—mainly at the front-end—led to revisit our decision.

Unlike branch prediction, memory dependence prediction is tightly coupled with the core's aggressiveness. Thus, problems in the front-end of the processor leads to unrealistic fewer memory order violations. As a result, we made the decision to migrate to an internal in-house simulator used in the CAPS research group. This simulator already had memory disambiguation and store-to-load forwarding implemented, as well as a simple Store Sets memory dependence predictor which I implemented as an undergraduate. In addition, this simulator had already implemented the non-consecutive instruction fusion mechanism of Helios.

This in-house simulator models a nine-stage pipeline with decoupled front-end. The simulator can be configured to utilize Sniper [16] for X86 and Spike [87] for RISC-V to provide instructions to the processor model. The memory hierarchy and cache coherence is modeled with GEMS [53], while the interconnection is modeled with GARNET [9].

While the thesis is focused only on single thread performance, the simulator offers support to both multi-core and SMT simulation.

The simulator has been actively developed by the research team throughout the duration of this thesis. Section 3.5 will provide the simulation parameters used for each chapter, as well as the most notable changes in the simulator.

## 3.3 Benchmark Suite

To evaluate the proposed mechanisms, we run applications from the SPEC CPU 2017 [88] and MiBench [28] suites.

- **SPEC CPU 2017:** The SPEC CPU 2017 benchmark suite is widely recognized for evaluating processor performance across a broad range of workloads. It includes both rate and speed modes. SPECspeed focuses on comparing

time for a computer to complete single tasks, while SPECrate measures throughput or work per unit of time.

- **MiBench:** To complement SPEC CPU 2017, we employed a subset of applications from the MiBench suite. MiBench targets embedded and resource-constrained environments, with benchmarks drawn from domains such as automotive, telecommunications, consumer devices, and office applications.

In Chapter 4, we ran the Spec CPU 2017 rate suite with reference inputs [88], compiled with GCC 7.3.0 for X86. For each pair of application-input, we fast-forward the first 4 billion instructions, and simulate the next 100 millions. The first 10 millions of simulation are used for warm-up, and the rest for statistic collection.

The evaluation of Chapter 5 has been carried out with the SPEC CPU 2017 [88] rate suite using the reference inputs, compiled with GCC 7.3.0 for X86. Applications that support multiple input sets are relabeled with an incrementing counter, where each counter corresponds to a different input. For every application-input pair, representative execution intervals are selected using Simpoints [72]. Each interval is then simulated for 100 million instructions.

Lastly, for Chapter 6, the evaluation is carried out with the SPEC CPU 2017 (Speed suite, reference inputs) [88][1] and MiBench [28] suites. We skip the Linux kernel boot and setup for all applications. For MiBench, the full application is run. For SPEC, we fast-forward until the start of the ROI and proceed to report results for the next 500M instructions. All binaries where compiled with GCC 13.2 targeting the *RV64GC* ISA with *-O3 -static*. The use of both the Speed suite instead of Rate suite, as well as the addition of MiBench was to evaluate the

[1]Due to a Spike limitation (`https://github.com/riscv-collab/riscv-gcc/issues/175`), we were unable to run application 625.x264_s in full system mode.

instruction fusion predictors with the same benchmarks used in the Helios paper.

## 3.4 Energy estimation

Energy estimation was calculated with CACTI (Computer-Aided Design of Interconnects and Caches Tool) [50], a widely used framework for modeling the power, area, and timing of cache memories and other on-chip structures.
CACTI enables detailed modeling of diverse memory components—including caches, main memories, and interconnects—across different technology nodes and design parameters. It accounts for both dynamic and static power consumption, incorporating factors such as cache size, associativity, line size, and technology scaling.
In this thesis, the energy consumption of the memory dependence predictors presented in Chapter 5 are derived using CACTI with a 7nm process technology [2]. Per-access energy is obtained from CACTI, including costs for reads, writes, and searches. This per-access energy is then multiplied by the total number of accesses to compute the overall energy consumption.

## 3.5 Simulation parameters

This section provides details of the simulation parameters used for each experiment, as well as the benchmarks employed in the evaluations. Each experiment has been carried out mimicking the most recent Intel architecture, hence why the different processor models simulated.

**Chapter 4:** This chapter evaluates the limitations of the Store Sets [18] memory dependence predictor. The study was carried out in an Icelake-like (Sunny Cove)

Table 3.1: Parameters for Chapter 4 (Characterizing the Limits of the Store Sets Memory Dependence Predictor)

| **8-core Icelake-like processor** | |
|---|---|
| Front-end width | 5-wide fetch, decode and rename |
| Branch predictor | LTAGE [80] |
| Back-end width | 10 execution ports, 5-wide commit |
| AQ/ROB/LQ/SB | 140/352/128/72 entries |
| **Memory hierarchy** | |
| L1I (private) | 32KiB, 8-ways, 4-cycle lat., pipelined, 64 MSHRs |
| L1D (private) | 32KiB, 8-ways, 5-cycle lat., pipelined, 64 MSHRs |
| L1D prefetcher | IP-stride with a prefetch degree of 3 |
| L2 (private) | 512KiB, 16-ways, 12-cycle lat., 64 MSHRs |
| L3 (shared) | 1MiB, 16-ways, 36-cycle lat., 64 MSHRs |
| Main memory | 4GiB, 100-cycle lat. |

microarchitecture employing macro-op and $\mu$op fusion. The store queue and store buffer are unified, as done in Intel architectures. The main simulation parameters are summed in Table 3.1.

**Chapter 5:** The evaluation of this chapter, focused on the design of a novel memory dependence predictor, was carried out using an upgraded version of the simulator, in which the most relevant modification was the addition of latency to snoops of the store queue and load queue, and adding a cycle of latency between the store's address generation and the snoop of the load queue.
In this study, the simulated core was based on an Intel Golden Cove [76,90]. As stated above, the energy consumption of the memory dependence predictors are computed with Cacti-P [50] using a 7nm process technology [2] (see Table 5.1). The main system parameters are summed up in Table 3.2.
In addition, the pipeline has 3 ports for load execution and 2 ports for store execution. The load and store queue have 2 and 3 ports for snooping, respectively, and are searched associatively and in parallel with the L1D access, having both

Table 3.2: Parameters for Chapter 5 (Effective Context-Sensitive Memory Dependence Prediction)

| **4-core Golden Cove-like processor** | |
|---|---|
| Front-end width | 6-wide fetch, decode, and rename |
| Branch predictor | TAGE-SC-L [81] |
| Back-end width | 12 execution ports and commit width |
| AQ/ROB/IQ/LQ/SB | 140/512/204/192/114 entries |
| **Memory hierarchy** | |
| L1I (private) | 32KiB, 8-ways, 4-cycle hit lat., pipelined, 64 MSHRs |
| L1D (private) | 48KiB, 12-ways, 5-cycle hit lat., pipelined, 64 MSHRs |
| L1D prefetcher | IP-stride with a prefetch degree of 3 |
| L2 (private) | 1.25MiB, 10-ways, 14-cycle lat., 64 MSHRs |
| L3 (shared) | 3MiB/bank (4 banks), 12-ways, 36-cycle lat., 64 MSHRs |
| Main memory | 4GiB, 100-cycle lat. |

the same latency [25]. Each cycle, up two stores and up to three loads can initiate the search.

Stores wait for both data and address dependencies to issue. We perform eager squash for branch misprediction, but lazy squash for the less frequent memory order violations. Having memory dependences squashes delayed until commit simplifies the design of PHAST and, as shown in the evaluation, the mis-speculations that trigger such squashes are very low in all state-of-the-art MDPs.

**Chapter 6:** For the evaluation of our instruction fusion predictor, the simulator was upgraded to perform the different instruction fusion mechanisms described in [86]. It also features splitting stores in address computation (STA) and data transfer (STD).

The front-end is driven by a modified version of the RISC-V Spike Simulator [87]. Spike runs in full-system mode with a Linux kernel and injects instructions into our out-of-order processor, configured with parameters similar to an Intel Lion

Table 3.3: System configuration for chapter 6 (Dynamic Pathing Speculative Instruction Fusion)

| **8-core Processor** | |
|---|---|
| Front-end width | 10-wide fetch/decode/rename |
| Allocation queue | 192 entries |
| Branch predictor | TAGE-SC-L [81] |
| Back-end width | 18 execution ports, 20-wide commit |
| AQ/ROB/IQ/LQ/SB | 192/576/388/212/126 entries |
| **Memory hierarchy** | |
| L1I (private) | 32KB, 8 ways, 4-cycle hit latency, pipelined, 64 MSHRs |
| L1D (private) | 192KB 12 ways, 5-cycle hit latency, pipelined, 64 MSHRs |
| L1D prefetcher | IP-stride with a prefetch degree of 3 |
| L2 (private) | 3MB, 24 ways, 14-cycle hit latency, 64 MSHRs |
| L3 (shared) | 3MB/bank (8 banks), 24 ways, 36-cycle hit latency, 64 MSHRs |
| Memory | 4GB, 100-cycle access latency |

Cove P-Core microarchitecture [19]. The main system parameters are summed up in Table 3.3.

The pipeline has 3 ports for load execution, 3 ports for store address generation, and 2 ports for store data. The load queue and store buffer are searched associatively and in parallel with the L1D access, incurring the same latency as the L1D [25]. Each cycle, up to 3 memory instructions can initiate the access, as either 1 store and 2 loads, 2 stores and 1 load, or 3 loads. [15, 25]. Our model implements a Total-Store-Order consistency model, being compliant with the RISC-V TSO extension (Ztso).

Chapter 4

# Characterizing the Limits of the Store Sets Memory Dependence Predictor

Store Sets [18] is the most widely used memory dependence prediction mechanism—in academy—and continues to serve as the standard baseline in modern studies. Their simplicity, low hardware overhead, and effectiveness in enforcing store-load ordering have made them the default choice in academic research. However, despite their widespread adoption, Store Sets exhibit several limitations that can hinder performance and scalability, especially in the context of complex memory access patterns. This chapter presents a characterization of these limitations, highlighting scenarios where this predictor falls short. The goal is to motivate the need for more adaptive and accurate prediction techniques, as explored in subsequent chapters.

The main contributions of this chapters are:

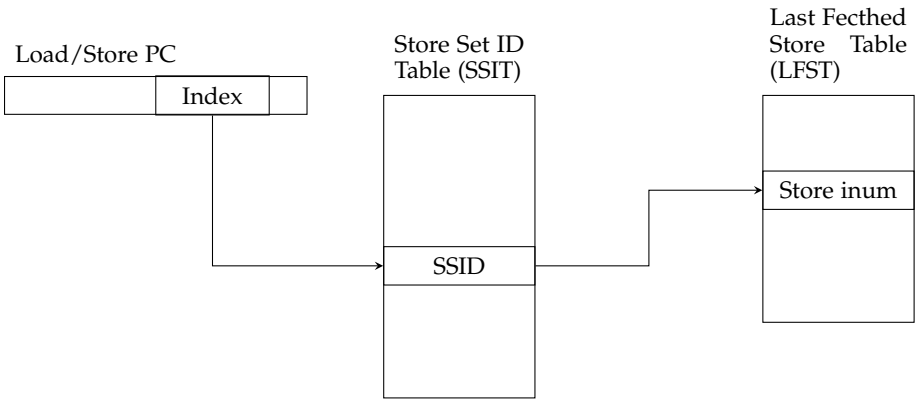



Figure 4.1: Structure of Store Sets

- Describing the main limitations of this predictor and analyze their impact on performance through cycle-level simulation.
- Showing that store-to-load forwarding information can be used to ignore memory order violations when they trigger with an store older than the one forwarding the data.

## 4.1 Background

The Store Sets memory dependence predictor [18] is based on the idea that each load is associated with a set of stores with which it has previously conflicted. Each set is identified by a unique Store Set identifier (SSID) which is created upon a memory order violation and assigned to both the load and the store.
This predictor is implemented using two tagless tables, as shown in Figure 4.1. The first table, called the Store Set ID Table (SSIT), holds the SSID and a valid bit for each set. This table is accessed by every decoded load and store using their PC. If the valid bit of the entry is set, then the SSID is retrieved and used as an index into the second table.
The second table, the Last Fetched Store Table (LFST), holds the identification of

the most recently decoded store of each set, along with a valid bit. Both loads and stores that retrieves a valid SSID use this identification to establish a dependence on that last store.

When a load or store is decoded, it first accesses the SSIT using a hash of their PC to obtain the SSID if the valid bit associated is set. The SSID is then used as index into the LFST. If the LFST entry contains a valid store id, the querying instruction is made dependent on that store, ensuring that all stores within the same set execute in program order.

Stores overwrite the LFST entry with its own identification once the lookup has been done. When a store issues, it accesses again the LFST and invalidates the entry if it still references itself.

Initially, loads are allowed to execute as soon as their operands are ready. When a memory order violation id detected, Store Sets uses the PC of the load and the store to learn the conflict, and add the store to the load's set. To do so, they must share the same SSID, which is done based on the following rules:

- **Case 1:** Neither the load nor the store has a valid SSID.
- **Case 2:** The load has a valid SSID while the store does not.
- **Case 3:** The store has a valid SSID while the load does not.
- **Case 4:** Both instructions have a valid SSID.

In Case 1, where neither instruction has an active SSID, a new one is created. Chrysos et Emer decided to generate the SSID using an exclusive-or hash of the load's PC. The new SSID is then assigned to both memory instructions.

Instead, if the load already has an active SSID assigned, then the store inherits it and becomes part of the load's store set. This scenario is depicted in Case 2.

However, when multiple loads depend on the same store, problems arise if the store already belongs to a set. Overwriting the store's old SSID would effectively remove it from the previous set and insert it into the new one, but it could then

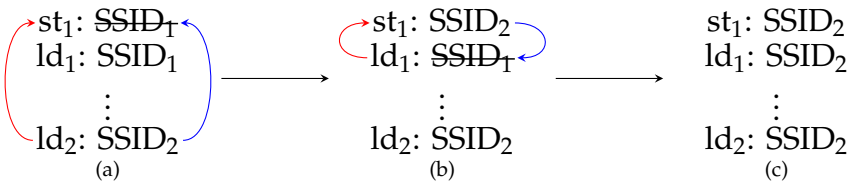



Figure 4.2: Store Sets merging algorithm

potentially trigger a memory order violation with its previous set owner, causing the store to be ping-ponging between the two sets.

A simple solution would be to add multiple sets per instructions—i.e. to have two or more SSIDs per SSIT entry—but this would require additional read and write ports in both the SSIT and LFST structures, and it would also increase the size and complexity of the instruction scheduling hardware that imposes the dependencies.

The authors of Store Sets solved this problem by adding a mechanism known as *merging*. Merging allows a same store set to be shared among multiple loads. In the scenario depicted in the Case 3, since the load does not have a valid SSID, it inherits the store's SSID, effectively sharing the same set between two or more loads.

Lastly, if both the load and the store have a valid SSID (Case 4), then an arbitration algorithm is used to select a *winner*, which is then assigned to the two memory instructions. This algorithm must be consistent—it must always select the same winner for a given pair of SSIDs, such as choosing the smaller value.

Figure 4.2 illustrates this process. Initially, a store ($st_1$) and a load ($ld_1$) share $SSID_1$, while another second load ($ld_2$) has $SSID_2$. In step (a), a conflict between $st_1$ and $ld_2$ causes the store's SSID to be overwritten. Later, in step (b), another violation between $st_1$ and $ld_1$ results in the the load's SSID being overwritten. The resulting scenario is depicted in step (c), where all three memory instructions share the same SSID.

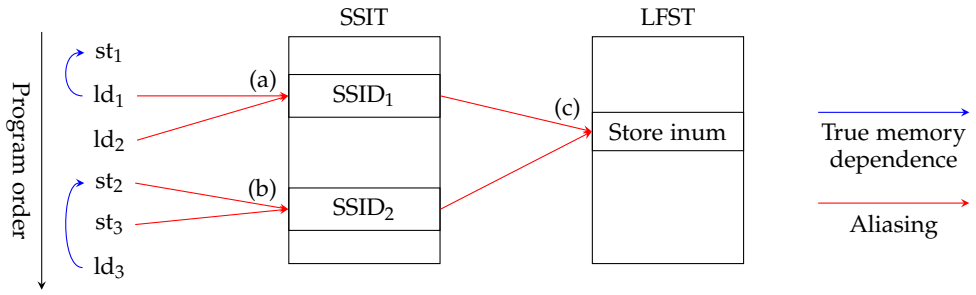



Figure 4.3: Examples of different types of aliasing in Store Sets

To prevent false dependencies, and also to prevent all entries of the SSIT from becoming active, the predictor is reset periodically.

## 4.2 Characterization

The focus of this work is to analyze the Store Sets memory dependence predictor and study its main limitations that hinders the processor's performance. In particular, we focus on five key limitations of this predictor.

**Aliasing:** Store Sets has two tables. The first one—SSIT—maps the PCs of memory instructions to the set ID, the SSID. The second table is used to check which was the last fetched store of that set, and if it is still active. Since the tables present limited sizes, aliasing is bound to happen. When multiple instructions map to the same entry, false dependencies are created. In the first table, the SSIT, unrelated loads and stores may share the same SSID, while aliasing in the second table can lead to loads and stores waiting for an unrelated store. Lastly, since the Store Sets is accessed by both loads and stores, there exists a greater chance of having aliasing, compared to other predictors that focus solely on one type of memory instructions.

Figure 4.3 shows multiple examples of aliasing in Store Sets. In (a), two loads

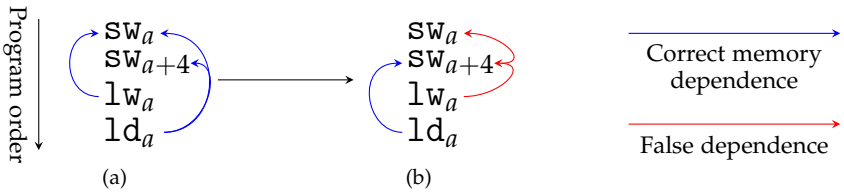



Figure 4.4: The problem of forcing the dependence to the youngest store

happen to collide in the same entry, sharing the $SSID_1$ as if they were merged. A similar situation happens in (b), where two stores collide in a same SSIT entry, and the younger one ($st_3$)— which is not part of the store set—is made dependent on $st_2$, and the load of that set will have to wait for $st_3$ instead. Lastly, aliasing in the LFST is shown in (c), where two different SSIDs fall to the same LFST entry. Similar to (a) this will cause the sets to act as if merged.
Aliasing introduces false dependences that limit parallel execution. Although Store Sets maintain correctness, these false dependences reduce ILP, particularly in workloads with dense or irregular memory access patterns.

**Load-Store pairs:** A fundamental aspect of the Store Sets predictor is that each load is constrained to depend on the youngest store within its set. While this rule ensures memory ordering correctness, it introduces a limitation: loads may be unnecessarily delayed even when older stores are unrelated or do not write to the same address. This behavior can reduce ILP, particularly in loops or code regions with multiple stores to different addresses within the same set. In addition, to maintain each store in a single set, Store Sets merges the sets of loads that share a common store, establishing multiple false dependencies, since each load is made dependent on the stores of both sets.
Figure 4.4 describes this problem. The example shows four instructions, two store words that write to addresses $a$ and $a$+4, followed by a load word and a load double that both reads address $a$. In (a), the real memory dependences are

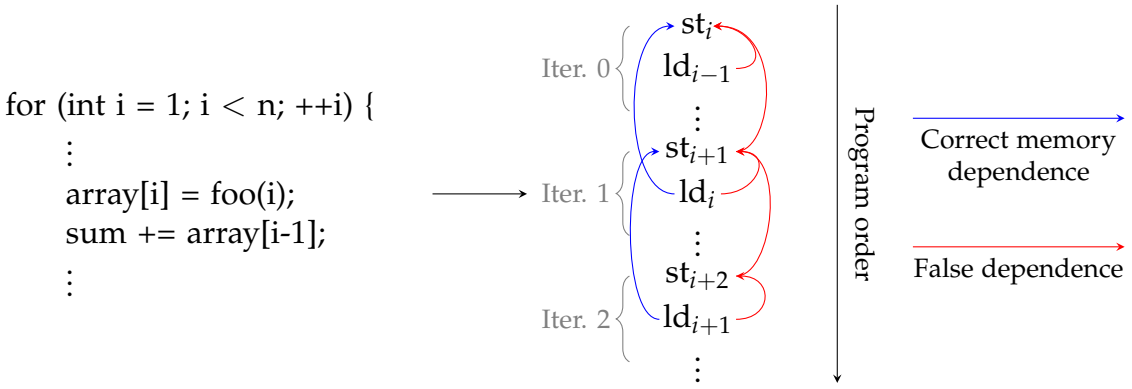



Figure 4.5: Store and loop serialization problems with Store Sets

shown: the load word only conflicts with the first store, while the load double conflicts with both stores. Since both loads share a common store, their sets will be merged. The final scenario is shown in (b), where the load double correctly waits on the second store, but the load word is made dependent on the same store, even though it should only wait for the first store.

By always enforcing dependence on the youngest store, Store Sets minimizes memory-ordering violations at the cost of potential over-serialization. This limitation highlights the conservative nature of the predictor, as it favors safety over performance. Understanding this effect motivates designs that can dynamically identify truly relevant stores, reducing unnecessary serialization in performance-critical regions.

**Store serialization:** In addition to load limitations, another problem derived from the previous point is that Store Sets introduces serialization among stores. When multiple stores belong to the same set, the predictor enforces that stores execute in program order relative to one another. While this reduces OoO ordering between the load and the stores, it can unnecessarily delay stores that do not actually conflict, particularly in sets that have grown large due to merges, aliasing,

or occasional dependencies.
Figure 4.5 depicts an scenario where a load conflicts with a store from the previous iteration. However, since with each iteration the LFST is updated with the new instance of that store, all instances end up added to the set and forced to execute in order.
Store serialization within a set minimizes memory-ordering violations but reduces parallelism when multiple stores do not truly conflict. This limitation is particularly significant in workloads with frequent updates to independent addresses.

**Loop dependencies:** Store Sets enforce memory ordering by making each load depend on the youngest store in its set. While this reduces memory-ordering violations, it can create significant cross-iteration serialization in loops. Consider a conflict between a store in iteration i and a load in iteration i+8: because the load always waits for the youngest store in the set, it may have to wait for stores up to iterations i+7 or i+8, even if only the original store at iteration i actually conflicts. As a result, all subsequent instances of that store become serialized, delaying otherwise independent loads and stores in future iterations.
Figure 4.5 depicts this situation, where a load conflicts with a store from the previous iteration. Due to the conservative nature of Store Sets, the load is made dependent on the store of the same iteration and, at the same time, that store is made dependent on the store at the previous iteration, as seen in the previous limitation. The end result is that all loads and stores must execute in-order.
Memory order violations in loops can propagate far beyond the immediate conflict, limiting parallel execution across multiple iterations. This behavior reveals a conservative aspect of Store Sets: while memory-ordering violations are minimized, unnecessary serialization may occur across iterations, motivating

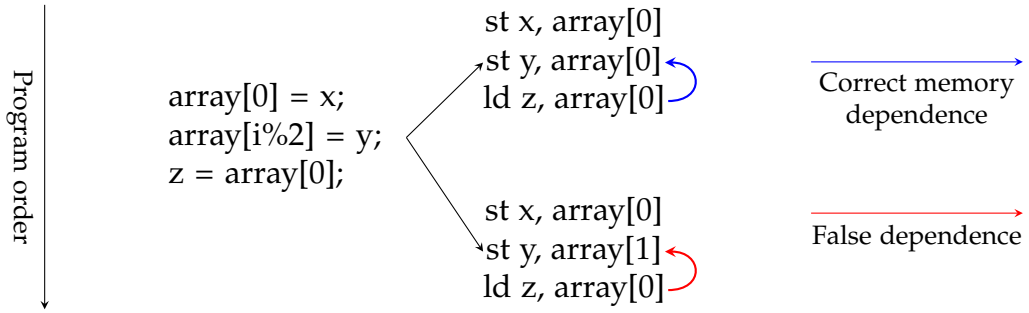



Figure 4.6: Example of occasional dependencies in Store Sets

predictors that can track relevant conflicts more precisely and allow independent iterations to execute in parallel.

**Occasional dependencies:** A final limitation of the Store Sets predictor arises in scenarios where loads and stores only conflict sporadically. Because Store Sets rely solely on instruction PCs to assign the SSID, any observed conflict—even if it occurs only once every N executions—will cause the predictor to enforce a dependence for all future instances of the same load and store pair. This conservative behavior can unnecessarily serialize instructions that would otherwise execute safely in most iterations.

Figure 4.6 illustrates this situation. The second store shown in the code on the left may access two different addresses depending on the iteration. However, once the store becomes part of the set of the load, the load will be made dependent on the second store in every iteration.

By predicting dependencies solely based on instruction PCs, Store Sets may enforce unnecessary serialization for loads and stores that rarely conflict. This limitation highlights the conservative nature of the predictor and motivates mechanisms that can dynamically adapt to conflict frequency, reducing over-serialization while preserving memory correctness.

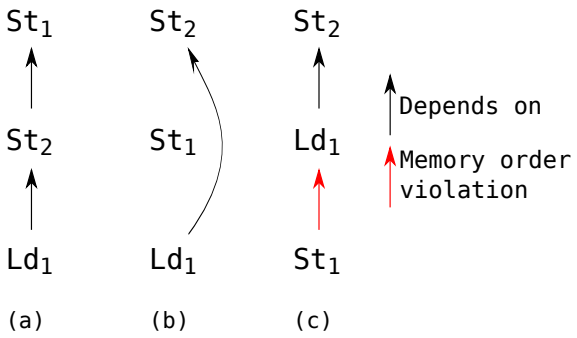



Figure 4.7: Problem of out-of-order execution within the same set. In (a) the dependence chain of two stores and a load assigned to the same set is shown. In (b) a reordering is shown where $St_2$ executes before $St_1$ without issues. In (c) the case is shown where the load also executes before $St_1$, which would generate a violation if the effective addresses of $St_1$ and the load match.

## 4.3 Filtering violations through store-to-load forwarding

The store set associated with a load contains all the previous stores with which that load had previously conflicted. However, since the LFST only records the id of the most recently decoded store within a set, the load depends only on that last store. This could lead to additional violations if the stores were allowed to execute out-of-order, as depicted in Figure 4.7.

Figure 4.7 describes different execution orderings between a load and two stores that belong to the load's store set. In (a), the instructions execute in program order. If such ordering is relaxed between the stores, we could find scenarios (b) and (c). In (b)—where $St_2$ executed before $St_1$—the reordering poses no problem, as the load is still executed the last. However, in (c), where both $St_2$ and the load bypass the first store, a memory order violation is triggered.

The last scenario motivated the design choice of Store Sets to enforce in-order execution among stores within the same set. Nevertheless, if the load obtains the

data via store-to-load forwarding from an older store, the data from stores older than the forwarding one is generally outdated and will not be used by the load. Based on this observation, we propose filtering memory order violations when the conflicting store is older than the forwarding store. The purpose of this filter is to relax the strict in-order execution among stores from a same set.

## 4.4 Incremental Enhancements to Store Sets

The characterization presented above highlights several fundamental limitations of the traditional Store Sets predictor, including aliasing, conservative load-store pairing, store serialization, loop dependencies, and occasional dependencies. Each of these limitations constrains instruction-level parallelism in different ways.

Building on this analysis, the subsequent work in this thesis explores incremental improvements toward an increasingly ideal predictor. Starting from the baseline Store Sets, we systematically address each limitation: first eliminating aliasing, then relaxing the youngest-store constraint, mitigating store serialization, enabling finer-grained loop-dependence handling, and finally adapting to occasional conflicts. This stepwise approach allows us to isolate the performance impact of each refinement, with an oracle memory dependence predictor as the ultimate goal. Each of the Store Sets implementations are:

- **StoreSet:** This is the base implementation of the Store Sets predictor described by Chrysos et Emer [18].
- **Variant for aliasing (InfiniteStoreSet):** A Store Sets with infinite entries in both tables. It uses the full PC to both index the first table and generate the SSID.
- **Variant for relaxing load-store pairs (SeveralStoreSet):** The LFST table is

upgraded to hold information of various stores with different PC. For each different PC, the load is made dependent with the youngest instance of the corresponding store.

- **Variant for loop dependencies (LoopStoreSet):** The LFST stores the PC of the store, and it is used to retrieve the instance whose effective address matches the load's, or the youngest if none is found.
- **Combining variants:**
  - **InfiniteSeveralStoreSet:** Combination of InfiniteStoreSet and SeveralStoreSet.
  - **InfiniteSeveralLoopStoreSet:** Combination of the previous and LoopStoreSet.
- **Perfect:** An oracle memory dependence predictor.

To analyze the impact of relaxing the serialization of stores, variants with the *OoO* label have been implemented, which does not force an order in stores of the same set. Similarly, we study the impact of not squashing a load if it received the value from a store that is older than the load but younger than the conflicting store (labeled *FWD* in Figure 4.8).

## 4.5 Evaluation

In this section, we are going to evaluate the impact of each limitation of the Store Sets memory dependence predictor, using an oracle predictor as baseline. The study is carried out simulating a SunnyCove like processor running the SPEC CPU 2017 rate suite. Refer to chapter 3 for the detailed description of the methodology.

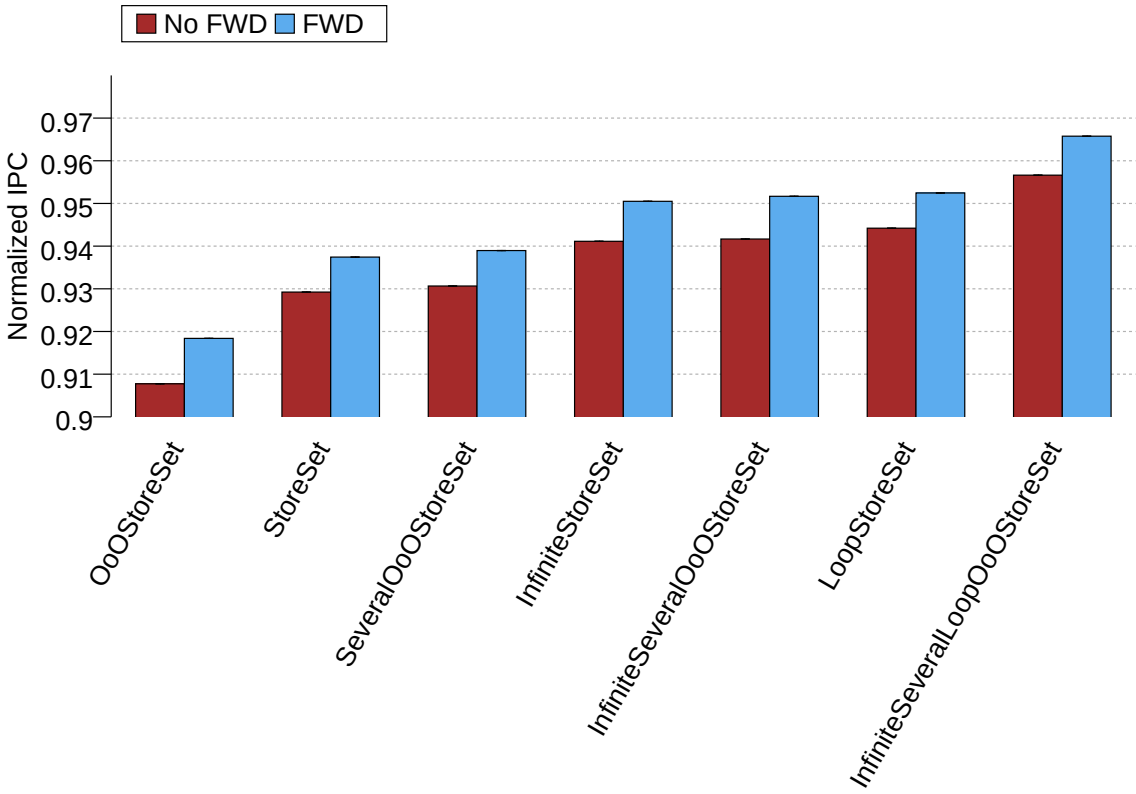



Figure 4.8: IPC of all Store Sets implementations normalized to the ideal predictor. The blue bars (FWD) denote the inclusion of the squash filter

### 4.5.1 The problem of aliasing

To study the impact of aliasing in Store Sets, we modified its tables so that they had an infinite size, thus allowing each load and store to access an entry in the SSIT using its full PC. For the LFST, the SSID was generated using the full PC of the load as identifier, instead of applying a hash function. By completely eliminating aliasing from both tables, Figure 4.8 shows that the performance gap between Store Sets and the ideal predictor decreases from 7% to 6%, meaning aliasing accounts for about one-seventh of this difference.

When analyzing how aliasing in each table affected performance, we observed that the main issue lies in the SSIT, while the LFST showed almost no performance

variations. This is summarized in Figure 4.9, where a rapid increase in IPC is observed as we increase the number of SSIT entries, up to approximately 1024–2048 entries, where its performance becomes similar to that of the infinite table.

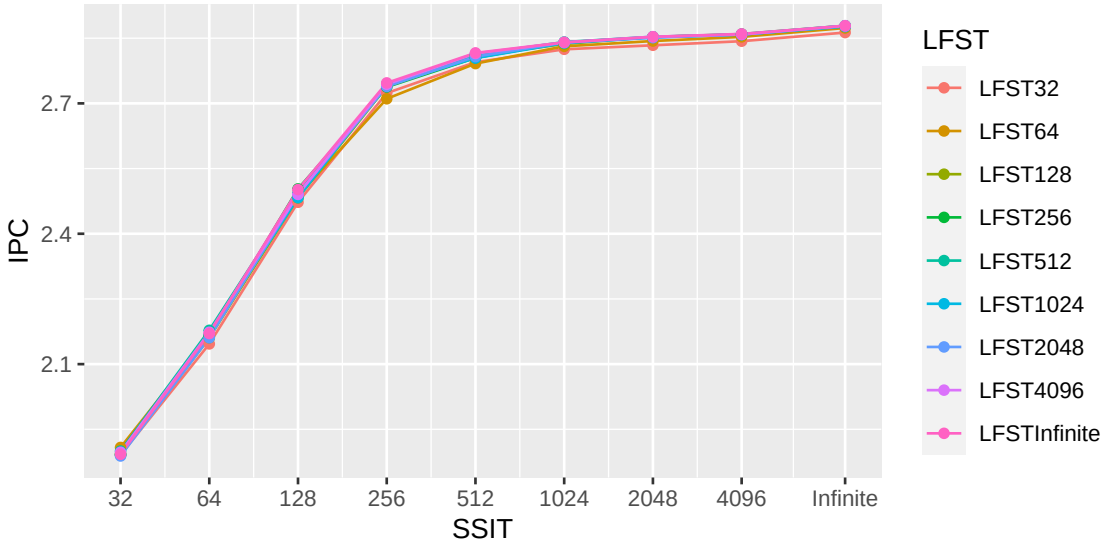



Figure 4.9: IPC evolution as a function of SSIT and LFST table sizes in Store Sets.

The main issue with the SSIT is that it is accessed by all loads and stores, which increases the probability that one of these instructions incorrectly obtains a valid SSID. Furthermore, the index is generated using the least significant n bits of the PC, which facilitates collisions.

In contrast, the LFST is only accessed by those instructions that obtained a valid SSID from the SSIT, meaning that the SSIT acts as a filter for the second table. Additionally, the SSID is generated using a hash function that, in general, should be able to reduce collisions among instructions with similar least significant bits.

### 4.5.2 Store Sets and out-of-order execution

The role of a memory dependence predictor is to enable fully out-of-order execution, ensuring that loads wait only as long as necessary to avoid memory ordering violations. However, as mentioned earlier in this chapter, Store Sets enforces in-order execution of all stores in the same set to avoid performance losses [18].

When enabling out-of-order execution of stores within the same set, we confirmed that IPC indeed decreased because the load waits for the most recent store in its set. However, in presence of more than one store of the same set, if the load executes before any of them, memory violations occur, as shown in Figure 4.7. This ordering relaxation results in a performance degradation, as illustrated in Figure 4.8, where *OoOStoreSet* decreases the performance to below 91% of the ideal predictor.

In the application 503.bwaves we found this problem combined with aliasing (depicted in Figure 4.10):

1. A memory violation is recorded between load $PC_2$ (index i) and store $PC_1$ (index j). Both SSIT entries now contain $SSID_1$.
2. Another violation is recorded between load $PC_5$ (index k) and store $PC_3$ (index j). Since SSIT[j] holds a valid SSID, merging occurs to add the load and store to the same set as before.
3. In the decode stage, the store $PC_4$ (index i) appears right after the previous store. Since SSIT[i] contains a valid SSID, it leaves its identifier in LFST[$SSID_1$].
4. Load $PC_5$ queries the predictor and receives the identifier of store $PC_4$, while its true dependence is with store $PC_3$.

With serialized execution of stores, this aliasing problem merely reduces IPC

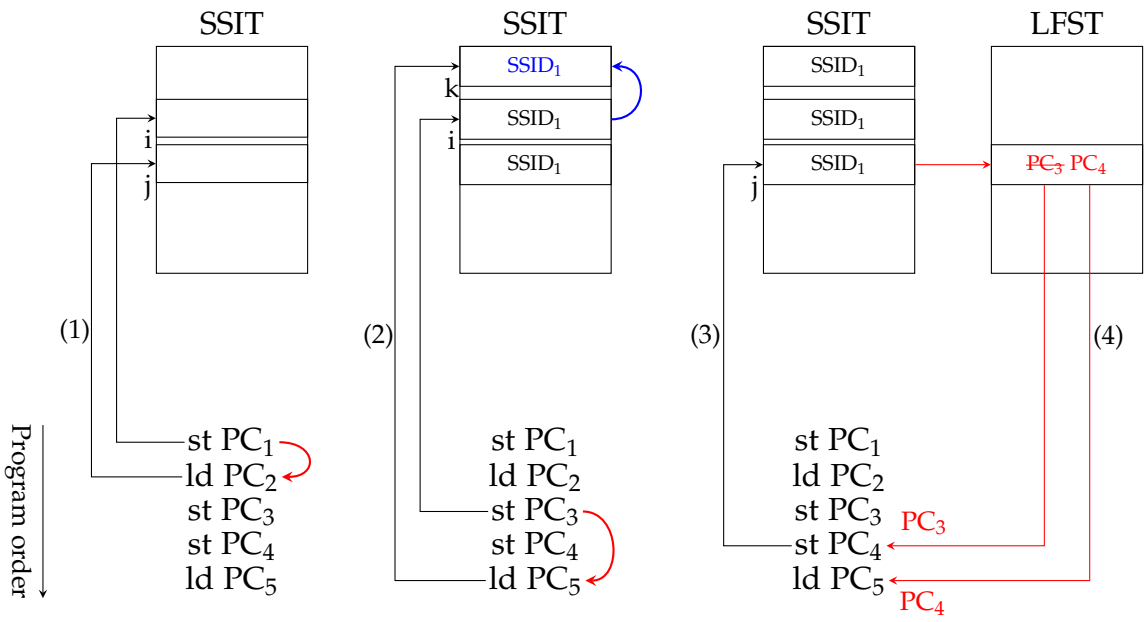



Figure 4.10: Out-of-order execution of stores and aliasing problem in 503.bwaves

slightly. However, when enabling out-of-order execution of stores, store $PC_4$ no longer waits for $PC_3$, and both store $PC_4$ and load $PC_5$ can bypass $PC_3$. When store $PC_3$ finally executes and checks the load queue, it finds that load $PC_5$ executed before it and shares the same effective address, causing a squash. Although different hashes can reduce the chance of such scenarios to happen, the smaller the tables are, the more frequently this problem will happen.

### 4.5.3 Matching loads with multiple stores

Another major issue with Store Sets is that even if many stores are active in the same set, the load is always made dependent only on the most recent one.
To analyze this issue, we modified the LFST to store the identifiers of all active stores in each set and disabled both the *merging* and the store serialization, allowing each store to belong to multiple sets and to execute out-of-order with

```
do {
  uVar1 = *puVar16;
  iVar18 = iVar18 + 2
  lVar14 = (uVar13 >> 6 ^ (uVar13 >> 0x13 | uVar13 << 0x2d) ^ (uVar13
      << 3 | uVar13 >> 0x3d)) + uVar20;
  uVar20 = puVar16[1];
  uVar13 = ((uVar1 >> 1 | (ulong)((uVar1 & 1) != 0) << 0x3f) ^ (uVar1
      >> 8 | uVar1 << 0x38) ^ uVar1 >> 7) + puVar16[8] + lVar14;
  puVar16[0xf] = uVar13;
  W[15] = ((uVar20 >> 8 | uVar20 << 0x38) ^ (uVar20 >> 1  |
      (ulong)((uVar20 & 1 != 0) << 0x3f) ^ uVar20 >> 7) + (W[15] >> 6 ^
      (W[15] << 3 | W[15] >> 0x3d) ^ (W[15] >> 0x13 | W[15] << 0x2d)) +
      puVar16[9] + uVar1;
  puVar[16] = W[15];
  puVar16 = puVar16 + 2;
} while(iVar18 != 0x4e);
```

Figure 4.11: Loop inside the Sha512 function of 500.perlbench.3. Obtained with Ghidra

regard of each other. Nonetheless, the speedup obtained was small (shown in Figure 4.8, *SeveralOoOStoreSet*). This is because most loads with memory dependencies only depend on a single store at a time. Moreover, when two loads unify their sets, they may have real dependencies with more than one store in the combined set, meaning in some cases merging helps reduce the number of squashes.

### 4.5.4 The serialization problem in loops

Another limitation of Store Sets is that the LFST always stores the identifier of the last instance of the conflicting static store. Therefore, if there are dependencies within a loop, the predictor may not be able to correctly relate the load to the actual conflicting store.

We found this issue in applications 500.perlbench.3 and 557.xz. Below we describe the problem observed in 500.perlbench.3. Figure 4.11 shows the code of a loop inside the Sha512 function of 500.perlbench.3 where this phenomenon occurs, and Table 4.1 explains the problem in more detail.

Table 4.1: Reads and writes of variable puVar16 for the first eight iterations of the loop in Figure 4.11.

| Iteration | Reads | Writes |
|---|---|---|
| 1 | 0 1 8 9 | 15 16 |
| 2 | 2 3 10 11 | 17 18 |
| 3 | 4 5 12 13 | 19 20 |
| 4 | 6 7 14 15 | 21 22 |
| 5 | 8 9 16 17 | 23 24 |
| 6 | 10 11 18 19 | 25 26 |
| 7 | 12 13 20 21 | 27 28 |
| 8 | 14 15 22 23 | 29 30 |

If we analyze the reads and writes of variable `puVar16` (first row of Table 4.1), at first sight no conflicts appear. However, when breaking down the read/write positions by iteration, conflicts start appearing from the fourth iteration onward. Table 4.1 shows the reads and writes of `puVar16` for the first eight iterations.
In the code, variable `puVar16` has four reads and two writes per iteration. Table 4.1 shows that from the fourth iteration onward, the fourth read collides with the first write of the first iteration. From the fifth iteration onward, the third read also starts colliding with the second write of the first iteration.
By modifying the predictor to return the instance of the store that the load truly depends on, the difference from the ideal drops from 7% to 5.5%, indicating that these false dependencies account for about one-fifth of this difference.

### 4.5.5 Store Sets and occasional dependencies

When combining all the previous implementations (*InfiniteSeveralLoopOoOStoreSet*), there is still a 4% IPC gap between Store Sets and the ideal predictor. Logically, part of this difference is due to cold misses, which occur the first time a load triggers a memory ordering violation and which repeat when the predictor

is reset.

However, the remaining performance gap is due to occasional dependencies. These occur when the load intermittently depends on the store. Since Store Sets only relies on the instruction PC, it cannot take into account the context of each load/store. Jin and Önder quantified that one third of conflicting loads had occasional dependencies [39] and most of them did not had any dependence with another store.

### 4.5.6 Alleviating the out-of-order problem with store-to-load forwarding

In Section 4.5.2 we showed that out-of-order execution of stores in the same set has worse performance than Store Sets. However, we also observed that the violation shown in Figure 4.7c could be ignored if the effective addresses of the load and $St_2$ match and store-to-load forwarding occurs, since the load would use this value and ignore the one from $St_1$.

In the results shown in Figure 4.8, we can see that this improvement—FWD version—increases IPC by up to 1% in most Store Sets variants, especially in cases where out-of-order execution is enabled within sets. The benefits do not come only from allowing out-of-order issuing of stores belonging to the same set, but also from preventing squashes between a load and a store that does not belong to the load's set.

## 4.6 Conclusion

The main goal of this work was to characterize the limitations of the Store Sets predictor and to quantify the degree of performance degradation caused by

each limitation relative to an ideal predictor. We have identified five limitations: aliasing, collapsing a dependence from a load to multiple stores into a single dependence on the youngest store in the set, the serialization of loads and stores as a consequence of the former, loop-carried dependencies, and occasional dependencies. In addition, we propose avoiding false mispredictions when the load already received the value (via store-forwarding) from a younger store than the one with which it generated the violation.

For the Sunny Cove configuration, Store Sets falls short of the ideal by 7% in performance. Among the five limitations analyzed, the existence of multiple dependencies does not affect performance. In contrast, false dependencies that arise when a store has multiple instances in the instruction window and the load does not depend on the latest instance account for approximately one-fifth of the gap to the ideal, while aliasing accounts for about one-seventh. By combining all the above with out-of-order execution of stores, the gap to the ideal is reduced to nearly 4%, part of which is attributed to cold-start misses of the predictor that occur periodically when it is reset, and occasional dependencies. Finally, by using forwarding to avoid the squash penalty, IPC is observed to increase by up to 1%.

Chapter 5

# Effective Context-Sensitive Memory Dependence Prediction

While Store Sets has long represented the de facto mechanism for memory dependence prediction (MDP), the limitations analyzed in the previous chapter constrain both accuracy and performance in modern high-performance processors. In particular, their reliance on static instruction correlations and coarse-grained grouping leads to unnecessary serialization and delayed learning in dynamic program contexts. These limitations motivate the design of a new memory dependence predictor that adapts faster to runtime behavior, generalizes across diverse dependence patterns, and minimizes false dependences. This chapter introduces such a design, presenting a novel memory dependence predictor that fundamentally rethinks how it is trained and what information it leverages to capture and enforce dependences.

While branch prediction has received extensive attention over the last decades [37, 43, 49, 54, 56, 78, 79, 81–84, 97], comparatively fewer works have addressed MDP. Notable proposals include Store Sets [18], CHT [96], Store Vectors [89], the predic-

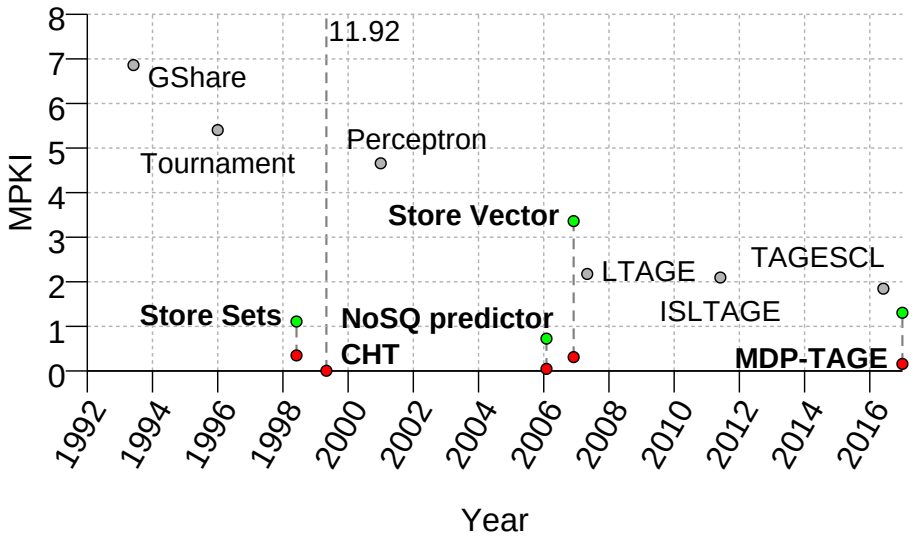



Figure 5.1: Average MPKI for SPEC CPU 2017 of branch (gray) and memory dependence (red-green) predictors proposed over the past 30 years (x axis). For MDP, we show the MPKI reported by a Nehalem-like processor [36], released in 2008

tor in NoSQ [85], and MDP-TAGE [71]. Branch prediction has traditionally been more critical as it directly influences the fetch stage. At the same time, early MDP techniques already achieved lower mispredictions per kilo-instructions (MPKI) than branch predictors in contemporary processors. This trend is illustrated in Figure 5.1, which compares average MPKI for the SPEC CPU 2017 benchmark suite [88], considering both branch predictors (gray circles) and memory dependence predictors (red and green circles) proposed over the past 30 years. For memory dependence predictors, red circles represent MPKI due to memory order violations (false negatives), whereas dotted lines leading to green circles represent MPKI from false positives. False negatives are more severe, as they lead to pipeline flushes, but false positives can also degrade performance by introducing unnecessary stalls.

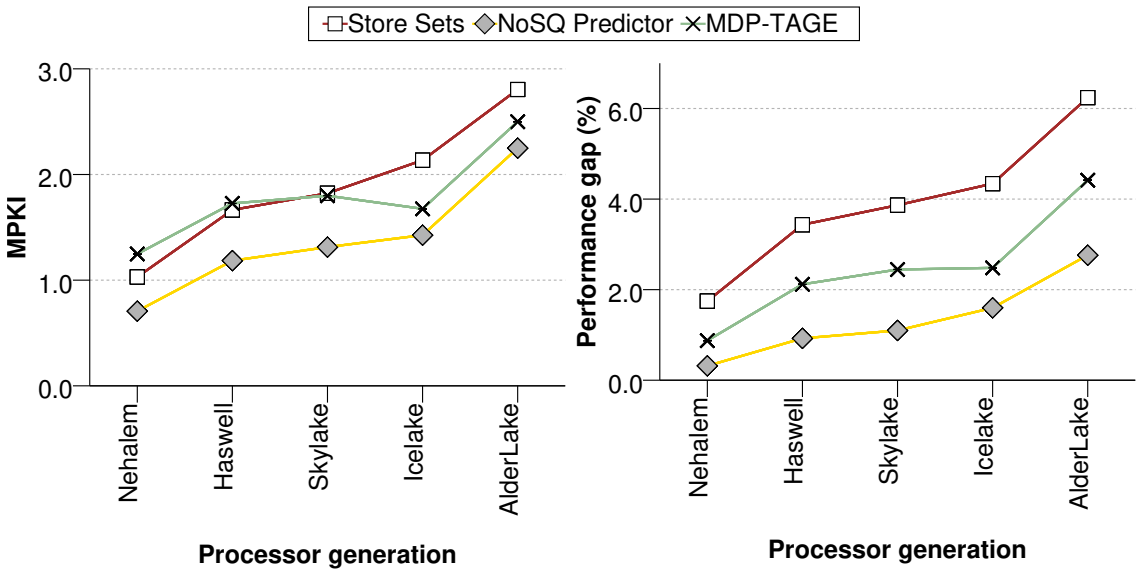
The increasing number of in-flight instructions in modern processors exacerbates

the challenge for MDP. Larger SQ sizes and wider execution windows result in more unresolved stores and more opportunities for loads to overtake them. Figure 5.2a shows the MPKI of state-of-the-art predictors across processor generations.[1] For example, Store Sets, which achieve around 1 MPKI in a Nehalem-like processor [36], reach nearly 2 MPKI in an Golden Cove-like processor [76]. Since the number of branch mispredictions remains relatively independent of processor size, memory dependence mispredictions now account for a larger share of squashes in modern processors. Moreover, the cost of squashes has grown, as more instructions are discarded per misprediction. Consequently, memory dependence predictors fall increasingly short of ideal behavior. As illustrated in Figure 5.2b, the performance gap of Store Sets relative to an ideal predictor increases from 1.8% in Nehalem-like processors to 6.0% in Golden Cove-like processors. This trend motivates a renewed study of memory dependence prediction.

Early predictors such as Store Sets, CHT, and Store Vectors present two main limitations: (i) they associate loads with sets of stores, and (ii) they do not explicitly incorporate execution context such as branch history. These limitations result in unnecessary stalls caused by false positives.[2] More recent approaches, such as NoSQ [85] and MDP-TAGE [71], track a single store (in particular, the *store distance* [96], defined as the number of intervening stores between the conflicting load and store) and incorporate context information to handle path-dependent dependencies. However, these designs train predictors using fixed history lengths, which, as we show in Section 5.1, such training provides sub-optimal performance when the history length is either too short (leading to false

[1] Results for CHT and Store Vectors are omitted as they underperform Store Sets.

[2] In Figure 5.1, Store Sets does not exhibit high false-positive MPKI because artificial store-store dependencies substitute them, see Section 5.4.




(a) Average MPKI (lower is better)

(b) Performance gap over ideal prediction (lower is better)

Figure 5.2: Trends in MDP for successive processor generations and the five state-of-the-art memory dependence predictors evaluated in this work: Store Sets, Store Vectors, the NoSQ predictor, and MDP-TAGE

positives) or unnecessarily long (causing excessive resource usage as the number of tracked histories dramatically increases).

This thesis makes two key observations. First, at most one store determines the value consumed by a load: when multiple preceding stores target the same address, only the youngest is relevant for dependence. The only exceptions are rare cases where a wide load reads partially overlapping data written by multiple narrow stores (Section 5.1.1). Second, the minimal context needed for accurate MDP is the execution path between the store and the load. Including branches older than the conflicting store does not improve prediction accuracy and pollutes prediction structures (Section 5.1.2).

Based on these observations, this chapter introduces PatH-Aware STore-distance (PHAST), a memory dependence predictor that leverages (i) the execution path

from the conflicting store to the load and (ii) the store distance. The path is represented by the global history of divergent branches, i.e., any branch that can take different paths on different executions, including both conditional and indirect branches. PHAST is trained using a history length corresponding to $N+1$ divergent branches, where $N$ is the number of such branches between the store and the load. During prediction, loads consult multiple histories, and on a match, the predictor provides the associated store distance (Section 5.2).
The main contributions of this chapter are:

- Demonstrating that the path from the store to the dependent load is the only context information required to achieve near-ideal accuracy (within 0.47% of an oracle).
- Designing PHAST, a cost-effective predictor that minimizes resource usage by training only with the history length required for each dependence.
- Showing through cycle-level simulation that PHAST achieves an average MPKI of 0.766, reducing mispredictions by 62.0% compared to NoSQ. With a 14.5KB budget, PHAST outperforms larger predictors such as 19KB NoSQ and 38.6KB MDP-TAGE, achieving mean speedups of 1.29% (up to 22.0%) and 3.04% (up to 38.2%), respectively.

## 5.1 Motivation

This section elaborates on the two guiding observations: loads typically depend on a single store, and accurate prediction requires only the execution path between store and load.

### 5.1.1 Store sets versus single store

Early memory dependence predictors [18,89,96] linked loads with sets of stores. While this approach effectively reduces false negatives (i.e. avoids memory order violations), it increases false positives, delaying loads unnecessarily (Chapter 4, Section 4.2). In practice, a load typically depends only on a single store. Even when several previous stores target the same address, the load only depends on the youngest among them.

Figure 5.3 illustrates several scenarios involving two stores and a load targeting the same address to motivate our claim. In case (a), the load correctly forwards data from the appropriate store. In case (b), the load bypasses the second store and gets the data from the older one. The younger store will detect the memory order violation once it executes. In case (c), the load executes after the younger store, but before the older one. Similar to case (b), once the unresolved store computes its effective address, it will detect the bypassing load and trigger squash. Since the load has obtained the correct value, it should not be squashed. In case (d), the load overtakes both stores; however, the predictor needs only to capture the dependence with the younger one. To summarize, in all cases where the load waits for the second store, it is not squashed. Thus, learning the correct store distance suffices for accuracy.

A small fraction of loads depend on multiple stores (e.g., due to partial writes). Still, our analysis (Figure 5.4) shows that (1) such cases are rare (0.04% of executed loads, on average) and (2) mostly execute in order. In addition, the benchmark with the highest amount of wide loads depending on multiple narrow stores is *503.bwaves*, where such loads account for only 0.25% of the total, while fourteen benchmarks exhibit none at all. Therefore, predicting a group of stores is unnecessary; tracking only the youngest conflicting store is sufficient. When

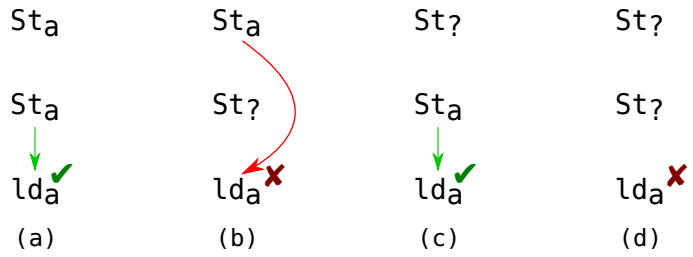



Figure 5.3: Examples of two stores targeting the same address as a subsequent load. Stores with a question mark as a subscript indicate that they have not yet computed their target address. Arrows indicate forwarding (if red, incorrect forwarding). The red *x* indicates that the load will be squashed when the store computes its target address

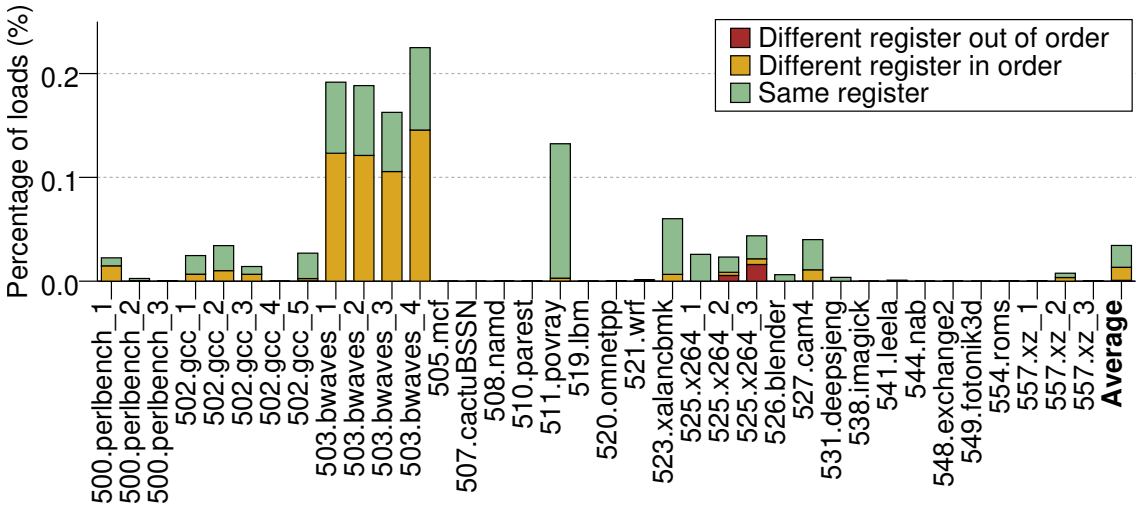



Figure 5.4: Percentage of loads that depend on multiple stores

the dependence varies across paths, context information must disambiguate the correct store.

### 5.1.2 Context information

Store Vectors [89] attempted to integrate context information in their prediction using global branch direction history but had limited impact. Later designs, such

as NoSQ and MDP-TAGE successfully used global branch history. When using context information, the benefits are brought when a single store distance is predicted.

However, both NoSQ and MDP-TAGE rely on predetermined lengths. NoSQ uses a fixed history length of eight branches, while MDP-TAGE uses several geometrically increasing history lengths. Training blindly with such predetermined history lengths is suboptimal: if the length is too short, false dependences may be introduced; if the length is too long, resources are wasted. In contrast, this chapter presents a mechanism that dynamically determines the appropriate history length based on the context of the conflict.

Existing memory dependence predictors that exploit context information are adapted from designs originally proposed for branch prediction where there is no inherent notion of an optimal history length. For instance, MDP-TAGE needs to perform a brute-force-like search to find a suitable history length for each prediction. In contrast, this chapter argues that the *optimal history length for MDP corresponds precisely to the path between the store and the load*. If this path recurs, the dependence is likely to recur as well. Consequently, PHAST trains with this path length.

Specifically, only divergent branches (conditional and/or indirect) are considered, as they uniquely identify control flow. For conditional branches, the direction bit suffices, while for indirect branches, the target address is required. Additionally, the destination of the divergent branch immediately preceding the store is included to disambiguate cases where different paths converge. In total, $N + 1$ branches are considered, where $N$ is the amount of branches comprehended between the conflicting store and the load.

Figure 5.5 illustrates why the extra branch is needed, using two examples. In both cases, the left path includes a store and a direct jump. On the right

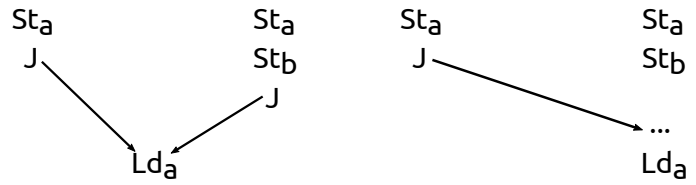


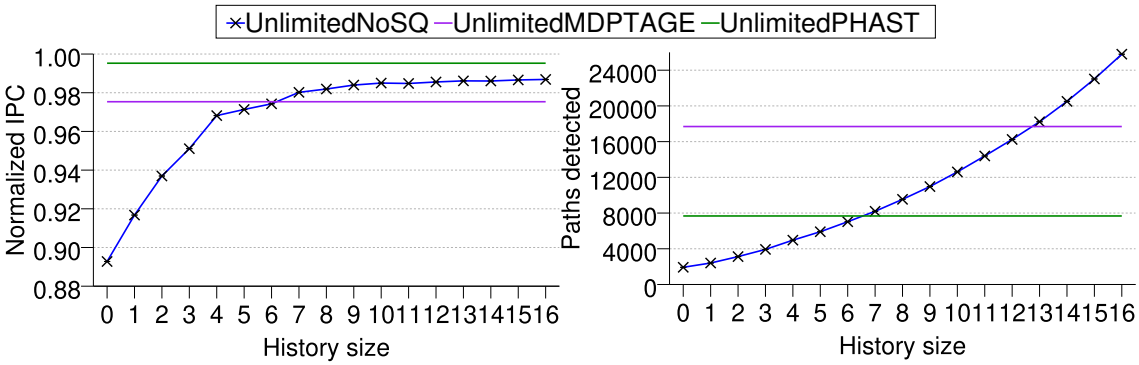

Figure 5.5: Two scenarios in which information about where the conflicting store is located in the code is required for path disambiguation

path, additional instructions may exist, but only non-divergent branches are executed. The store distance on the left path is 0, while on the right path is 1. Since no divergent branch are executed in either case, the same history would lead to conflicts with different store distances (0 or 1), depending on the previous execution path. By taking into account the target of the divergent branch preceding the conflicting store, PHAST can disambiguate these paths effectively.

### 5.1.3 Analyzing unconstrained predictors

To demonstrate the claims made in this section, we conducted a study using unconstrained predictors (Figure 5.6). We executed unlimited versions of NoSQ (blue line) considering history lengths from 1 to 16 branches (x-axis), and MDP-TAGE (purple line). The history tracks the taken/not-taken outcome of conditional branches and either the full PC of calls for NoSQ, or the targets of indirect branches for MDP-TAGE, and finishes with the PC of the dependent load. No compression is performed to prevent aliasing from occurring. Figure 5.6a shows the IPC normalized to an ideal predictor, while Figure 5.6b tracks the average number of paths detected to perform the predictions.

NoSQ exhibits only marginal IPC improvement beyond a history length of nine branches, while the number of paths detected increases exponentially. MDP-TAGE achieves higher IPC than the 6-branch NoSQ predictor, but lower than




(a) IPC normalized to ideal MDP (higher is better)

(b) Average number of paths (lower is better)

Figure 5.6: IPC and average number of paths detected for UnlimitedNoSQ, using different history sizes (x axis), for UnlimitedMDPTAGE, and for UnlimitedPHAST

7-branch NoSQ, as its ideal performance is biased towards the shortest history used (six branches). Although longer histories (allocated when the smaller history misses dependent stores) can in principle improve performance, this gain is minimal and comes at the cost of tracking a considerable number of paths—over 16000, on average—which will not help prediction when the branches are older than the one preceding the conflicting store, as claimed in this work.

In contrast, unlimitedPHAST (green line) is trained using the history length that effectively captures the context of the conflict. This adaptative training improves performance over both NoSQ and MDP-TAGE by always updating the entry that corresponds to the optimal length. In other words, when the path between a load and its conflicting store recurs, PHAST allocates a single entry, whereas MDP-TAGE distributes the same path across multiple entries and updates only the entry used for that prediction.

An example of the advantages of PHAST can be seen in *511.povray*, where a load can conflict with three different stores separated from the load by a single indirect branch. MDP-TAGE, which begins training with a history length of six branches,

suffers from extra memory order violations until all possible combinations are learned. In contrast, PHAST incurs only a single violation per store by using a two-branch history.
Finally, unlimitedPHAST achieves these improvements with less than one-third of the paths detected by the 16-branch NoSQ and about half of those detected by MDP-TAGE. This reduced number of paths will derive in either less aliasing or less storage requirements in practical, limited implementations of the predictor. The performance gap between unlimitedPHAST and an ideal predictor is just 0.47%, comfirming that the proposed selection of history length is effective for MDP.

## 5.2 PHAST

This section introduces PHAST, a novel memory dependence predictor based on the principles that (1) loads typically depend on at most one store and that (2) accurate prediction requires the minimum history length identifying the path between store and load. The following subsections describe the behavior of PHAST and its implementation.

### 5.2.1 Predictor behaviour

The operation of a memory dependence predictor involves four aspects: detecting dependencies, updating the predictor, predicting dependencies, and propagating them to the scheduler.

#### 5.2.1.1 Detecting dependencies

Before describing how PHAST is updated, it is important to clarify two main techniques used to squash mispeculated instructions: *eager squash* and *lazy squash*.

With eager squash, instructions are squashed as soon as a violation is detected, potentially discarding instructions that are part of the wrong path. In contrast, lazy squash dealys squashing until the commit stage, ensuring that only truly mispeculated instructions are discarded.

The next design question concerns when the predictor should be updated. Updating upon mispeculations enables faster training but risks learning wrong conflicts, as it may record dependencies that either de not exist in the correct path or do not correspond to the youngest conflicting store for that load. Conversely, updating at commit guaranteeing correctness, but slows down the training process.

Our evaluation uses lazy squash for memory conflicts. We analyzed both update strategies. All state-of-the-art predictors performed better when updating at mispeculation, with the exception of NoSQ, which remained unaffected. However, PHAST benefits from updating at commit, as this prevents pollution of the predictor with spurious dependencies. Notably, updating at commit remains compatible with eager squash through the use of a buffer that tracks mispeculated instructions with the necessary information to train the predictor.

As illustrated in Figure 5.3d, updating the predictor at mispeculation may cause training with the first store if it happens to execute earlier, as discussed previously. By waiting until commit, all stores have executed, avoiding updates with unnecessary or transient dependencies.

On the other hand, as explained in the previous chapter, when a load receives the data through store-to-load forwarding, it should not be squashed by stores older than the forwarder (Figure 5.3). Although this observation may seem intuitive, such squashes occur even in accurate research simulators such as gem5 [51] and can significantly impact the predictor behavior, as shown later in Figure 5.12. To prevent these false squashes, the load can track the sequence number of the forwarding store. When stores snoop the LQ, they compare their sequence

number against that of the forwarder and ignore the memory order violation if the forwarder is younger. NoSQ employed a different mechanism to achieve a similar effect, filtering squashes through the Store Vulnerability Window concept (see Section 5.4 for details).

#### 5.2.1.2 Updating the predictor

PHAST is updated when a true dependence is detected, using both the store distance and the path from the store to the load. The store distance is derived from SQ indices by computing the difference between the index of the store immediately preceding the load and that of the conflicting store, similar to the method used in Store Vectors [89].

The history length is determined with a global branch counter that tracks the number of decoded divergent branches. Each decoded load and store receives a copy of the register value. Upon conflict, the history length can be computed as the difference between the two recorded values. To account for wraparounds, the register should be wide enough to track the maximum number of divergent branches plus one extra bit [14].

When a load is about to be squashed at the commit stage, the predictor is updated with the PC of the load and the corresponding history length. The branch history is then collected from a global history register at commit[3]. This global register tracks—both at decode and commit—per divergent branch: a bit indicating the type of branch (conditional or indirect), a bit indicating if it is taken or not, and a few bits of the target of the branch (e.g., 5 least significant bits). Maintaining all history entries at a uniform length simplifies parallel processing.

The recorded history is constructed using the branch outcome (taken/not taken)

[3]Alternatively, we could access the history at decode since we know the number of branches that the load is ahead of.

bit for conditional branches and the destination for both indirect branches and the divergent branch immediately preceding the store, regardless of its type. The history is then hashed together with the PC of the load to generate the index and tag of the predictor caches.
The predictor entry stores a tag, a distance field, and an n-bit confidence counter used to disable aliased dependencies that cause mispredictions. When allocating a new entry, the store distance is recorded and the confidence counter is initialized to its maximum value. If an existing entry is found with non-zero confidence, it is replaced.
Finally, when a load with a predicted distance commits, it updates the confidence of the corresponding prediction. If the prediction was correct—i.e., the load received its data from the store it waited for—the confidence counter is saturated to the maximum; otherwise, it is decremented.

#### 5.2.1.3 Predicting dependencies

At decode, loads consult the memory dependence predictor. The prediction is performed across the set of history lengths tracked by the predictor. Using more history lengths improves accuracy but also increases the number of lookups. The PC of the load is hashed with each history in the same manner as during predictor updates. Matches with non-zero confidence yield the store distance. If multiple matches are found, the prediction corresponding to the longest history length is selected; otherwise, no dependence is predicted.
Although this process requires multiple searches using different history lengths, memory dependence prediction is not as critical as branch prediction and can therefore take several cycles to provide the prediction without impacting performance. The prediction can begin as soon as the load is decoded, while its

outcome is only required once the load is inserted into the issue queue. Notably, high-performance branch predictors such as TAGE also perform several parallel lookups using different history lengths.

#### 5.2.1.4 Propagating predictions

Predicted store distances must be translated into dependencies so that the load waits for the appropriate preceding store to execute. To achieve this, the predicted store distance is subtracted from the SQ index of the most recent store, thereby determining the dependent store entry—similar to the mechanism used in NoSQ and MDP-TAGE. It is worth noting that PHAST, like other store distance predictors, operates independently of the underlying synchronization method.

### 5.2.2 A cost-effective implementation

Our implementation of PHAST maintains one table per history length, which are searched in parallel on each prediction—similar to the structure of a TAGE branch predictor, a design already adopted in commercial processors [24].
Since tracking the full range of history lengths is impractical from both scalability and lookups perspectives, the first design decision concerns the selection of the history lengths to be tracked. Based on the performance analysis of the unlimitedPHAST across several maximum history lengths (see Section 5.3.1), we determined that a maximum of 32 branches is enough for highly accurate prediction. Consequently, PHAST uses a sequence of eight history lengths: (0, 2, 4, 6, 8, 12, 16, 32). Histories not covered are truncated. For example, a conflict with a 9-branches history uses the 8 branches closer to the load. It is worth noting that the optimal history lengths for MDP differ from those for branch prediction, implying that an omnipredictor [71] cannot be easily tuned to perform optimally for both.

Unlike the unlimited version, PHAST accesses with a compressed representation of the history. Sensitivity analysis shows that using the five least significant bits of the branch targets effectively prevents most aliasing scenarios. The history is folded until $S + T$ bits remain, where $S$ is the number of index bits and $T$ is the number of tag bits. The load's PC is then hashed as follows: $(PC \bigoplus (PC >> 2) \bigoplus (PC >> 5))$, while for the tag the PC is offset by 3 and 7. The hashed PCs and the compressed history are then combined with an exclusive OR[4].

Each table is four-way associative and contains entries with a 16-bit tag, a 7-bit store distance field, a 4-bit confidence counter, and a 2-bit field for the less-recently-used (LRU) replacement policy. With 128 sets per table (512 entries), the total storage requirement is 14.5KB, achieving an effective balance of accuracy and efficiency.

## 5.3 Evaluation

This section evaluates the performance of several memory dependence predictors, using an oracle predictor as the baseline. The experiments were conducted by simulating a Golden Cove-like processor executing the SPEC CPU 2017 benchmark suite. A detailed description of the methodology is provided in Chapter 3.

We compare our proposal, PHAST, against state-of-the-art memory dependence predictors listed in Table 5.1: Store Sets [18], the predictor employed by NoSQ [85], and MDP-TAGE [69]. For each predictor, Table 5.1 reports the configuration and storage corresponding to the best performance-storage trade-off is presented, while a detailed storage sensitivity analysis is provided in Section 5.3.3.

[4]This hash functions are used across all evaluated predictors, as they consistently improve performance.

Table 5.1: Configuration of the state-of-the-art predictors for Chapter 5

| **Predictor** | **Tables** | **Total entries** | **Fields per entry** | **Energy per access (pJ)** | **Size** (KB) |
|---|---|---|---|---|---|
| Store Sets | SSIT | 8K | valid bit<br>12 bit SSID | 0.2403 | **18.5** |
| | LFST | 4K | valid bit<br>10 bit St ID | 0.1026 | |
| NoSQ | 2 | 4K | 22 bit tag<br>7 bit counter<br>7 bit distance<br>2 bit lru | 0.3721 | **19** |
| MDP-TAGE | 12 | 16K | 7-15 bit tag<br>7 bit distance<br>1 bit $u$ | 1.3103 | **38.625** |
| MDP-TAGE-S | 8 | 4K | 16 bit tag<br>7 bit distance<br>2 bit lru<br>1 bit $u$ | 0.4421 | **13** |
| PHAST | 8 | 4K | 16 bit tag<br>4 bit counter<br>7 bit distance<br>2 bit lru | 0.4856 | **14.5** |

All state-of-the-art memory dependence predictors have been previously described in Chapter 2. However, the MDP-TAGE version evaluated in this work is a standalone memory dependence predictor (i.e. it only performs MDP), and the 3-bits counter has been expanded to 7-bits to track all store distances, thus avoiding coarse linking. The MDP-TAGE features 12 components with (6, 2000) geometric history lengths [80]. Additionally, we also evaluated a variant with the same table and history lengths configurations as PHAST (labeled MDP-TAGE-S for *S*horter history lengths) to demonstrate that PHAST's improvements stem from its design principles rather than specific configuration choices.

The NoSQ predictor was originally proposed for speculative memory bypass. For

this evaluation, we adapted it for memory dependence prediction. It uses two set-associative tables: one indexed solely by the load PC (path independent) and another indexed by the PC combined with 8 bits of global history (1 taken/not-taken bit per conditional branches and 2 bits of the PC per call).
The analysis proceeds as follows. First, we evaluate the potential of the unlimited version of PHAST. Next, we study the impact of filtering memory order violations when store-to-load forwarding is enabled. Finally, we compare PHAST against the best-performing state-of-the-art predictors.

### 5.3.1 Potential of PHAST and analysis

We begin by analyzing the per-application performance of the unlimitedPHAST predictor relative to an ideal predictor. Figure 5.7 presents the IPC of UnlimitedPHAST normalized to an oracle predictor. On average, UnlimitedPHAST achieves performance only 0.47% below the ideal case. The largest performance gaps are observed in *502.gcc_1*, *502.gcc_2*, *510.parest*, and *541.leela*. The *502.gcc* benchmarks exhibit the highest number of unique paths among all workloads, along with many occasional dependencies that are not path-dependent. In contrast, *541.leela* exhibits below-average number of paths but experiences a similar level of memory order violations as *502.gcc*, in addition to a higher incidence of false dependencies.
Figure 5.8 shows the MPKI values for UnlimitedPHAST. Most memory order violations stem from cold misses at the beginning of execution. Benchmarks such as *502.gcc_1* and *502.gcc_3* display high violation counts, though these are largely attributable to cold-start effects. Both benchmarks also exhibit an unusually large number of paths to track.
False dependencies are most pronounced in *510.parest*, *531.deepsjeng*, *541.leela*,

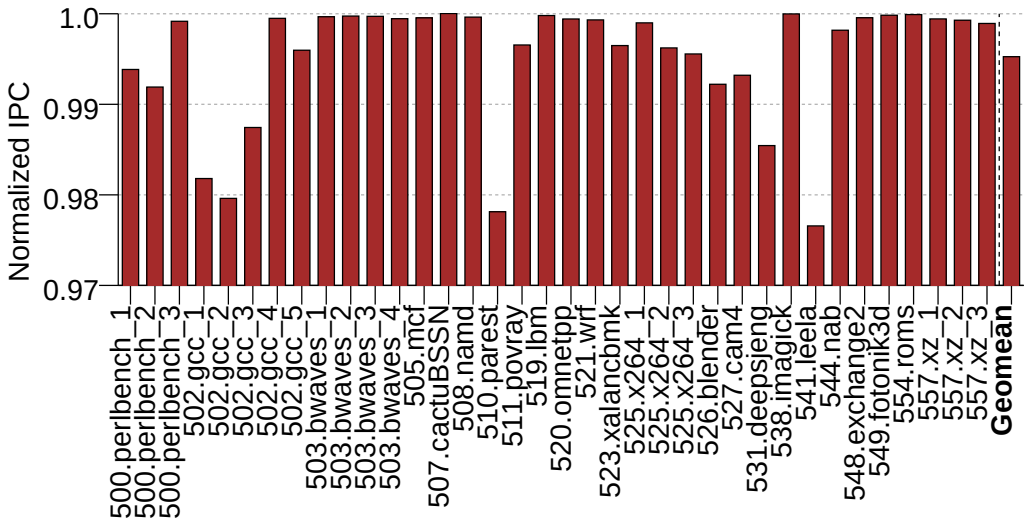



Figure 5.7: IPC of the UnlimitedPHAST predictor normalized to a perfect memory dependence predictor (higher is better)

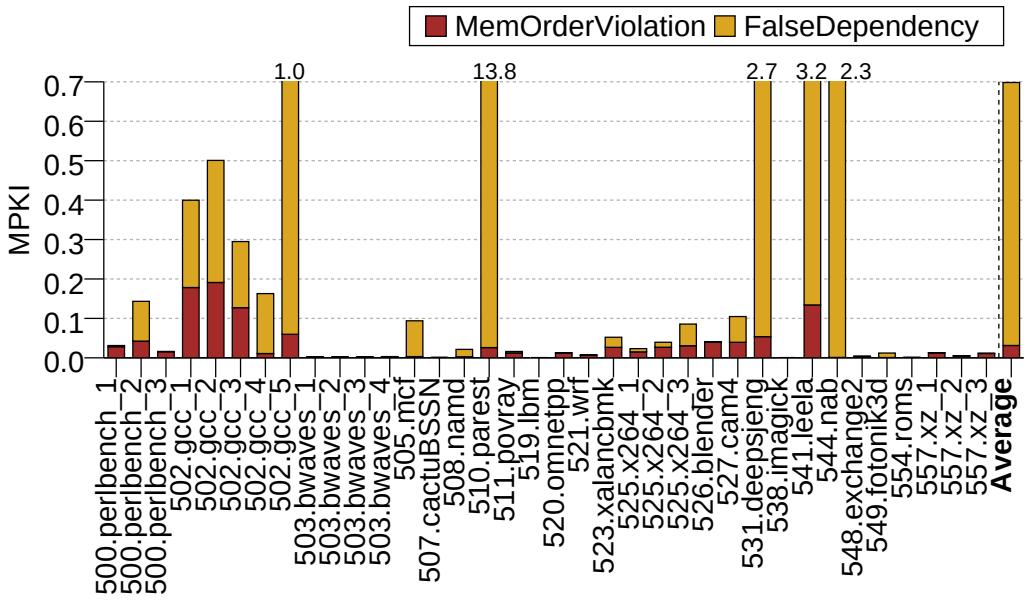



Figure 5.8: MPKI of the UnlimitedPHAST predictor (lower is better)

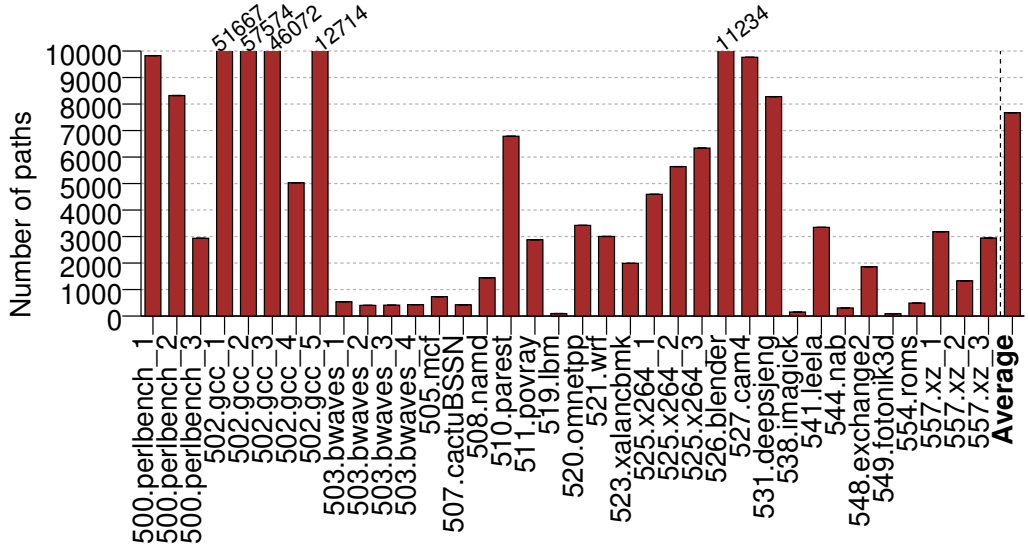



Figure 5.9: Number of paths registered per application with UnlimitedPHAST

and *544.nab*, as well as all variants of *502.gcc*. Because UnlimitedPHAST explicitly tracks the full path from the branch preceding the conflicting store to the dependent load, aliasing can be ruled out. Instead, these cases arise from data-dependent rather than path-independent load–store pairs, so they conflict only occasionally. False dependencies degrade performance only if the affected loads lie on the critical path.

Figure 5.9 illustrates the number of paths per application detected with UnlimitedPHAST. For most benchmarks, the number of paths remains below five thousand. Notable exceptions include *500.perlbench_2*, *502.gcc_1–5*, *526.blender*, *527.cam4*, and *531.deepsjeng*. However, in these applications, the majority of paths are exercised only a few times, particularly the longest ones. This observation suggests that pruning long histories would have negligible impact on overall performance.

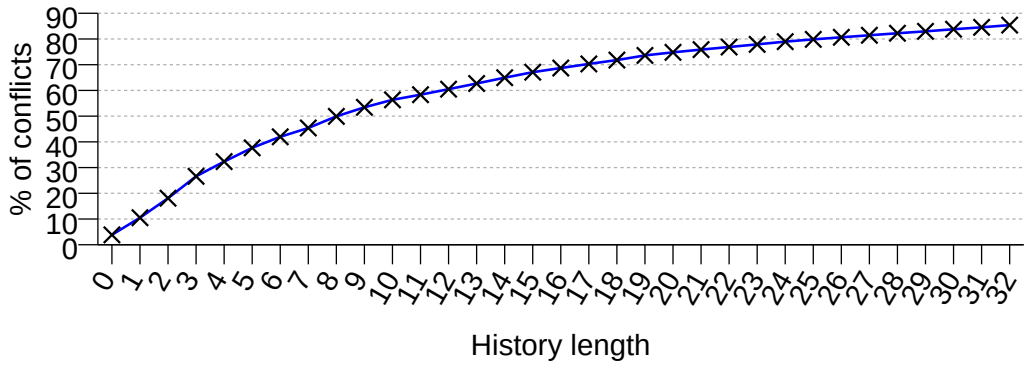



Figure 5.10: Percentage of conflicts detected at each history length

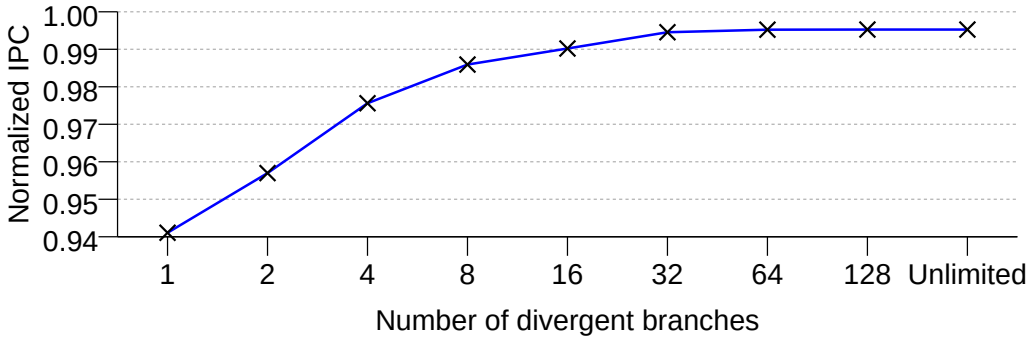



Figure 5.11: Normalized IPC of UnlimitedPHAST at several maximum history lengths (higher is better)

Finally, Figure 5.10 presents the distribution of unique conflicts across history lengths. The results show that 85.4% of conflicts occur within histories of up to 32 branches, with 73.6% falling within [0–19] branches. Dependencies requiring longer histories are rare and have limited impact on execution time. Consistently, Figure 5.11—which shows the normalized IPC of unlimitedPHAST with different maximum history lengths—demonstrates that capping the maximum history length at 32 branches achieves performance equivalent to the unlimited case. For most benchmarks, even 16 branches suffice, though a small subset benefits from longer histories.

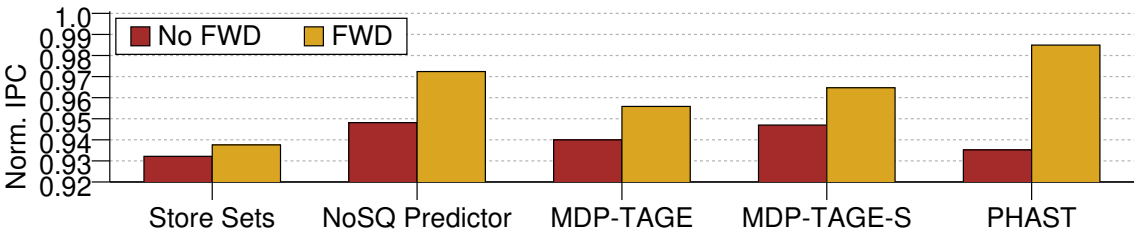



Figure 5.12: IPC of memory dependence predictors against a perfect predictor with (FWD) and without filtering through forwarding (higher is better)

### 5.3.2 Effect of avoiding squashes on forwarding

Figure 5.12 compares the IPC of the evaluated predictors relative to an ideal predictor, both with (FWD) and without (No FWD) the forwarding-based optimization described in Section 5.2.1.1. Although absent in prior state-of-the-art simulators, this optimization proves critical when predicting dependencies involving a single store. Unless otherwise specified, all results in this chapter are reported with FWD enabled.

Enabling the forwarding optimization yields a negligible improvement for Store Sets (less than 1%), primarily due to (1) loads must still wait for a set of stores, and (2) set merging effects introduce many false dependencies. In contrast, NoSQ and MDP-TAGE gain approximately 2%, since loads wait for at most one store, and filtering re-executions reduces incorrect memory order violations such as those depicted in Figure 5.3(c).

PHAST benefits the most, with a 5% IPC improvement. When multiple stores share the same load address (Figure 5.3), the forwarding filtering ensures that PHAST learns the dependence only on the most recent store, avoiding spurious associations with older stores. Without forwarding, PHAST would learn older, *incorrect* dependencies— often associated with longer histories—leading to persistent mispredictions until confidence counters are reset. At that point, although

PHAST would produce the correct distance, the memory order violation with the older store could recur. Although NoSQ and MDP-TAGE are also exposed to this issue, their strategies for mitigating violations (e.g., path-insensitive tables or probabilistic resets) make them less sensitive, at the cost of increasing the false dependencies.

### 5.3.3 Comparison to state-of-the-art predictors

We begin our comparison of PHAST against state-of-the-art predictors by examining the performance-storage trade-off shown in Figure 5.13. PHAST consistently outperforms competing predictors while requiring less storage. Both PHAST and MDP-TAGE-S share similar prediction structures, but differ in training: PHAST trains at the *minimum effective history length*, while MDP-TAGE starts from a short fixed length and extends upon mispredictions. Even with a modest budget of 7.25KB, PHAST achieves 97.74% of the ideal IPC, still surpassing all baselines. For Store Sets and NoSQ, doubling the storage yields no measurable gains. For the remainder of the evaluation, each predictor is configured with the storage size yielding the best trade-off, as summarized in Table5.1.

Next, we report the per-benchmark MPKI in Figure 5.14. Store Sets implicitly captures some path sensitivity, since dependencies are predicted only when the store instance is in-flight [85]. While allowing both loads and stores to access the predictor can reduce false dependencies by not stalling the load unnecesarily if the store is not present—compared to predicting a store distance and linking the load with whichever store is present at that distance—these additional accesses, combined with set merging effects, lead to an increase of aliasing due to unrelated loads and stores being mapped to the same set, which ultimately increases false dependencies, as discussed in Chapter 4. NoSQ and MDP-TAGE mitigates many mis-speculations through explicit path information. MDP-TAGE-S further

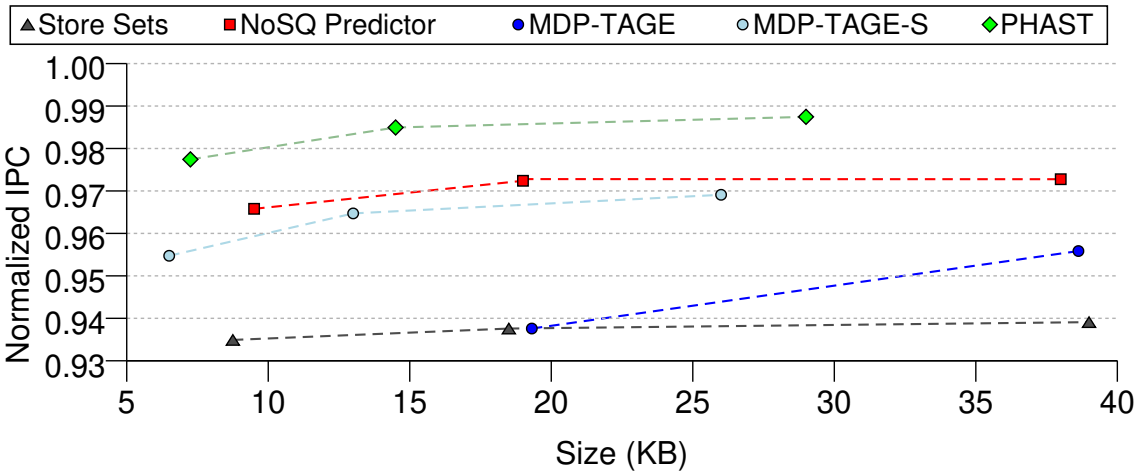



Figure 5.13: Performance (higher is better) versus storage of Store Sets, NoSQ predictor, MDP-TAGE, MDP-TAGE with PHAST configuration and PHAST at different budgets compared against an ideal MDP

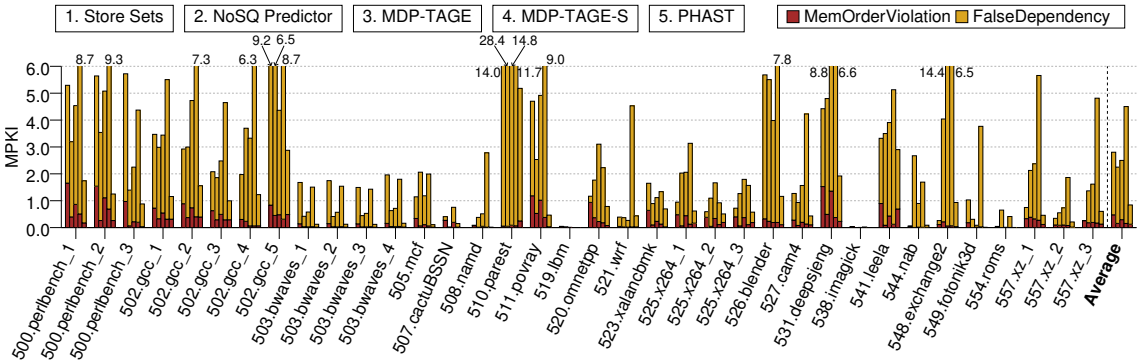



Figure 5.14: MPKI of the evaluated memory dependence predictors with the exception of CHT (lower is better)

reduces false negatives compared to MDP-TAGE by favoring shorter histories, confirming that omnipredictors cannot be easily tuned for both branch and memory dependence prediction. Yet, MDP-TAGE-S suffers from the highest false-positive MPKI due to the abundance of short-history tables, particularly the 0-branch table. PHAST achieves the lowest MPKI overall, minimizing both false negatives and false positives.

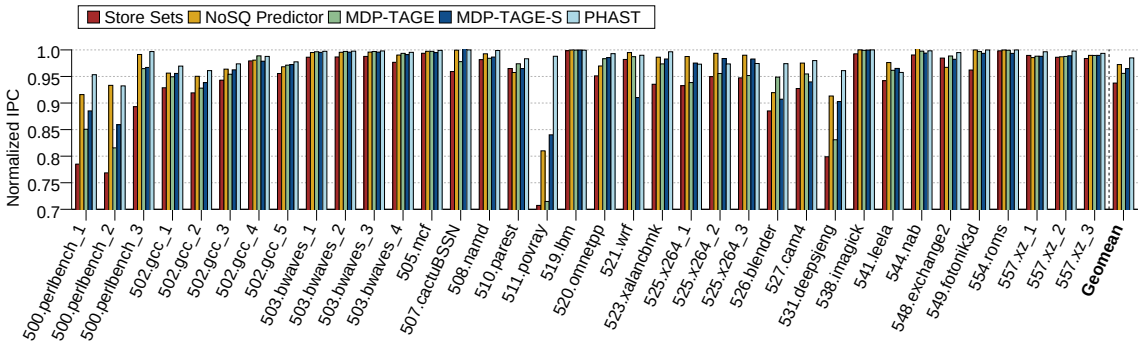



Figure 5.15: IPC of the memory dependence predictors per application normalized to the perfect MDP (higher is better)

Although UnlimitedPHAST struggled in *502.gcc_1–3* and *541.leela*, the bounded version significantly reduces MPKI compared to competing predictors. In *502.gcc*, PHAST halves the false positives relative to NoSQ and MDP-TAGE-S while maintaining similar false negatives. In *541.leela*, the lack of path-dependent conflicts reduces PHAST's advantage, and false positives increase. Once the confidence counter reaches zero, the next time the conflict is presented, PHAST would not predict it, leading to a memory order violation. NoSQ mitigates false negatives with its path-insensitive table but incurs more false positives.

Conversely, PHAST excels in benchmarks such as *500.perlbench*, *511.povray*, and *531.deepsjeng*, substantially lowering both types of mispredictions. The strong correlation between memory dependencies and branch history in *511.povray*, also reported in [69], highlights the benefit of well-chosen history lengths.

Figure 5.15 presents the IPC normalized to an ideal predictor. PHAST narrows the gap to 1.5%, outperforming Store Sets by 5.05% (up to 39.7%), NoSQ by 1.29% (up to 22.0%), MDP-TAGE by 3.04% (up to 38.2%), and MDP-TAGE-S by 2.10% (up to 17.6%). These gains directly stem from the MPKI reductions in Figure 5.14. Another limitation of Store Sets is its handling of multiple in-flight store instances, which forces in-order execution and always binds loads to the youngest instance.

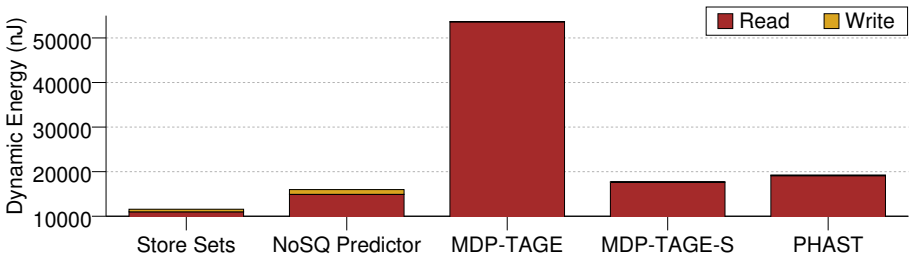



Figure 5.16: Energy consumption (nJ) of the evaluated predictors

NoSQ and PHAST overcome this by incorporating path information—as observed in *500.perlbench_3*, where all path-aware predictors achieve at least 95% of ideal IPC while Store Sets lags behind.

MDP-TAGE slightly outperforms Store Sets by restricting each load to depend on at most one store. However, its reliance on blind training across geometric history lengths increases mispredictions compared to NoSQ and PHAST, proving that when predicting for a single store, accuracy is paramount. NoSQ avoids memory order violations with its path-insensitive table but at the cost of more false dependencies.

Compared to NoSQ, PHAST achieves an average speedup of 1.29% and matches or exceeds its performance in all applications except *525.x264* and *541.leela*. Across the suite, PHAST reduces MPKI by 62% relative to NoSQ, with 20% fewer memory order violations and 65% fewer false dependencies. This improvement arises from PHAST's selective use of path segments between the conflicting store and dependent load, whereas NoSQ's fixed-length histories either explode in number or increase false positives.

Finally, in terms of energy, PHAST reduces re-executed instructions by 2% relative to NoSQ and by 8% compared to Store Sets. This leads to a meaningful reduction in overall core energy consumption. Figure 5.16 breaks down predictor energy into reads and writes, showing that standard TAGE-like predictors are

substantially more power-hungry. Nevertheless, the memory dependence predictor accounts for a small fraction of the total core energy budget compared to larger structures such as branch predictors and BTBs.

## 5.4 Related work

NoSQ [85] implements speculative memory bypassing (SMB) [59, 92] without using a SQ. It leverages the Store Vulnerability Window (SVW) proposed by Roth et al. [77], which employs a small untagged direct mapped address-indexed table—named Store Sequence Bloom Filter (SSBF)—to track the sequence number of the youngest committed store per address. A load records the sequence number of the last committed store when it executes; if data is forwarded from another store, this value is updated accordingly. Prior to commit, the load queries the SSBF to verify whether a younger store has written to its address. For non-bypassing loads, re-execution is skipped if the recorded sequence number is less than or equal to the SSBF entry; for bypassing loads, the numbers must match exactly. NoSQ enhanced the SSBF into a tagged set-associative structure managed in FIFO order, improving the filtering of unnecessary squashes. Similarly, PHAST filters squashes arising from older stores conflicting with a forwarding one (Section 5.2.1.1).

Jin and Önder [39] extended NoSQ by addressing cases where the predictor has low confidence. Instead of bypassing, such loads wait until the corresponding store commits and updates the cache. To avoid the resulting delay, the authors introduced a predication mechanism in which both cache and store data are speculatively fetched; the correct value is selected once the prediction outcome is verified.

Alves et al. [11] proposed a mechanism to filter the L1/TLB probes by using a store-queue/buffer/cache (S/QBC) that partitions the SQ/SB into three logical structures by adding a buffer for data already written back. A memory dependence predictor based on store distance [96] predicts hits or misses in the S/QBC. Correctly predicted hits reduce energy by avoiding unnecessary L1/TLB probes, while correctly predicted misses reduce latency by enabling parallel probing of S/QBC and L1/TLB.

Lustig et al. [52] combined memory dependence prediction with memory disambiguation to support high-performance forwarding. The predictor anticipates same-address dependencies between stores and loads, and memory disambiguation validates the prediction. When correct, this pairing enables synonym detection while keeping the TLB off the critical path.

Huang et al. [31] introduced compiler-assisted memory dependence prediction. Their approach analyzes binaries to annotate loads guaranteed not to overlap with unresolved older stores. Hardware then uses these annotations to bypass memory disambiguation checks, reducing contention for LQ and SQ resources. Such annotated instructions require neither SQ lookups nor LQ entries.

Hasan [29] developed a perceptron-based predictor for energy-constrained systems. Inspired by perceptron branch predictors, the scheme uses a history vector recording outcomes of the last $n$ retired loads and whether they caused violations. This predictor achieved nearly the same IPC speedup as Store Sets. Although predictor performance was found to be largely insensitive to history length, we argue that path information between the conflicting store and load remains a more effective basis for accurate prediction.

## 5.5 Conclusion

This chapter introduced PHAST, a memory dependence predictor that leverages execution paths between a conflicting store and its dependent load, and predicts the exact store distance along that path. By using that path information, an unlimited PHAST predictor approaches the performance of an ideal predictor while requiring only a modest number of tracked paths.

With a storage budget of 14.5KB, PHAST surpasses all evaluated state-of-the-art predictors, delivering average speedups of 5.05% over Store Sets, 3.04% over MDP-TAGE, and 1.29% over NoSQ, with improvements of up to 22% relative to NoSQ. Overall, PHAST achieves within 1.5% of the performance of an ideal predictor, demonstrating both its efficiency and effectiveness.

## 5.6 Other outcomes of this work

Beyond the technical contributions discussed in this chapter, this research also gave rise to several noteworthy achievements and collaborations.

First, it was recognized at the 30th International Symposium on High-Performance Computer Architecture (HPCA), where the corresponding publication received a *Best Paper Honorable Mention* award.

Second, the ideas and results developed here have served as the basis for new collaborations with other research groups. Notably:

- A collaboration with the University of Cambridge resulted in the design of a new memory dependence predictor, MASCOT, presented at HPCA 2025 [57]. MASCOT performs both memory dependence and store-to-load speculative memory bypass prediction, and learns the context of dependencies as well as non-dependencies, thereby achieving even lower MPKI.
- Joint work with Luke Panayi from Imperial College London was initiated to integrate PHAST into the `gem5` simulator [61].

Chapter 6

# Effective Context-Sensitive Instruction Fusion Prediction

Computer architects continually seek to improve processor performance and energy efficiency by exploiting instruction- and memory-level parallelism and optimizing the microarchitectural execution pipeline. Instruction fusion is a technique that contributes to all of these goals by combining certain sequences of instructions into a single operation [4,7,8,32].

Instruction fusion offers several advantages. The utilization of back-end structures—such as issue queue (IQ), reorder buffer (ROB), load/store queue (LSQ), and register file—can be increased when a fused operation occupies a single entry. In addition, it also alleviates resource pressure, reducing the bottlenecks of register renaming and retirement as fewer operations are processed. Instruction fusion can also improve the front-end efficiency by increasing the decode bandwidth. In some cases, instruction fusion can reduce the execution latency of an instruction, as is the case of compare-and-branch fusion, reducing critical paths [25,33].

While instruction fusion has seen interest in both industry and academy, most

of the implementations share two key limitations: (1) they are non-speculative, and (2) they are restricted to consecutive instructions and in the case of memory operations to contiguous memory accesses.

Helios [86] was the first work that demonstrated the benefits of speculative, non-consecutive instruction fusion. The proposed predictor is a large tournament predictor [42] adapted for fusion. Although tournament predictors have historically improved accuracy by combining simpler predictors, they are limited in their ability to adapt to complex or context-dependent patterns. Given the advances in branch prediction—particularly TAGE [78,83], which leverages multiple history lengths to capture both short- and long-range correlations—a tournament-based approach remains suboptimal, as we show in this chapter.

However, TAGE-style predictors are not without drawbacks, as in the previous chapter, we demonstrated that—for memory dependence prediction—to identify the appropriate history length for a given prediction they often rely on inefficient brute-force searches across multiple tables. Instead, our PHAST design introduced a new type of context-sensitive predictor that learns the suitable history length for each conflicting pair. We argue that such a model is better suited for instruction fusion prediction than the current state-of-the-art or even TAGE-like predictors: if a fusible instruction pair consistently follows the same execution path, it is highly likely to form a valid fusion idiom.

Beyond predictor design, the Helios microarchitecture introduces additional challenges that limit performance. Chief among these is its reliance on oversized fusion windows. This design choice is problematic for two reasons. First, a non-consecutively fused (NCF'd) $\mu$-op cannot be issued until it is validated, introducing unnecessary delays. Second, the older $\mu$-op inherits all dependencies of the younger one, further stalling execution.

To further enhance non-consecutive instruction fusion and reduce associated

overheads, we propose FLIP (from *Fixed-history-Length Instruction Predictor*), a non-consecutive fusion predictor that trains accounting for the context between fused instruction pairs, and more importantly, using always the same branch history length for training the same pair. In conjunction with FLIP, we introduce a set of *Non-consecutive Fusion Optimizations* (NFOs). These optimizations are designed to either (i) relax execution constraints for fused $\mu$-ops—such as permitting unvalidated NCF'd $\mu$-ops to execute when ready—or (ii) suppress harmful fusion patterns by incorporating a watchdog mechanism to detect NCF'd $\mu$-ops in which the older instruction is ready but experiences prolonged stalls due to the younger instruction dependencies.

The main contributions of this chapter are:

- Designing FLIP, a non-consecutive instruction fusion predictor that leverages the context information between the fused instructions.
- Including a set of NFOs that relax execution constraints for fused $\mu$-ops or prevent harmful fusion from repeating.
- Showing through cycle-level simulation that FLIP, compared to Helios is able to increase coverage by 5.5%, reduce mispredictions by 83%, and deliver IPC improvements of 2.27% and 2.09% on SPEC CPU 2017 and MiBench, respectively, while requiring only half of the storage requirements of Helios' fusion predictor. When coupled with NFOs, enhanced FLIP delivers speedups o 2.44% and 2.94% over Helios.

## 6.1 Background

Non-consecutive fusion is a relatively recent research area with limited prior work in the literature. Aside from industry patents [91] that lack a detailed description and an evaluation, the only substantial proposal available in the

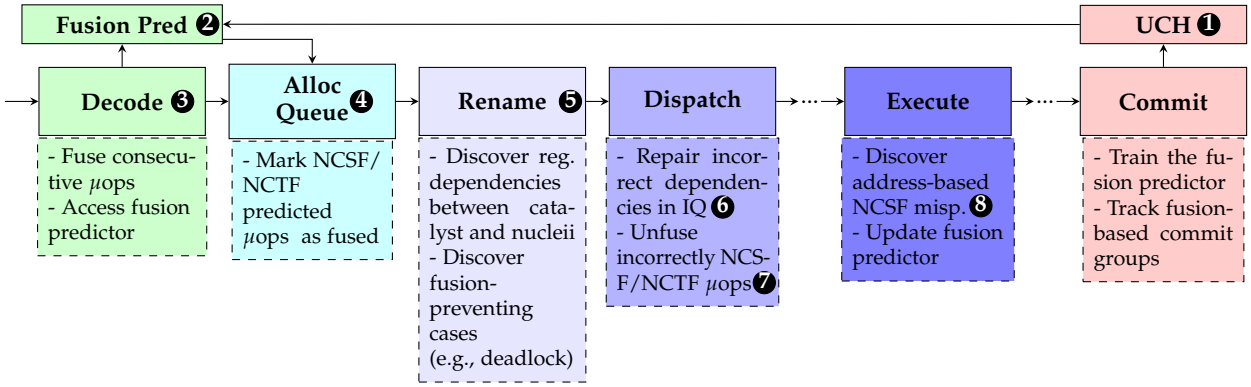



Figure 6.1: Helios fusion-related responsibilities overview. Recreated from [86]

literature is Helios. Therefore, this background section focuses on Helios, which serves as a reference point and as a baseline for comparison in our study.

In Helios, a Non-Consecutive Fused (NCF'd) $\mu$-op consists of three components: (1) the head nucleus, which refers to the older $\mu$-op in the fused pair and is replaced by the NCF'd $\mu$-op; (2) the tail nucleus, representing the younger $\mu$-op that is removed from the pipeline; and (3) the catalyst, comprising all intervening instructions (in program order) between the head and tail nuclei.

The Helios microarchitecture focuses on fusing load-load and store-store pairs, and introduces four key mechanisms to enable its fusion capabilities: (1) a fusion predictor, (2) a mechanism to perform the fusion, (3) a mechanism to validate the fused $\mu$-ops, and (4) support for handling incorrect fusion cases and mispredictions within the catalyst. Figure 6.1 provides an overview of the pipeline modifications to support Helios.

### 6.1.1 Helios Fusion Predictor

The predictor is responsible for identifying tail nuclei and providing them with the distance, in $\mu$-ops, to their corresponding head nuclei. It consists of two primary components: the *Unfused Committed History (UCH)*, which tracks recently committed $\mu$-ops that were not fused, and the *Fusion Predictor (FP)*, which leverages this information to detect fusion opportunities and supply the appropriate fusion distance.

UCH ❶ is implemented as a fully-associative cache with entries containing a partial cache line address tag, and a commit number (CN). The structure maintains several entries for loads and one for stores. When a retiring memory $\mu$-op matches an existing tag in the UCH, it indicates a potential fusion opportunity (two loads or stores accessing the same cache line). In this case, the FP is updated using the distance computed from the difference in CNs, and the matching UCH entry is invalidated, to avoid predicting the fusion of that $\mu$-op with another $\mu$-op. The maximum allowed fusion distance is 64 $\mu$-ops. If no match is found, the retiring $\mu$-op is inserted into the UCH, preferring invalid entries and falling back to the least-recently used (LRU) entry otherwise.

FP ❷ is a tournament predictor that selects from a "local" predictor indexed by the $\mu$-op's program counter (PC), and a "global" predictor that uses a hash of the PC and the global branch direction history. Fusion table selection is managed by an auxiliary selector, indexed using the PC hashed with the global branch direction history. Each entry in the predictor stores a confidence counter and a $\mu$-op distance. Correct predictions leave the confidence counter unchanged. However, if a misprediction is detected, the counter is reset to zero. The counter is increased until it saturates using the UCH-based mechanism.

### 6.1.2 Fusing the instructions

At decode ❸, the predicted distance is retrieved and used to attempt fusion in the Allocation Queue (AQ) ❹. Fusion is performed if: (i) the saturating counter for the predicted distance is at its maximum value, (ii) the candidate pair forms a valid idiom (e.g., both are loads or both are stores, and the head has not already been fused), and (iii) the head $\mu$-op is still resident in the AQ or is part of the same decode group as the tail.

At fusion time, the head nucleus is replaced by the NCF'd $\mu$-op, while the tail $\mu$-op remains in the Allocation Queue. The fused $\mu$-op includes a ready bit, initially set to zero, that is set to 1 after the NCF'd $\mu$-op has been validated by the tail nucleus. This bit prevents the fused $\mu$-op from issuing until it has been validated, ensuring that its source register identifiers are correct, that fusion will not introduce deadlocks, and, in the case of a fused store pair, that there is no intervening store between the head and the tail.

### 6.1.3 Validating the fused $\mu$-op

As previously discussed, a NCF'd $\mu$-op must undergo validation before it can be issued. This validation is performed when the tail nucleus reaches the rename stage ❺.

First, the tail nucleus checks for Read-after-Write (RaW) dependencies within the catalyst. If such a dependency is detected, the tail nucleus is dispatched to *repair* ❻ the NCF'd $\mu$-op by supplying the correct source registers. This repair process introduces an additional cycle of latency as penalty.

Second, the tail nucleus verifies whether any fusion-preventing conditions exist—such as potential deadlocks, the presence of serializing instructions within the catalyst, or, in the case of NCF'd stores, intervening stores. If any of these

conditions are met, the fused $\mu$-op is *unfused* ❼: all associated information from the tail nucleus is invalidated, and the tail nucleus is dispatched as a standalone instruction.

### 6.1.4 Handling incorrect fusion and mispredictions within the catalyst

Once an NCF'd $\mu$-op has been validated, its corresponding tail nucleus is no longer present in the pipeline. However, if the memory addresses accessed by the fused head and tail $\mu$-ops span more than cache line size region (64 bytes), the fused pair is considered incorrect and must be re-executed. In such cases, Helios *flushes* ❽ the pipeline and refetches the original instructions.
Similarly, to recover from events originating within the catalyst—such as mispredictions (e.g. branch misprediction), exceptions, or interrupts— and to preserve memory consistency, Helios delays the commit of a NCF'd $\mu$-op until both nuclei and catalyst are ready to retire. In other words, all $\mu$-ops comprehended between the head and tail nuclei are treated as a single extended commit group to ensure precise exceptions handling.
Helios performs a precise pipeline flush starting from the faulting instruction. To support this, each $\mu$-op that can potentially cause a pipeline flush maintains two pointers to its encapsulating NCF'd $\mu$-ops, allowing the system to identify and unfuse NCF'd $\mu$-ops during recovery. These pointers, are stored in each ROB entry and, along with additional metadata such as the PC, also in a FIFO queue that tracks the state necessary for restoring correct execution after a pipeline flush.

## 6.2 Motivation

This section highlights three key limitations of the Helios instruction fusion predictor: (i) the rigidity and inefficiency of its tournament-based design, (ii) the challenges of spanning a large fusion window, and (iii) its conservative and slow training policy. These issues motivate a more compact, adaptable, and context-aware approach to fusion prediction.

### 6.2.1 Limitations of a tournament-based fusion predictor

Helios adopts a tournament-style predictor, comprising a meta-predictor or selector and two sub-predictors: a local predictor indexed solely by the PC, and a global gshare-style predictor [54] indexed by a hash of the PC and a global branch history comprised of the last nine branches. Although tournament predictors are frequently used because of their ability to reconcile differing local and global behaviors, their suitability for instruction fusion is limited.

The main limitation lies in the binary nature of the selector's choice: It must choose between no history at all and a history context that may be overfitted for instruction fusion. This design introduces unnecessary rigidity. Instruction fusion tends to exploit local patterns, that is, short-range dependencies between adjacent or nearby instructions. Our analysis shows that the average length of the catalysts is only 8 instructions in an Oracle fusion predictor (see Section 6.6 for details regarding the Oracle fusion predictor), as shown in Figure 6.2. Moreover, as shown in Figure 6.3, the average number of intervening branches between fused pairs is 1.37, suggesting that long branch histories are not only redundant but may lead to path explosion and aliasing in the global predictor table. The exception is 600.perlbench, in which taking into account up to six branches could benefit in capturing the behavior of the last 20% of non-consecutive instruction fusion.

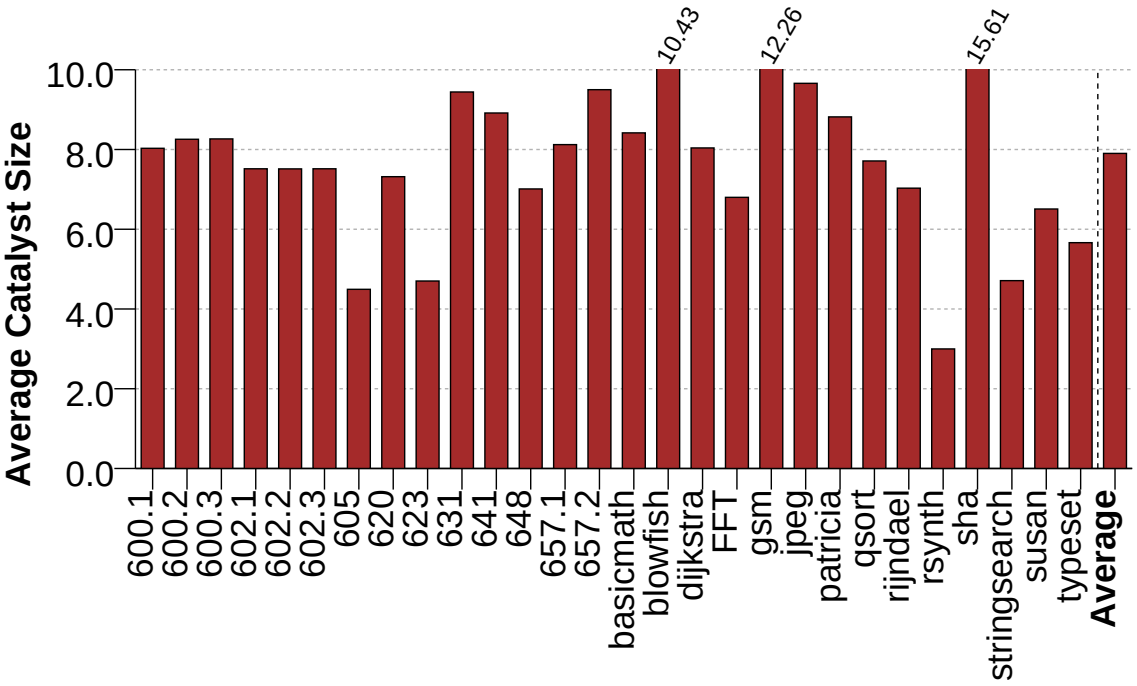



Figure 6.2: Average catalyst size in an Oracle predictor

In contrast, as illustrated in Figure 6.4, Helios must learn, on average, six different histories per fused pair, with worst-case behavior in 631.deepsjeng, where up to 17 distinct paths are observed.

Furthermore, the meta-predictor must also be trained independently, requiring repeated exposure to the same context before it can reliably choose the more appropriate sub-predictor. This increases warm-up time and hampers responsiveness to dynamic program behavior. Moreover, in Helios, a misprediction results in the invalidation of the corresponding entry, which effectively resets the learning process and undermines the benefit of using a meta-predictor. This combination of delayed adaptation and hard resets is particularly ill-suited for capturing short-lived or phase-sensitive fusion opportunities.

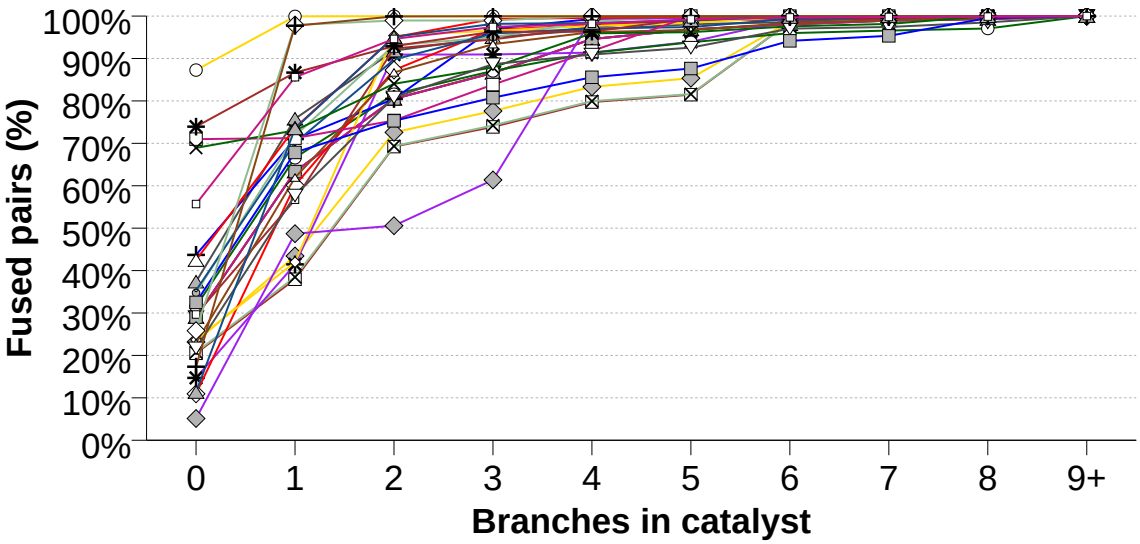



Figure 6.3: Percentage of fused pairs in an Oracle predictor classified by the number of branches in the catalyst

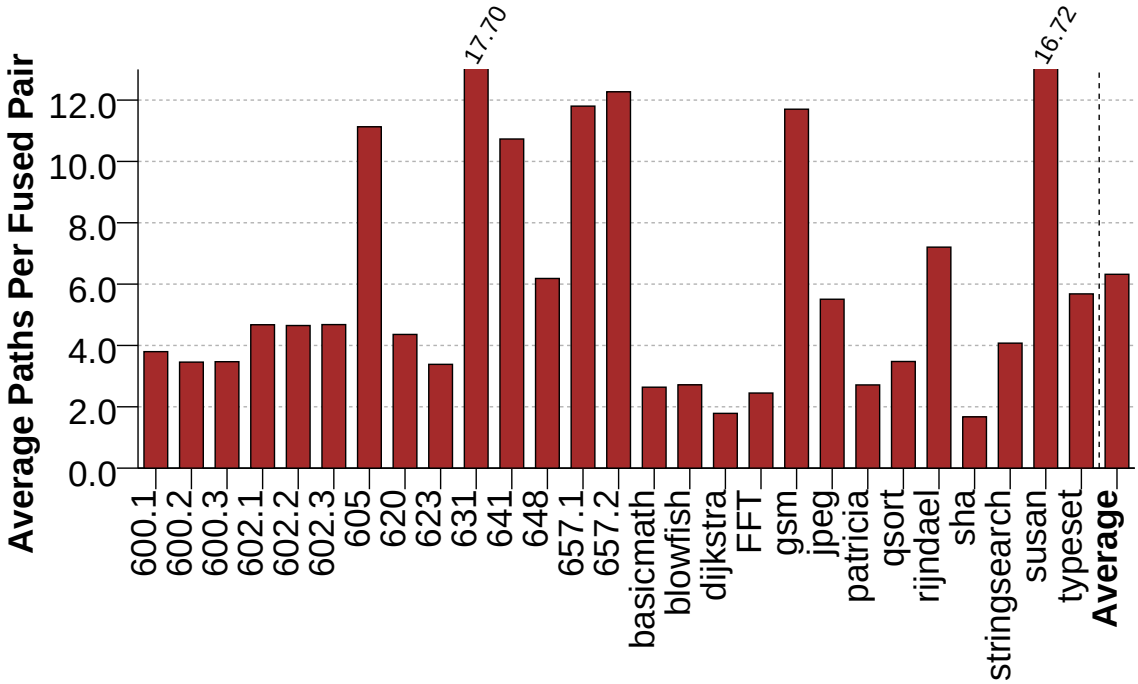



Figure 6.4: Average number of paths seen per fused pair with Helios

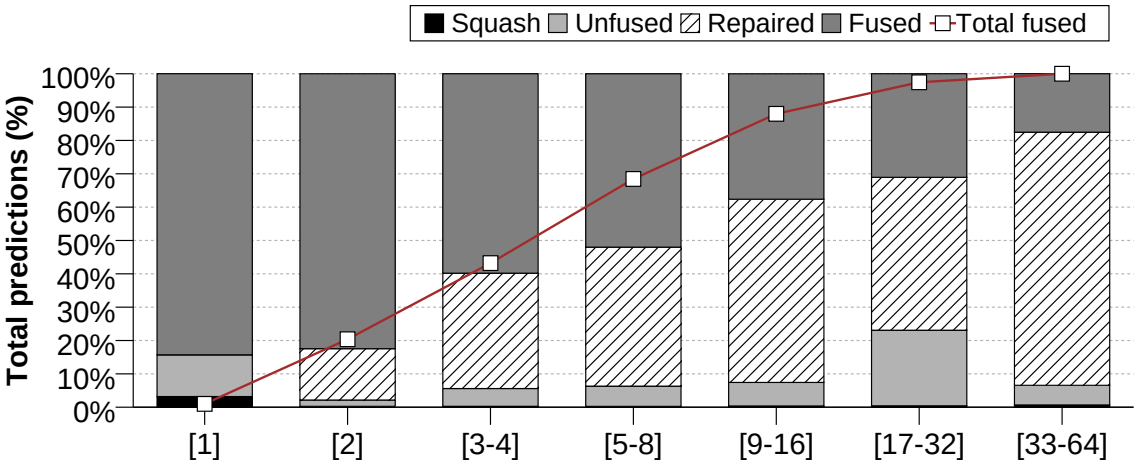



Figure 6.5: Per-distance percentages of correct predictions, predictions unfused due to cyclic dependencies, and predictions that caused a squash with Helios

### 6.2.2 Locality of Fused Instructions

Instruction fusion is most effective when the head and tail nuclei appear close together in the dynamic instruction stream. However, Helios allows to fuse instructions with a distance of up to 64 instructions. Such length is unnecessary and may lead to learn many pathological cases that will be unfused because of dependency cycles, leading to unnecessary stalls and losing fusion opportunities. Figure 6.5 quantifies this behavior by categorizing fused instruction pairs into four outcomes: *squashes:* incorrectly fused pairs that triggered a pipeline flush; *unfused opportunities:* cases where fusion was reverted due to dependency cycles; *repaired NCF'd µ-ops:* due to RaW dependencies in the catalyst; and *correctly fused pairs:* without repair. The line in the figure shows the contribution of each distance interval to the set of correct fusion predictions.

These results reinforce the intuition that short-distance fusion is significantly more reliable and effective. The majority of fusions at distances of 1–2 instructions are correct and rarely require amend. However, as the dynamic distance between the head and tail nuclei increases, the reliability of fusion drops sharply. Long-

distance fusions—those spanning more than 16 instructions—account for only 12% of total correct predictions and are highly error-prone, with up to 82% requiring repair or being undone due to cyclic dependencies. This trend correlates strongly with the length of the catalyst: as it grows, so does the probability of encountering RaW or cyclic dependencies. For catalysts of 9–16 instructions, only 40% of fusion attempts succeed without repair, a figure that drops to just 18% beyond that range.

These findings suggest that focusing on shorter-range fusion pairs is both practically effective and architecturally simpler. Long-distance fusions are not only rare in practice, but also entail higher risk and overhead for the predictor. As a result, refining the predictor to prioritize nearby fusion opportunities can improve both accuracy and efficiency. This insight motivates a reevaluation of the history length and training policies employed in Helios, which we explore in the following sections.

### 6.2.3 Revisiting the training policy

Although speculative instruction fusion offers substantial performance benefits, mispredictions are costly. Mispredictions that are detected late in the pipeline result in pipeline flushes, making accuracy essential. To avoid such penalties, Helios adopts a conservative training policy, issuing a prediction only when a confidence counter is saturated.

In practice, this means that for a two-bit confidence counter, a fusion pair must be encountered at least four times before it becomes active. Any misprediction immediately invalidates the entry. Although this policy minimizes risk, it also leads to under-utilization of profitable fusion opportunities.

However, not all fusion pairs carry equal risk, and such a conservative approach

should happen only for fusion pairs with different base registers. The rationale is that when the head and tail of the nucleus share the same base register, if no intermediate instruction modifies it, such pairs are, in general, safe to fuse.
In these cases, waiting for multiple confirmations may be unnecessary. We propose that such low-risk fusion pairs saturate their confidence counters at the first encounter. Conversely, pairs that involve different base registers should still follow the conservative path since mispredicting is always worse than not fusing.
In summary, Helios's rigid tournament structure and uniformly cautious training policy lead to inefficiencies in both predictor size and learning latency. By tailoring the history depth and training aggressiveness to the characteristics of the fusion pair, we can build a more responsive, accurate, and compact predictor.

## 6.3 The FLIP Instruction Fusion Predictor

This section introduces FLIP, a fusion predictor for load-load and store-store instruction pairs based on (i) the fact that fusion opportunities are close in space, (ii) the use of the same history length for each fusion pair, and (iii) an improved training policy.

### 6.3.1 Adaptative History Length

As established in the previous sections, instruction fusion predominantly exploits spatial locality, with most fused pairs occurring within a few dynamic instructions of one another. This locality implies that the use of long branch histories in the predictor is not only unnecessary but may also degrade performance by introducing several global histories to be learned for each fusion pair. Consequently, the history length must be carefully chosen to provide sufficient context without degrading the predictor's efficiency.

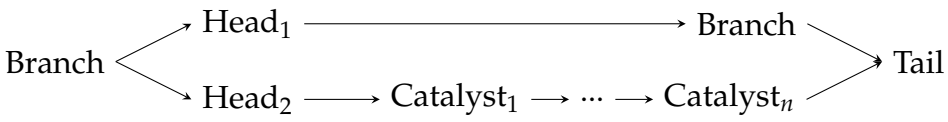



Figure 6.6: Examples of path ambiguity

Inspired by the PHAST memory dependence predictor presented in the previous chapter, we apply a similar approach in which each fusion pair is always trained with the same history length, which corresponds to the context information between the head and tail nuclei. Similarly, in refining history usage, we focus on divergent branches, that is, conditional and indirect jumps. For conditional branches, the branch direction is recorded; for indirect jumps, only a partial target address is used to encode the path signature. To further disambiguate between execution paths that converge at the same fusion region but originate from distinct control flows, we include the target of the divergent branch immediately preceding the head nucleus in the path history. This ensures that the fusion context reflects not just the fusion region itself, but also the entry path into it.
Figure 6.6 illustrates this concept. On the upper path, the catalyst has a direct branch that leads immediately into the tail nucleus. On the lower path, a longer catalyst without divergent branches, leading to path aliasing. Without recording the branch preceding the head nucleus, the predictor is unable to distinguish between these scenarios, resulting in increased mispredictions. By including that branch, the context becomes disambiguated, thereby improving prediction accuracy.
The benefits of this model are illustrated in Figure 6.7 using a representative fusion pair ($\texttt{ld}_a$, $\texttt{ld}_{a+8}$). In this example, the catalyst contains four divergent branches. By selecting the exact four branches preceding the tail nucleus, plus the one previous to the head nucleus, the predictor captures the entire control-

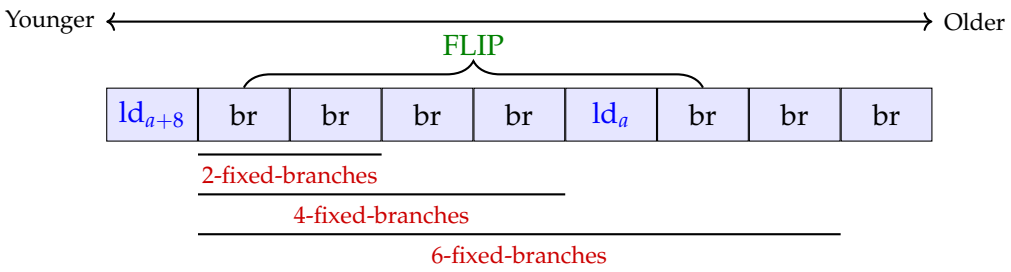



Figure 6.7: Comparison of global history selection strategy

flow path from head to tail, preserving essential context. In contrast, arbitrary fixed-length history schemes introduce several limitations: shorter histories (e.g., two branches) may omit critical context, leading to false positives that trigger squashes or, alternatively, cause fusion opportunities to be missed; intermediate lengths (e.g., four branches) may still be insufficient to disambiguate similar paths; and longer histories (e.g., six branches) introduce superfluous context, exponentially increasing the number of distinct entries (up to $2^n$ for $n$ extra branches), which leads to reduced efficiency and slower learning.

To implement this, each UCH entry can be extended with a *divergent branch commit number* ($CN_{br}$) field, indicating the most recent divergent branch observed at the time of committing each memory operation. When a match is found in the UCH, the number of divergent branches between the head and tail nuclei can be determined by subtracting their respective $CN_{br}$ values—analogous to how their fusion instruction distance is computed. Given that UCH is limited to 6 load entries and a single store entry, the storage overhead is minimal, with just $log_2(max\_branch\_tracked) + 1$ additional bits per entry.

In summary, tailoring history length to each fusion pair instance, and encoding minimal yet meaningful control-flow information, offers an effective means to enhance prediction accuracy and training speed while reducing the number of entries required.

### 6.3.2 Safe and Fast Training

As argued in Section 6.2, the training policy employed can be improved through a more nuanced approach. Specifically, we propose partitioning the training behavior based on the base register of the memory operations involved. Fusion pairs that access memory through different base registers are more susceptible to prediction errors that cause a pipeline flush and should thus be treated conservatively. Conversely, fusion pairs that share the same base register tend to be more stable, making them suitable candidates for a more aggressive training strategy.

To enable this distinction, each entry in UCH can be augmented with 5 bits to track the architectural base register. At commit time, the base register can be reconstructed from the physical register mapping before inserting the unfused load into the table. When a potential fusion opportunity is detected, the predictor compares the stored base register field: if it matches, the corresponding confidence counter is saturated; if it differs but a matching distance entry exists, the counter is incremented conservatively; otherwise, it is initialized to one.

### 6.3.3 A cost-effective FLIP implementation

Our implementation of FLIP consists of four prediction tables, each accessed in parallel on every prediction, similar to the structure of PHAST, the memory dependence predictor presented in the previous chapter.

Each table corresponds to a different control-flow history length (number of branches), selected from the set {0, 1, 2, 3}. Since the divergent branch immediately preceding the head nucleus is included, the effective history lengths range from 1 to 4. Histories exceeding this set are truncated to limit complexity and storage overhead. For instance, if a given pair has 5 branches in their catalyst, only the

4—3 plus the extra branch—closest to the tail nucleus are considered, discarding the rest.

To generate the set index and tag, the PC (after discarding its two least significant bits) is combined with the corresponding history via a hash function. Because distributing predictions based on history length is not uniform, interleaving is used to balance access frequency: the next two least significant bits of the PC select the starting table, which rotates in a round-robin fashion across the remaining tables.

Each table is 4-way associative, comprising 128 sets, for a total of 512 entries per table and 2,048 entries in total. Every entry includes: an 8-bit tag, a 5-bit distance field to encode the offset between the fused $\mu$-ops, a 2-bit confidence counter, and a 2-bit LRU field. This results in 17 bits per entry, amounting to 4.25KB of total storage. If the watchdog mechanism (Section 6.4.2) is enabled, an additional 1-bit field is appended to each entry to mark *not-worth-fusing* instruction pairs. This increases the overall storage requirement to 4.5KB.

## 6.4 Non-consecutive Fusion Optimizations (NFO)

This section introduces a set of optimizations to address performance bottlenecks and inefficiencies in non-consecutive fusion, aimed to improve overall performance and/or reduce overheads.

### 6.4.1 Speculative execution

A NCF'd $\mu$-op cannot issue until it has been validated, a process performed when the tail nucleus passes through register renaming and checks for RaW dependencies within the catalyst. If such dependencies are found, the fused instruction is repaired. This mechanism exists because repairing a NCF'd $\mu$-op

that has already executed with incorrect source register mappings for the tail is non-trivial and may lead to incorrect execution. Conversely, if the tail depends on the head nucleus, a similar mechanism allows unfusing a NCF'd $\mu$-op—to avoid introducing circular dependencies that could lead to a deadlock—by invalidating the tail nucleus information from IQ, ROB, and LSQ.

We observed that this validation introduces unnecessary stalls in cases where the fusion was, in fact, valid. If the NCF'd $\mu$-op has all its dependencies satisfied and is ready to execute, but has not yet been validated, it will be delayed until the tail nucleus is dispatched, even though it could have executed correctly.

To improve performance in this situation, we explored speculatively ignoring these stalls. A NCF'd $\mu$-op is allowed to execute as soon as it is ready, even if validation is pending. Instructions that depend on the tail nucleus will not wake up until the NCF'd $\mu$-op gets validated. If the tail nucleus later discovers a RaW dependency, it will trigger the unfusion mechanism. Note that if the tail is found to depend on the head nucleus, unfusion would have occurred regardless. If, on the other hand, the NCF'd $\mu$-op gets validated before being ready to execute, any required repair will still take place.

### 6.4.2 Identifying inefficient fusion

While fusing memory operations is generally beneficial, doing so indiscriminately can be counterproductive. Consider the code snippet illustrated in Listing 6.1, where loads [1] and [4] form a valid fusion idiom:

Listing 6.1: Example of extra delay with non-consecutive fusion

```
[1] lw      t0, 0(a0)
[2] lw      a0, 0(a2)
[3] add     a0, a0, a4
[4] lw      t1, 0(a0)
```

Although instructions [1] and [4] meet the criteria for fusion, doing so may delay execution unnecessarily. In this example, had load [1] remained unfused, it could have been issued and executed early. However, as part of a fused $\mu$-op, it becomes dependent on instructions [2] and [3], which form the catalyst. Even though the fused load reduces ROB and LQ pressure, this delay undermines that advantage by increasing the time the instruction—and all its dependents—remain occupying entries in the IQ, ROB, and LQ, especially if instruction [1] lies on the critical path or is part of a loop-carried dependency. The result is increased pipeline congestion and a degradation in overall performance. Moreover, when the head and tail nuclei are close, such as this example, the speculative execution mechanism described in the prior section is insufficient to recover performance, as the head cannot break free from the fusion-induced delay.

To address such cases where fusion-induced delays congest the pipeline, we introduced a *watchdog* mechanism. The watchdog resides in the issue stage and tracks the readiness imbalance between fused instructions. Specifically, it consists of a saturating counter that is probabilistically incremented each cycle when the head's dependencies are ready, but the fused $\mu$-op is still stalled due to tail dependencies. Empirically, we found that a 3-bit counter, incremented with a probability of $\frac{1}{32}$ per cycle under these conditions, provides a good trade-off between detecting harmful fusions and avoiding false positives. Once the counter saturates, the fusion predictor is updated to discourage future fusion of that specific pair. This amounts to 1164 (1.14Kbits) additional storage bits (3 bits per IQ entry). A similar mechanism exists in Intel Core's smart memory disambiguation [22].

In addition to the watchdog, the fusion predictor can also be updated whenever a fused pair is unfused due to validation failure to prevent wasting resources on ineffective fusion attempts.

### 6.4.3 Revisiting fusion priority

Non-consecutive fusion occurs opportunistically during the decode stage when a valid instruction pair is identified. Once a $\mu$-op is fused with an older, non-consecutive pair such as instruction i with i-j, where j>1, the fusion decision is irreversible.

An alternative policy to the current strategy would allow overriding a non-consecutive fusion decision when a consecutive fusion opportunity arises. Specifically, if instruction i has already been fused with a distant predecessor i-j (where j>1), but the subsequent instruction i+1 forms a valid CSF idiom with i, the system could reassign the fusion, unfusing the NCF'd $\mu$-op in favor of the consecutive pair (i,i+1).

### 6.4.4 Removing nested fusion

Non-consecutive fusion permits a certain degree of nesting, where multiple NCF'd $\mu$-ops become entangled. However, we argue that allowing such nesting is often counterproductive for two key reasons: (i) it can increase execution latency by introducing complex dependency chains between fused instructions, and (ii) it does not necessarily improve the overall number of fused instruction pairs. To illustrate the first point, consider the code in Listing 6.2:

Listing 6.2: Dependency chain between nested non-consecutive fused pairs

```
[1] ld      r1, 0(r2)
[2] ld      r3, 0(r4)
[3] ld      r4, 8(r3)
[4] ld      r5, 8(r6)
```

In this sequence, instructions [1] and [3] are fused into a NCF'd pair, and so are [2] and [4]. However, since instruction [3] has a data dependency on [2], this causes [1], the head of the first fusion, to become dependent on [2] and any dependencies it carries. As a result, fusion extends the dependency chain and increases contention in the execution window. For the second point, consider the example in Listing 6.3:

Listing 6.3: Example where non-nested fusion does not necessarily reduce the total number of fused pairs

```
[1] ld      r1, 0(r2)
[2] ld      r3, 0(r4)
[3] ld      r4, 8(r3)
[4] sd      r6, 0(r7)
[5] ld      r5, 8(r4)
[6] sd      r8, 8(r7)
```

This code contains three valid NCF idioms: ([1], [3]), ([2], [5]), and ([4], [6]). Under a design that allows 2-level nesting, only two pairs—([1], [3]) and ([2], [5])—are fused. The third pair ([4], [6]) will be treated as unfused.
In contrast, if nested NCF'd $\mu$-ops is not allowed, then the pair ([2], [5]) is not fused, and ([4], [6]) becomes available to be fused instead. The total number of fused pairs remains the same in both cases.

### 6.4.5 Filtering the Unfused Committed History

The Unfused Committed History is a small, fully-associative structure that holds the six most recent committed, unfused load $\mu$-ops, and serves as the primary mechanism for training the Fusion Predictor. As such, inserting low-quality or irrelevant entries into the UCH can lead to polluting the predictor with wrong patterns.

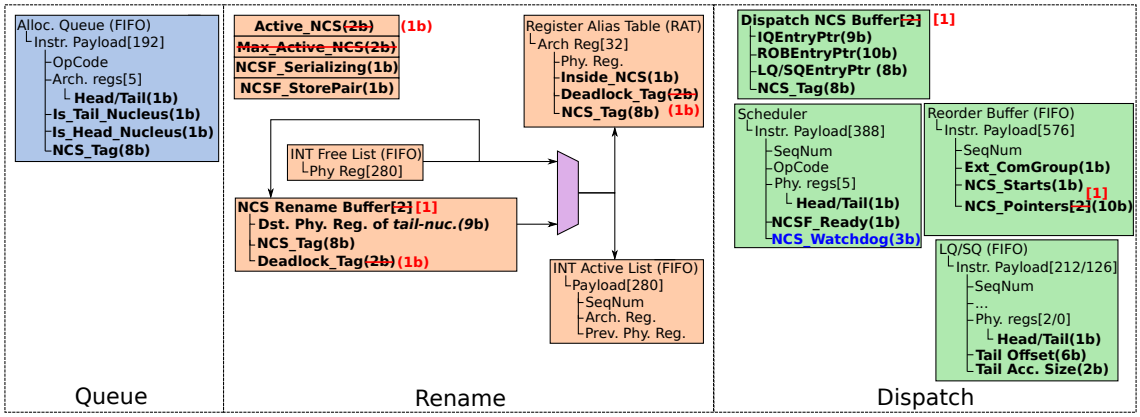



Figure 6.8: Key pipeline structure changes for Helios (bold), and modifications for FLIP (red and blue)

Each UCH entry records only the tag of the base address and the size. Crucially, register operand information is not retained. This can lead to incorrect learning. For example, suppose a load such as `ld a5, 0(a5)` is inserted into the UCH. If a subsequent load like `ld s2, 0(a5)` snoops the UCH and matches the address tag, the fusion predictor is incorrectly trained to believe that the two loads form a valid fusion idiom.

As a result, when the first load is decoded again and fused based on the predictor's suggestion, it will stall at issue, waiting for the tail to validate the fusion. Upon validation, the tail detects the dependency cycle, a deadlock is uncovered, and the pair must be unfused. If the system is already at its maximum allowed level of nested NCF'd $\mu$-ops, this incorrect fusion can also block valid fusions from forming, hurting overall performance.

To mitigate this issue, we propose filtering out read-modify loads—i.e., loads where the destination and base registers are the same—when populating the UCH to prevent such problematic patterns from corrupting the fusion predictor.

## 6.5 Storage Requirements in Helios and FLIP

Figure 6.8 illustrates the architectural modifications required to support non-consecutive load-load or store-store fusion in Helios. The components modified or added to enable fusion are highlighted in bold. In its original design, Helios allocates 2.59KB for various pipeline structures supporting NCF and an additional 9KB for the fusion predictor, bringing the total hardware budget to 11.59KB.
By replacing the Helios fusion predictor with FLIP, the predictor's storage footprint is reduced from 9KB to 4.25KB (Section 6.3.3), bringing the total to 6.84KB. Moreover, if nesting of NCF'd $\mu$-ops is disallowed, the pipeline can be significantly simplified. Several structures that previously scaled with the maximum nesting level can be reduced to a single entry, and many per-instruction counters and bit-vectors can be replaced with 1-bit flags. Additionally, to maintain precise exceptions and branch misprediction recovery, only a single NCF pointer is needed per ROB entry. These simplifications, indicated in red in Figure 6.8, reduce the pipeline structure budget from 2.59KB to 1.88KB.
Lastly, enabling the watchdog mechanism (Section 6.4.2) requires 3 bits per IQ entry, totaling 1164 bits or approximately 0.14KB. These additions are colored blue in Figure 6.8.

## 6.6 Evaluation

In this section, we are going to evaluate the performance of FLIP compared against consecutive fusion with same base register (CISSR), Helios, and an oracle predictor as baseline. The study is carried out simulating a Lunar Lake like processor running the SPEC CPU 2017 (speed suite) and MiBench suites. Refer to chapter 3 for the detailed methodology.

The fusion mechanisms evaluated in this work include: *CISSR* for Consecutive Instruction with Same Source Register, which fuses consecutive instructions sharing the same base register, where the accesses may overlap or be assymetric (different access sizes); *Helios*; our implementation of *FLIP* without NFOs; an *NFO-enhanced version of FLIP*; and a final configuration of *NFO'd FLIP with nesting disabled*, which restricts the system to not interweave NCF'd $\mu$-ops. In addition, we evaluate an *Oracle* predictor that knows in advance the effective addresses of all memory operations and whether fusion-preventing elements are present in the catalyst. However, this Oracle is not ideal: it lacks awareness of dynamic constraints such as excessive delays due to dependencies.

This section is divided into three parts. First, we execute Helios under shorter history lengths. Although the original design used a 9-branch history—given that we advocate for shorter histories—we evaluate shorter histories to determine whether they offer further improvements in Helios. Then, we evaluate the Non-consecutive Fusion Optimizations introduced in Section 6.4 on top of FLIP. Lastly, we proceed to compare the FLIP using those optimizations against the other fusion mechanisms, using CISSR as the baseline.

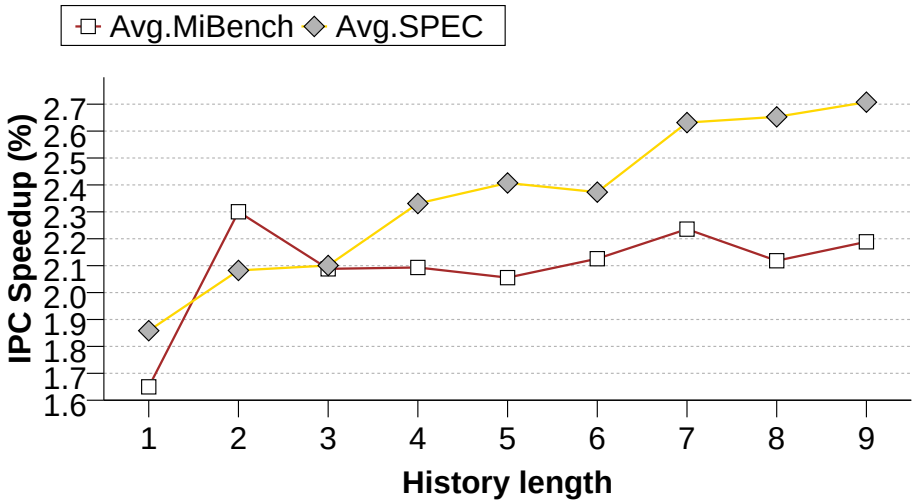



Figure 6.9: Average speedup for Helios with different history lengths over CISSR

### 6.6.1 Impact of History Length on Helios

Since in this work we advocate for shorter history lenghts, we evaluate the impact of history length on Helios by sweeping history sizes from 1 to 9 branches while keeping all other predictor parameters constant. Figure 6.9 reports the IPC speedup of each configuration relative to CISSR.

For the SPEC CPU 2017 suite, decreasing the history length consistently degrades IPC, with a pronounced fall at a history size of 6 branches, and 3 branches. We did not observed notable improvements on 10 branches and forward.

In contrast, for the MiBench suite, the effect of history length is less pronounced, with most configurations showing minimal performance variation—except for the 1-branch history, which performs noticeably worse. An exception occurs at a history length of 2 branches, which yields a noticeable performance increase driven primarily by the *blowfish* benchmark. In this case, Helios with a 2-branches history favors non-consecutive fusion of stores over loads: compared to the 9-branches configuration, the 2-branch version exhibits 9% fewer non-consecutive load fusions and 33% more non-consecutive store fusions (resulting in an overall

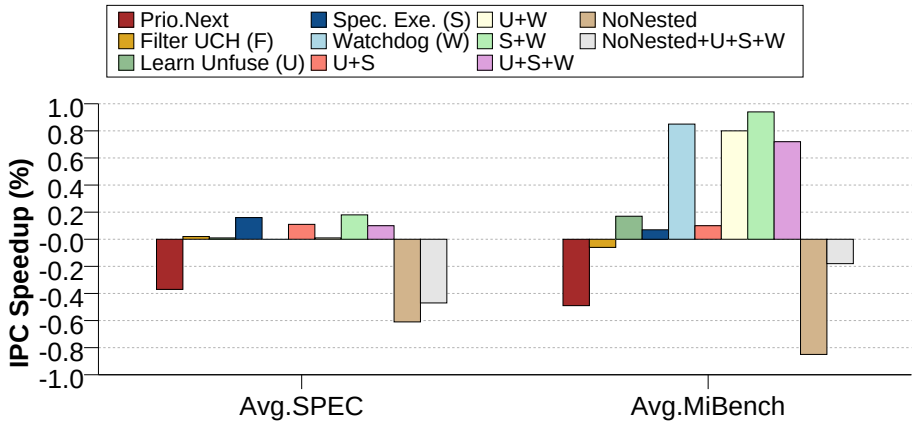



Figure 6.10: Average speedup for the FLIP variants over FLIP

2.8% decrease in total non-consecutive fusions, as most fused instances involve loads).

These results highlight that (1) short histories are not suitable for a tournament-like predictor, and (2) that a fixed history length is not optimal across all workloads. Instead, they suggest the need for a more adaptive approach capable of dynamically adjusting the effective history length based on program behavior. Nevertheless, for the remainder of this evaluation, we use the 9-bit history configuration of Helios, as it provides the best overall performance across the benchmark suites.

### 6.6.2 Non-consecutive Fusion Optimizations

Figure 6.10 reports the performance improvements (in percentage) over FLIP for various non-consecutive fusion optimizations evaluated in this work:

- *Prio. Next:* If instruction *i* is fused with a non-consecutive older instruction *(i–j)*, but *(i+1)* is a valid consecutive fusion candidate with *i*, the non-consecutive pair is unfused in favor of the consecutive pair *(i, i+1)*.
- *Filter UCH (F):* Prevents loads that overwrite their base register from being inserted into the UCH.

- *Learn Unfuse (U):* Marks entries that resulted in an unfusion event as not worth fusing.
- *Speculative Execution (S):* Allows NCF'd $\mu$-ops to execute without waiting for validation if their dependencies are resolved.
- *Watchdog (W):* Identifies fusion pairs where the head nucleus is excessively delayed due to tail dependencies.
- *NoNested:* Disables nested non-consecutive fusion.

From the results, we observe that prioritizing the consecutive pair *(i, i+1)* over a non-consecutive fusion *(i–j, i)* is detrimental, reducing average performance by 0.4% on both SPEC and MiBench.

Filtering the UCH yields a slight improvement in SPEC but negatively affects MiBench, suggesting the need for a more selective filtering policy. Marking unfused pairs has limited benefit in just some of the SPEC benchmarks (e.g., +0.12% in *600.perlbench_3*) but provides more noticeable gains in MiBench (e.g., *blowfish* +2.9% and *patricia* +0.13%). Similarly, the watchdog mechanism has no effect in SPEC but proves highly beneficial for MiBench, especially in *blowfish*, which we analyze in the next subsection. Speculative execution shows consistent benefits across both suites, improving performance by almost 0.2% in SPEC and 0.1% in MiBench. However, combining speculative execution with unfuse learning yields diminishing returns. This is due to cases where speculative execution triggers fusion, but later uncovers RaW violations, causing those pairs to be misclassified as "not-worth" in the predictor. For instance, in *susan*, the number of fused NCF pairs dropped by 18.5% when both optimizations were applied together.

In contrast, combining speculative execution with the watchdog mechanism results in complementary benefits, since speculative execution primarily aids long-distance fusion, while the watchdog mechanism helps with short-range

Table 6.1: Predictor descriptions and storage budgets (KB)

| Predictor | NFOs | $\mu$arch budget / Predictor budget | Total budget |
|---|---|---|---|
| **Helios** | | 2.59 / 9 | **11.59** |
| **FLIP** | | 2.59 / 4.25 | **6.84** |
| **FLIP-S-W** | spec. exec. watchdog | 2.73 / 4.5 | **7.23** |
| **FLIP-NoNested -S-U-W** | no nesting spec. exec. mark unfused watchdog | 2.02 / 4.5 | **6.52** |

cases that were repaired. This synergy yields a nearly 1% speedup in MiBench, driven largely by improvements in *blowfish*.

Disabling nested fusion reduces performance by 0.61% in SPEC and 0.85% in MiBench. Nevertheless, combining speculative execution, the watchdog, and unfuse learning substantially mitigates this loss, narrowing the performance gap to 0.47% in SPEC and just 0.18% in MiBench.

For the remainder of the evaluation, we compare the CISSR model against the predictors summarized in Table 6.1, along with an Oracle predictor that perfectly fuses all eligible memory pairs.

### 6.6.3 Evaluation of the fusion predictor

#### 6.6.3.1 Performance and bottlenecks

Figure 6.11 shows the speedup over CISSR for Helios, the different configurations of FLIP, and the Oracle predictor. On average, FLIP outperforms both CISSR and Helios, with average speedups of 5.04% and 2.27% on SPEC, and 4.33% and 2.09% on MiBench, respectively. However, FLIP exhibits performance degradation in a

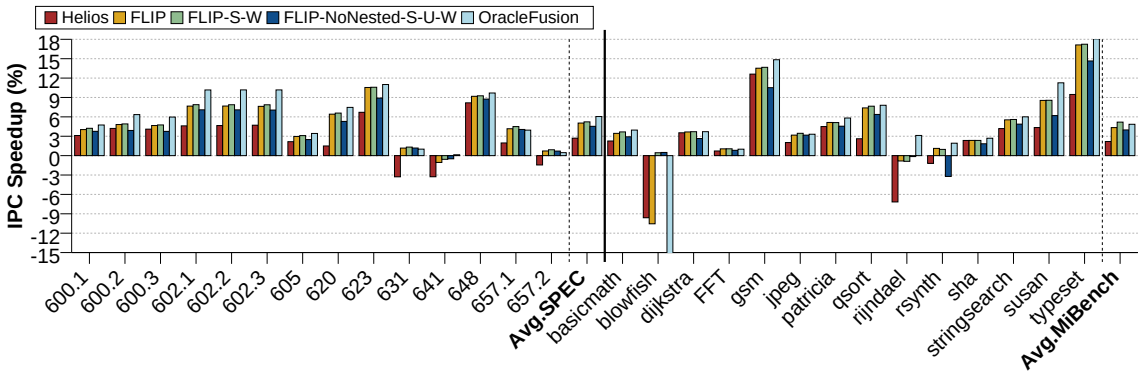



Figure 6.11: Speedup over CISSR for Helios, FLIP variants, and Oracle

few benchmarks: *641.leela*, *blowfish*, and *rijndael*.

In *641.leela*, the performance loss is due to pointer-chasing behavior. The memory addresses involved are difficult to predict and often not close enough for effective fusion, which leads to frequent pipeline flushes when the head and tail nuclei are fused incorrectly.

A similar issue appears in *rijndael*, which shows the highest number of incorrectly fused NCF'd $\mu$-ops causing flushes. This is primarily because many base addresses in rijndael are passed as function parameters. When these addresses happen to be close during training, they can incorrectly train the fusion predictor, which then predicts fusion in later iterations.

*blowfish* presents a unique case: a tight loop (4–5 lines in C, roughly 20–30 RISC-V instructions) with loop-carried dependencies and memory dependencies every 8 iterations. The binary generated by the compiler relies heavily on stack operations, resulting in inefficient code where base addresses are repeatedly loaded from and stored back to the stack on every iteration. From the small representative code fragment from *blowfish* presented in Listing 6.4, where instructions [1] and [6] are a valid NCF pair:

Listing 6.4: Blowfish example where NCF delays head nucleus execution

```
[1] sb      a5, -57(s0)
[2] mv      a4, s2
[3] ld      a5, -56(s0)
[4] add     a5, a5, a4
[5] lbu     a5, 0(a5)
[6] sb      a5, -58(s0)
```

Fusing stores [1] and [6] delays [1], as it inherits all the dependencies of [6]. This forces [1] to wait until all instructions in the catalyst have completed, delaying memory-dependent loads and causing a snowball effect. Eventually, the reorder buffer and issue queue fill up, stalling the front-end. This demonstrates that aggressive fusion is not always beneficial.

#### 6.6.3.2 Combining the fusion predictor and the NFOs

Improvements to FLIP using NFOs led to only marginal gains on SPEC, as already illustrated in the previous section. However, the watchdog mechanism was crucial in addressing the performance problems in *blowfish*, providing substantial benefits where aggressive fusion caused stalls.
Disabling nested fusion, combined with NFOs, produced performance similar to base FLIP. While reducing fusion opportunities, this simplification reduced pipeline storage overhead by around 25% and eliminated the need for the additional complexity and logic required by Helios-like nested fusion.

#### 6.6.3.3 Fusion pairs and paths detected

Figure 6.12 shows the percentage of memory $\mu$-ops fused. As observed by Singh et al. [86], Helios tends to favor consecutive fusion because it is performed

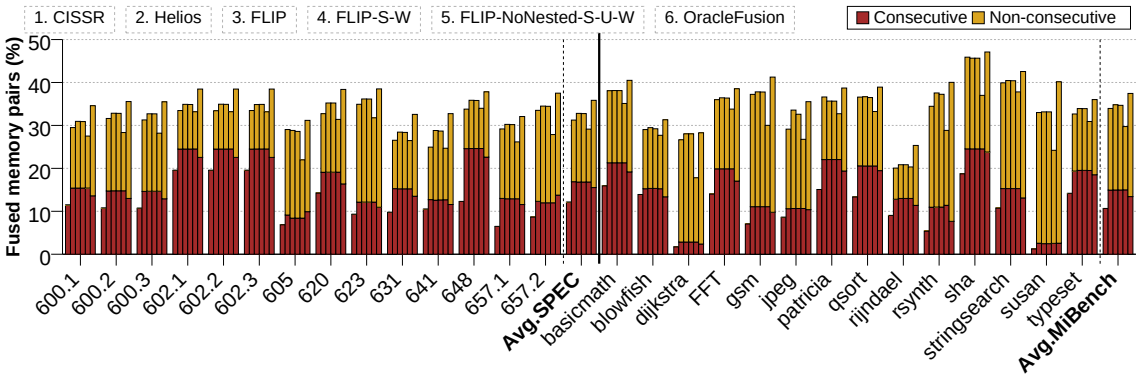



Figure 6.12: Percentage total dynamic memory instructions with CSF and NCF for CISSR, Helios, FLIP variants, and Oracle

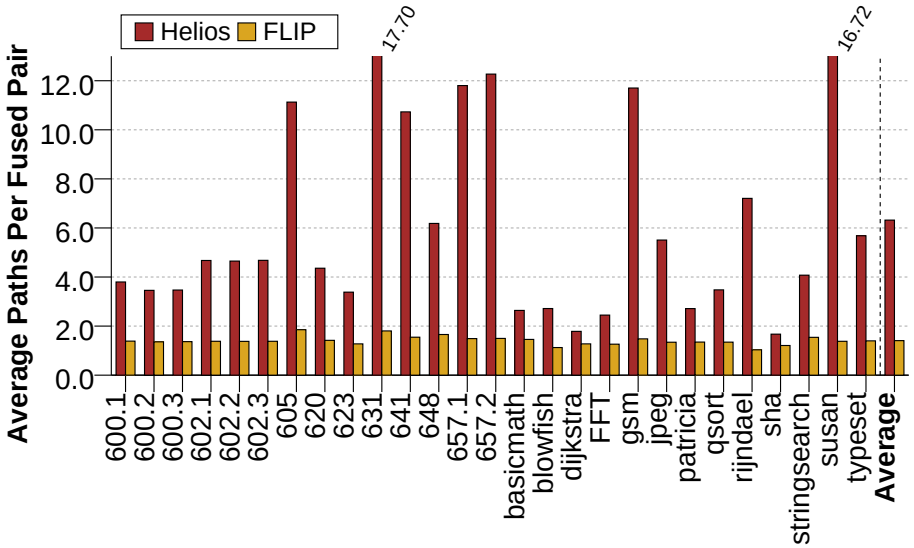



Figure 6.13: Average number of paths seen per fused pair in Helios and FLIP

directly at decode, without requiring prediction or training. In contrast, non-consecutive fusion relies on a predictor that is trained later at commit using unfused $\mu$-ops. As a result, consecutive fusion is always applied when possible, while non-consecutive fusion depends on the availability of leftover unfused $\mu$-ops and is therefore less aggressively exploited. The same is observed for FLIP. Non-consecutive fusion effectively doubles the number of fused memory operations, although the delays and overheads of NCF'd $\mu$-ops limit performance

Table 6.2: Fusion predictor coverage (%) vs. Oracle and MPKI from flush-inducing mispredictions

| Benchmark | Helios | FLIP | FLIP-S-W | FLIP-NoNested -S-U-W |
|---|---|---|---|---|
| *600.1* | 67.06 / 0.132 | 73.93 / 0.045 | 73.62 / 0.045 | 57.58 / 0.035 |
| *600.2* | 74.80 / 0.088 | 80.01 / 0.020 | 79.85 / 0.020 | 60.07 / 0.016 |
| *600.3* | 73.64 / 0.087 | 79.80 / 0.024 | 79.72 / 0.024 | 59.80 / 0.019 |
| *602.1* | 56.09 / 0.336 | 65.05 / 0.054 | 64.17 / 0.054 | 53.63 / 0.046 |
| *602.2* | 55.85 / 0.335 | 65.25 / 0.054 | 64.37 / 0.053 | 53.66 / 0.046 |
| *602.3* | 56.12 / 0.336 | 64.76 / 0.053 | 64.25 / 0.053 | 53.57 / 0.046 |
| *605* | 89.54 / 0.567 | 88.24 / 0.065 | 83.20 / 0.049 | 55.91 / 0.056 |
| *620* | 61.63 / 0.222 | 72.71 / 0.013 | 72.08 / 0.013 | 54.99 / 0.011 |
| *623* | 82.63 / 0.193 | 87.12 / 0.006 | 86.92 / 0.006 | 71.09 / 0.007 |
| *631* | 58.72 / 0.703 | 68.20 / 0.020 | 67.27 / 0.020 | 57.50 / 0.017 |
| *641* | 58.25 / 0.705 | 76.06 / 0.173 | 73.82 / 0.173 | 55.27 / 0.161 |
| *648* | 60.94 / 0.143 | 73.80 / 0.017 | 70.06 / 0.017 | 58.62 / 0.015 |
| *657.1* | 77.17 / 0.357 | 82.12 / 0.008 | 80.58 / 0.008 | 61.81 / 0.007 |
| *657.2* | 80.82 / 0.399 | 84.49 / 0.019 | 83.72 / 0.021 | 59.21 / 0.023 |
| *basicmath* | 78.84 / 0.060 | 79.01 / 0.000 | 78.41 / 0.000 | 64.45 / 0.000 |
| *blowfish* | 76.77 / 0.086 | 78.90 / 0.001 | 77.44 / 0.001 | 69.35 / 0.000 |
| *dijkstra* | 91.47 / 0.001 | 96.78 / 0.000 | 96.58 / 0.000 | 57.36 / 0.000 |
| *FFT* | 75.37 / 0.072 | 77.35 / 0.065 | 74.45 / 0.065 | 62.83 / 0.065 |
| *gsm* | 83.06 / 0.032 | 84.89 / 0.000 | 84.66 / 0.000 | 60.04 / 0.000 |
| *jpeg* | 73.63 / 0.118 | 91.16 / 0.018 | 87.08 / 0.018 | 63.92 / 0.017 |
| *patricia* | 76.18 / 0.034 | 71.44 / 0.000 | 67.56 / 0.000 | 53.00 / 0.000 |
| *qsort* | 83.84 / 0.286 | 84.21 / 0.000 | 74.85 / 0.000 | 59.56 / 0.000 |
| *rijndael* | 49.44 / 1.650 | 53.44 / 0.581 | 53.41 / 0.581 | 49.89 / 0.456 |
| *rsynth* | 72.62 / 0.023 | 82.09 / 0.000 | 81.12 / 0.000 | 53.89 / 0.000 |
| *sha* | 91.67 / 0.000 | 90.65 / 0.000 | 90.65 / 0.000 | 53.49 / 0.000 |
| *stringsearch* | 83.66 / 0.188 | 85.46 / 0.001 | 85.32 / 0.001 | 76.47 / 0.001 |
| *susan* | 80.50 / 0.368 | 80.99 / 0.093 | 79.98 / 0.092 | 56.49 / 0.080 |
| *typeset* | 76.89 / 0.384 | 83.12 / 0.007 | 77.66 / 0.007 | 61.61 / 0.005 |
| **Average** | **73.12 / 0.282** | **78.61 / 0.048** | **76.89 / 0.047** | **59.11 / 0.040** |

gains. Interestingly, disabling nested fusion reduces the overall fusion rate, but with negligible performance loss.

Figure 6.13 shows the average number of paths detected per unique fused pair by FLIP. The average remains below 2 across all benchmarks, contributing to its lower storage requirements and better performance.

#### 6.6.3.4 Coverage and MPKI

Finally, Table 6.2 summarizes the CIDBR (Contiguous Instruction with Different Base Register) and NCF coverage relative to the Oracle predictor for Helios and FLIP, as well as the MPKI for mispredictions that caused a pipeline flush. On average, FLIP is able to increase by 5.5% the coverage of Helios, with the same configurations. Additionally, FLIP, while having more coverage and half the storage of the tournament predictor in Helios, reduces the MPKI by 83%.

## 6.7 Related work

Many predictors in the literature leverage histories of varying lengths to improve accuracy in speculative execution. A prominent example is TAGE [78,79,81–84], which indexes multiple tagged prediction tables using exponentially increasing history lengths. This organization allows TAGE to capture both short- and long-range control-flow correlations, and to blend predictions based on availability and confidence. TAGE has been adapted for other domains, including speculative memory bypass [57,68], memory dependence prediction [69], and value prediction [70].

Regarding coalescing memory operations, prior work has also explored mechanisms to coalesce memory operations and reduce cache pressure. Olukotun et al. [94] propose leveraging the access FIFO in front of the L1D to match multiple load addresses against an incoming cache line, serving them in parallel. Rivers et al. [73] take a similar approach to multiporting the L1D by attaching line buffers to each cache bank, enabling multiple loads per cycle from recently accessed lines. Along the same line, Baoni et al. develop *Fat Loads* [13], which prefetch an entire cache line into a dedicated buffer to accelerate subsequent accesses to the same

region. Additional strategies include always reading double words regardless of the requested size [38], extending load/store lifetimes post-retirement to expand forwarding opportunities [11, 62, 68], and coalescing committed non-consecutive stores while preserving total store order [75].

These techniques operate independently of instruction order, data dependencies, or register reuse. Furthermore, they do not require changing $\mu$-op formats, thus, they do not reduce pressure in the pipeline structures. In contrast, instruction fusion collapse two $\mu$-ops into a single pipeline entity, directly reducing pressure on structures like the ROB, IQ, LQ, and SQ.

Finally, concerning instruction fusion, it has also been explored to simplify scheduling and improve efficiency. Kim and Lipasti [44] propose macro-op scheduling, where pairs of $\mu$-ops are treated as a single unit in the IQ to reduce select logic complexity. While this improves IQ throughput, the fused $\mu$-ops still execute as separate instructions. Thakker et al. [91] are more aggressive, yet still avoid fusing when RaW or WaR hazards exist in the catalyst. Celio et al. [17] advocate for fusion in RISC-V designs to boost "work per $\mu$-op" rather than extending the ISA. Their approach fuses load and stores pairs, but is limited to consecutive and contiguous access.

## 6.8 Conclusion

This chapter presents FLIP, a lightweight, path-based predictor for non-consecutive memory fusion that significantly improves upon the state-of-the-art Helios predictor. By leveraging the control flow between memory instructions, FLIP achieves better performance while halving the storage requirements, proving that the claims presented in the previous chapter for PHAST are not only for memory dependence prediction, but for other domains such as instruction fusion prediction.

The proposed optimizations address key challenges in non-consecutive fusion, including speculative execution of fused pairs, dynamic filtering of harmful patterns, and tracking fusion effectiveness through simple heuristics like watchdog mechanisms and unfuse learning. These optimizations enable FLIP to mitigate performance pitfalls from aggressive fusion, such as pipeline flushes or unnecessary stalls.

FLIP improves performance over Helios by 2.44% on SPEC CPU 2017 and 2.94% on MiBench, and reduces MPKI by 83%, while consuming only 62.38% of Helios' storage. Additionally, we demonstrate that disabling nested fusion, which simplifies implementation and reduces overhead, has minimal impact on performance. These results indicate that fixed-history-length fusion predictors can be both effective and practical, and that aggressive fusion mechanisms must be carefully balanced with runtime adaptability.

Chapter 7

# Conclusions and Future Ways

## 7.1 Conclusion

Despite decades of progress in processor microarchitecture, extracting high instruction-level parallelism remains challenging due to two persistent obstacles: uncertainty in memory dependencies and redundancy in memory instruction execution. Out-of-order processors rely on speculation to hide latencies and exploit parallelism, but incorrect speculation can incur costly squashes, wasted energy, and reduced overall efficiency.

Memory operations are particularly problematic because their dependencies are often unknown at rename time, forcing processors to either predict conservatively—sacrificing parallelism—or speculate aggressively—risking mis-speculation. At the same time, many memory access sequences are executed redundantly, missing opportunities for fusion that could simplify execution and reduce pressure on backend resources.

This thesis addresses these challenges through these contributions:

- **PHAST,** a novel memory dependence predictor that links the load with a

single store, and

- **FLIP,** an instruction fusion predictor for load-pairs and store-pairs.

Throughout this thesis, we have proposed a new mechanism to train the predictor based on the principle that selecting the appropriate history length for each case is enough for accurate prediction. When compared with state-of-the-art predictors, our mechanism resulted in higher performance and less storage requirements. Both PHAST and FLIP are guided by this principle. In PHAST, this history corresponds to the divergent branches between a conflicting store and load, plus the branch immediately preceding the store. In FLIP, the history length is defined by the divergent branches between the head and tail nuclei, along with the branch preceding the head nucleus.

For memory dependence prediction (Chapter 5), PHAST was shown to outperform all state-of-the-art predictors while requiring less storage. Even when PHAST's storage budget was halved to 7.25 KiB, it continued to surpass the best existing predictor, NoSQ, which requires 19 KiB.

For instruction fusion (Chapter 6), this thesis proposed **Non-consecutive Fusion Optimizations (NFO)** to either relax constraints of the state-of-the-art—Helios—or to prevent harmful fusion patterns from recurring. Coupled with these optimizations, FLIP achieves strong improvements in IPC while requiring only half the storage of the Helios fusion predictor. Even when accounting for the additional storage needed by NFOs, the combined design uses just two-thirds of the total storage required by Helios, while delivering superior performance.

## 7.2 Future works

The prediction training mechanism presented in this thesis proved effective for both memory dependence prediction and non-consecutive instruction fusion. A

natural extension is to explore its applicability to other prediction domains.

While PHAST and FLIP demonstrate strong improvements in memory dependence prediction and instruction fusion, respectively, several promising research directions remain.

For PHAST, three avenues stand out:

- *Speculative memory bypassing*. Extending PHAST to support store-load and load-load speculative bypassing could enable more aggressive latency hiding while preserving correctness. Integrating bypassing decisions into PHAST's framework may unlock additional performance with modest overhead.
- *Pathological access patterns*. Although PHAST achieves state-of-the-art accuracy, its coverage is still challenged by cases such as pointer-chasing or irregular, randomized memory accesses (e.g., arrays of pointers). Improving coverage and accuracy in such scenarios remains an important goal, potentially requiring hybrid schemes that combine PHAST with complementary predictors.
- *Software–hardware co-design.* Compiler hints identifying loads likely to never—or always—conflict could simplify PHAST by focusing hardware prediction on the more uncertain cases. Moreover, if the compiler identifies the most relevant branches for each conflicting load, PHAST's accuracy could improve while further reducing hardware complexity.

For FLIP, further opportunities include:

- *Improving coverage*. FLIP delivers strong performance gains, but its coverage still falls short of the oracle predictor. Extending the predictor to capture a wider variety of profitable fusion cases is a natural next step.
- *Pointer-chasing workloads*. Like PHAST, FLIP struggles with pointer-heavy or irregular memory patterns. Enhancing its ability to recognize and

exploit fusion opportunities in these contexts could further improve its effectiveness.

In summary, future work includes both broadening the applicability of the proposed training mechanism to other prediction domains, and deepening the exploration of PHAST and FLIP. Extending these predictors to more challenging scenarios, integrating compiler support and hybrid mechanisms, and refining their adaptability could further enhance robustness, coverage, and efficiency in speculative execution.

# Bibliography


[1] Scarab. https://github.com/hpsresearchgroup/scarab. 3.1.2

[2] PCACTI. https://sportlab.usc.edu/downloads/packages, 2021. 3.4, 3.5

[3] ChampSim simulator. http://github.com/ChampSim/ChampSim, May 2022. 3.1.2

[4] Advanced Micro Devices. Software Optimization Guide for AMD EPYC™ 7003 Processors, Pub 56665, Rev 3. [Online; accessed Apr.-2022]. 6

[5] Advanced Micro Devices, White paper. *Security Analysis of AMD Predictive Store Forwarding*, August 2023. 2.1.2.2

[6] Advanced Micro Devices Inc. Software optimization guide for the amd zen4 microarchitecture, April 2023. 2.3.2

[7] Advanced RISC Machines. Arm® Cortex™-A77 Core Software Optimization Guide, Issue 3. [Online; accessed Apr.-2022]. 6

[8] Advanced RISC Machines. Arm® Neoverse™-N2 Core Software Optimization Guide, issue 3. [Online; accessed Apr.-2022]. 6

[9] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *Int'l*

*Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42, April 2009. 3.2

[10] Muawya M. Al-Otoom, Conrado Blasco, Deepankar Duggal, Ethan R. Schuchman, Ian D. Kountanis, Kulin N. Kothari, and Nikhil Gupta. Program counter zero-cycle loads, April 2025. Apple Inc. 2.2.2.4

[11] Ricardo Alves, Alberto Ros, David Black-Schaffer, and Stefanos Kaxiras. Filter caching for free: The untapped potential of the store buffer. In *46th Int'l Symp. on Computer Architecture (ISCA)*, pages 436–448, June 2019. 5.4, 6.7

[12] Jean-Loup Baer. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge University Press, 1st edition, 2009. 1.1

[13] Vanshika Baoni, Adarsh Mittal, and Gurindar S. Sohi. Fat loads: Exploiting locality amongst contemporaneous load operations to optimize cache accesses. In *54th Int'l Symp. on Microarchitecture (MICRO)*, pages 366–379, 2021. 6.7

[14] Alper Buyuktosunoglu, Ali El-Moursy, and David H. Albonesi. An oldest-first selection logic implementation for non-compacting issue queues. In *15th Annual Int'l ASIC/SOC Conference*, pages 31–35, September 2002. 5.2.1.2

[15] Cardyak. Cardyak's microarchitecture cheat sheet. `https://t.co/u17qKZbeRw`. 3.5

[16] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Conf. on Supercomputing (SC)*, pages 52:1–52:12, November 2011. 3.1.2, 3.2

[17] Christopher Celio, Palmer Dabbelt, David A Patterson, and Krste Asanović. The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v. *arXiv preprint arXiv:1607.02318*, 2016. (document), 2.3.1, 2.2, 2.4.1, 6.7

[18] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *25th Int'l Symp. on Computer Architecture (ISCA)*, pages 142–153, June 1998. (document), 1.3, 2.2.1.2, 3.1.1, 3.5, 4, 4.1, 4.4, 4.5.2, 5, 5.1.1, 5.3

[19] clamchowder. Lion cove: Intel's p-core roars. `https://old.chipsandcheese.com/2024/09/27/lion-cove-intels-p-core-roars/`, September 2024. 3.5

[20] Advanced Micro Devices. Software Optimization Guide for AMD Family 15h Processors, Pub 47414, Rev 3.08, section 2.9.1, January 2014. [Online; accessed Jun.-2025]. 2.1.2.2

[21] Advanced Micro Devices. Software Optimization Guide for AMD Zen4 Microarchitecture, Pub 57647, Rev 1, January 2023. [Online; accessed Nov.-2023]. 2.1.2.2

[22] Jack Doweck. Inside intel core microarchitecture and smart memory access. In *Intel whitepaper*, pages 1–13, 2006. 2.1.2.1, 2.2.2.3, 6.4.2

[23] Travis Downs. *Memory Disambiguation on Skylake*, 2021. 2.2.2.3

[24] Mark Evers. AMD next generation "zen 3" core. In *33rd HotChips Symp.*, August 2021. 5.2.2

[25] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. `https://www.agner.org/optimize/microarchitecture.pdf`, November 2022. 1.2, 2.3.2, 3.5, 3.5, 6

[26] Johan De Gelas. The bulldozer aftermath delving-even-deeper. https://www.anandtech.com/show/5057/the-bulldozer-aftermath-delving-even-deeper, May 2012. 2.3.2

[27] Weidong Guo, Jiuding Yang, Kaitong Yang, Xiangyang Li, Zhuwei Rao, Yu Xu, and Di Niu. Instruction fusion: Advancing prompt evolution through hybridization. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3883–3893. Association for Computational Linguistics, August 2024. 2.3.1

[28] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *4th Int'l Workshop on Workload Characterization (WWC)*, pages 3–14, December 2001. 3.3

[29] Kamran M Hasan. Memory dependence prediction for energy constrained devices. Master's thesis, University of Toronto (Canada), 2021. 5.4

[30] James Henry Hesson, Jay LeBlanc, and Stephen J. Ciavaglia. Apparatus to dynamically control the out-of-order execution of load/store instructions in a processor capable of dispatchng, issuing and executing multiple instructions in a single processor cycle, September 1997. International Business Machines Corp. 2.2.2.1

[31] Ruke Huang, Alok Garg, and Michael C. Huang. Software-hardware cooperative memory disambiguation. In *12th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 244–253, February 2006. 5.4

[32] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual, Pub 248966-045. [Online; accessed Apr.-2022]. 6

[33] Intel. Intel® 64 and ia-32 architectures software developer's manual. www.intel.com, June 2021. 6

[34] Intel. Intel® 64 and ia-32 architectures optimization reference manual, section e.2.2.2, January 2023. [Online; accessed Nov.-2023]. 2.1.1

[35] Intel. Intel® 64 and ia-32 architectures optimization reference manual. www.intel.com, January 2024. 2.1.2.1, 2.3.2

[36] Intel Corporation, White paper. *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)*, April 2009. (document), 5.1, 5

[37] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 197–206, January 2001. 5

[38] Lei Jin, Hyunjin Lee, and Sangyeun Cho. A flexible data to l2 cache mapping approach for future multicore processors. In *2006 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, pages 92–101, October 2006. 6.7

[39] Zhaoxiang Jin and Soner Önder. Dynamic memory dependence predication. In *45th Int'l Symp. on Computer Architecture (ISCA)*, pages 235–246, June 2018. 4.5.5, 5.4

[40] Pradeep Kanapathipillai, Stephan G. Meier, III Gerard R. Williams, Mridul Agarwal, and Kulin N. Kothari. Load/store dependency predictor optimization for replayed loads, October 2019. Apple Inc. 2.2.2.4

[41] Richard E Kessler, Edward J McLellan, and David A Webb. The alpha 21264 microprocessor architecture. In *16th Int'l Conf. on Computer Design (ICCD)*, pages 90–95, 1998. 2.2.2.2

[42] Richard E Kessler, Edward J McLellan, and David A Webb. The alpha 21264 microprocessor architecture. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No. 98CB36273)*, pages 90–95, October 1998. 6

[43] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A. Jiménez, and Baris Kasikci. Whisper: Profile-guided branch misprediction elimination for data center applications. In *55th Int'l Symp. on Microarchitecture (MICRO)*, pages 19–34, October 2022. 5

[44] Changkyu Kim, Doug Burger, and Stephen W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. *IEEE Micro*, 23(6):99–107, November 2003. 6.7

[45] Ilhyun Kim and M Lipasti. Macro-op scheduling: Relaxing scheduling loop constraints. In *36th Int'l Symp. on Microarchitecture (MICRO)*, pages 277–288, December 2003. 2.3.1

[46] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, January 2018. 1

[47] Vignyan Reddy Kothinti Naresh, Anil Krishna, and Wright Gregoy Michael. Memory violation prediction, March 2018. Qualcomm Incorporated. 2.1.2.3

[48] Evgeni Krimer, Guillermo Savransky, Idan Mondjak, and Jacob Doweck. Counter-based memory disambiguation techniques for selectively predicting load/store conflicts, October 2013. Intel Corp. 2.2.2.3

[49] Johnny K. F. Lee and Alan Jay Smith. Analysis of branch prediction strategies and branch target buffer design. Technical Report UCB/CSD-83-121, EECS Department, University of California, Berkeley, August 1983. 5

[50] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *2011 Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 694–701, November 2011. 3.4, 3.5

[51] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Matthew D. Sinclair Boris Shingarov, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian.

The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020. 3.1.1, 3.2, 5.2.1.1

[52] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. Coatcheck: Verifying memory ordering at the hardware-os interface. In *21st Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 233–247, April 2016. 5.4

[53] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, September 2005. 3.2

[54] Scott McFarling. Combining branch predictors. Technical report TN-36, Digital Western Research Laboratory, June 1993. 5, 6.2.1

[55] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019. 1

[56] Pierre Michaud. An alternative TAGE-like conditional branch predictor. *ACM Transactions on Architecture and Code onOptimization (TACO)*, 15(3):30:1–30:24, October 2018. 5

[57] Karl H. Mose, Sebastian S. Kim, Alberto Ros, Timothy M. Jones, and Robert D. Mullins. Mascot: Predicting memory dependencies and opportunities for speculative memory bypassing. In *31st Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 59–71, March 2025. 1.3, 5.6, 6.7

[58] Andreas Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin-Madison, 1998. 2.2

[59] Andreas Moshovos, Scott E Breach, Terani N Vijaykumar, and Gurindar S Sohi. Dynamic speculation and synchronization of data dependences. In *1997 Int'l Symp. on Computer Architecture (ISCA)*, pages 181–193, June 1997. 2.2, 2.2.1.1, 5.4

[60] Andreas Moshovos and Gurindar S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *30th Int'l Symp. on Microarchitecture (MICRO)*, pages 235–245, December 1997. 1.1, 2.2

[61] muke101. gem5-phast. GitLab, `https://gitlab.com/muke101/gem5-phast/`, June 2024. 1.3, 5.6

[62] Dan Nicolaescu, Alex Veidenbaum, and Alex Nicolau. Reducing data cache energy consumption via cached load/store queue. In *2003 Int'l Symp. on Low Power Electronics and Design (ISLPED)*, pages 252–257, 2003. 6.7

[63] Lena E. Olson, Yasuko ECKERT, and Srilatha Manne. Specialized memory disambiguation mechanisms for different memory read access types, December 2016. Advanced Micro Devices Inc. 2.1.2.2

[64] Soner Önder. Cost effective memory dependence prediction using speculation levels and color sets. In *11th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 232–241, September 2002. 2.2.1.4

[65] Soner Önder and Rajiv Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. In *32nd Int'l Symp. on Microarchitecture (MICRO)*, pages 170–176, December 1999. 2.2.1.2

[66] Yale N. Patt, Stephen W. Melvin, Wen-mei Hwu, and Michael C. Shebanow. Critical issues regarding hps, a high performance microarchitecture. *SIGMICRO Newsl.*, 16(4):109–116, December 1985. 1.1

[67] Qizhi Pei, Lijun Wu, Zhuoshi Pan, Yu Li, Honglin Lin, Chenlin Ming, Xin Gao, Conghui He, and Rui Yan. Mathfusion: Enhancing mathematical problem-solving of llm through instruction fusion. *arXiv preprint arXiv:2503.16212*, 2025. 2.3.1

[68] Arthur Perais and André Seznec. Cost effective physical register sharing. In *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 694–706, March 2016. 6.7

[69] Arthur Perais and André Seznec. Storage-free memory dependency prediction. *IEEE Computer Architecture Letters*, 16(2):149–152, July 2017. 1.1, 1.1, 1.1, 2.2.1.8, 5.3, 5.3.3, 6.7

[70] Arthur Perais and André Seznec. Practical data value speculation for future high-end processors. In *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 428–439, March 2014. 6.7

[71] Arthur Perais and André Seznec. Cost effective speculation with the omnipredictor. In *27th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 25:1–25:13, November 2018. 1.1, 2.2.1.8, 5, 5, 5.2.2

[72] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):318–319, June 2003. 3.3

[73] Jude A. Rivers, Gary S. Tyson, Edward S. Davidson, and Todd M. Austin. On high-bandwidth data cache design for multi-issue processors. In *30th Int'l Symp. on Microarchitecture (MICRO)*, pages 46–56, December 1997. 6.7

[74] Franziska Roesner, Doug Burger, and Stephen W. Keckler. Counting dependence predictors. In *35th Int'l Symp. on Computer Architecture (ISCA)*, pages 215–226, June 2008. 2.2.1.7

[75] Alberto Ros and Stefanos Kaxiras. Non-speculative store coalescing in total store order. In *45th Int'l Symp. on Computer Architecture (ISCA)*, pages 221–234, June 2018. 6.7

[76] Efraim Rotem, Adi Yoaz, Lihu Rappoport, Stephen J. Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, Monica Gupta, and Ahmad Yasin. Intel Alder Lake CPU architectures. *IEEE Micro*, 42(3):13–19, March 2022. 2.1.2.1, 3.5, 5

[77] Amir Roth. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. In *32nd Int'l Symp. on Computer Architecture (ISCA)*, pages 458–468, June 2005. 5.4

[78] André Seznec. The L-TAGE branch predictor. *The Journal of Instruction-Level Parallelism*, 9:1–13, May 2007. 5, 6, 6.7

[79] André Seznec. A 64-Kbytes ITTAGE indirect branch predictor. In *2nd JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction*, June 2011. 5, 6.7

[80] André Seznec. A new case for the tage branch predictor. In *44th Int'l Symp. on Microarchitecture (MICRO)*, pages 117–127, December 2011. 3.1, 5.3

[81] André Seznec. TAGE-SC-L branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, June 2016. 3.2, 3.3, 5, 6.7

[82] André Seznec and Pierre Michaud. De-aliased hybrid branch predictors. Research report RR-3618, INRIA, 1999. 5, 6.7

[83] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism*, 8, 2006. 5, 6, 6.7

[84] André Seznec, Joshua San Miguel, and Jorge Albericio. The inner most loop iteration counter: A new dimension in branch history. In *48th Int'l Symp. on Microarchitecture (MICRO)*, pages 347–357, December 2015. 5, 6.7

[85] Tingting Sha, Milo M. K. Martin, and Amir Roth. NoSQ: Store-load communication without a store queue. In *39th Int'l Symp. on Microarchitecture (MICRO)*, pages 285–296, December 2006. 2.2.1.5, 5, 5, 5.3, 5.3.3, 5.4

[86] Sawan Singh, Arthur Perais, Alexandra Jimborean, and Alberto Ros. Exploring instruction fusion opportunities in general purpose processors. *55th Int'l Symp. on Microarchitecture (MICRO)*, pages 199–212, October 2022. (document), 1.2, 1.3, 2.3.1, 2.4, 3.5, 6, 6.1, 6.6.3.3

[87] RISC-V Software. Spike riscv-v isa simulator. `https://github.com/riscv-software-src/riscv-isa-sim`. 3.2, 3.5

[88] Standard Performance Evaluation Corporation. SPEC CPU2017, 2017. 3.3, 5

[89] Samantika Subramaniam and Gabriel H. Loh. Store vectors for scalable memory dependence prediction and scheduling. In *12th Int'l Symp. on*

*High-Performance Computer Architecture (HPCA)*, pages 65–76, February 2006. 2.2.1.6, 5, 5.1.1, 5.1.2, 5.2.1.2

[90] Areej Syed. Intel 12th gen alder lake golden cove-gracemont cache configuration detailed, July 2021. `https://www.hardwaretimes.com/intel-12th-gen-alder-lake-golden-cove-gracemont-cache-configuration-detailed` 3.5

[91] Harsh Thakker, Thomas P. Speier, Rodney W. Smith, Kevin Jaget, James N. Dieffenderfer, Michael Morrow, Pritha Ghoshal, Yusuf C. Tekmen, Brian Stempel, Sang Hoon Lee, and Manish Garg. Combining load or store instructions. United States patent 20200004550A1, February 2020. 2.3.2, 2.4, 6.1, 6.7

[92] Gary S. Tyson and Todd M. Austin. Improving the accuracy and performance of memory communication through renaming. In *30th Int'l Symp. on Microarchitecture (MICRO)*, pages 218–227, December 1997. 5.4

[93] WikiChip. Macro-operation fusion (mop fusion). `https://en.wikichip.org/wiki/macro-operation_fusion`. (visited on 2025-07-01). 2.3.2

[94] Kenneth M. Wilson, Kunle Olukotun, and Mendel Rosenblum. Increasing cache port efficiency for dynamic superscalar microprocessors. In *23rd Int'l Symp. on Computer Architecture (ISCA)*, pages 147–147, 1996. 6.7

[95] Yang Xu, Yongqiang Yao, Yufan Huang, Mengnan Qi, Maoquan Wang, Bin Gu, and Neel Sundaresan. Rethinking the instruction quality: Lift is what you need. *arXiv preprint arXiv:2312.11508*, 2023. 2.3.1

[96] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. Speculation techniques for improving load related instruction scheduling. In *26th Int'l*

*Symp. on Computer Architecture (ISCA)*, pages 42–53, May 1999. 2.2, 2.2.1.3, 5, 5, 5.1.1, 5.4

[97] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N. Patt. Branchnet: A convolutional neural network to predict hard-to-predict branches. In *53rd Int'l Symp. on Microarchitecture (MICRO)*, pages 118–130, October 2020. 5