Advanced speculation to increase the performance of superscalar processors

Kleovoulos Kalaitzidis

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

« Kleovoulos KALAITZIDIS »

« Advanced Speculation to Increase the Performance of Superscalar Processors »

Thèse présentée et soutenue à Rennes, le 06 Mars 2020
Unité de recherche : Inria Rennes - Bretagne Atlantique
Thèse N° :

Rapporteurs avant soutenance :

Pascal Sainrat	Professeur, Université Paul Sabatier, Toulouse
Smail Niar	Professeur, Université Polytechnique Hauts-de-France, Valenciennes

Composition du Jury :

Président :	Sebastien Pillement	Professeur, Université de Nantes
Examinateurs :	Alain Ketterlin	Maître de conférence, Université Louis Pasteur, Strasbourg
	Angeliki Kritikakou	Maître de conférence, Université de Rennes 1
Dir. de thèse :	André Seznec	Directeur de Recherche, Inria/IRISA Rennes

Dedicated to my beloved parents, my constant inspiration in life..

Computer theorists invent algorithms that solve important problems and analyze their asymptotic behavior (e.g. O(NlogN)). **Computer architects** *set the constant factors for these algorithms.*

Christos Kozyrakis

TABLE OF CONTENTS

RESUMÉ DE THÈSE

À l'ère moderne de la numérisation, où presque tout est traité par les ordinateurs, les exigences de calcul sont croissantes. Aussi les processeurs doivent continuer à augmenter leurs performances, quelle que soit la complexité de la conception. Ainsi, même si les processeurs polyvalents modernes offrent déjà des performances élevées, il y a un besoin constant d'amélioration, en termes de latence d'exécution et de consommation énergétique. En réalité, ceux-ci représentent les deux défis constants de l'architecture de processeur.

À cette fin, les concepteurs de circuits, qui agissent au niveau du transistor, tentent de fournir des optimisations qui visent généralement à raccourcir le chemin critique électrique pour une transmission plus rapide des informations. D'un autre côté, les architectes de processeurs opèrent à un niveau supérieur qui est toujours proche du circuit des portes logiques. La responsabilité d'un architecte est de définir l'organisation du processeur, qui comprend la description de chacun de ses composants. Afin d'atteindre ces objectifs de performance, les architectes peuvent concevoir de nouveaux composants, en supprimer d'autres des modèles traditionnels ou modifier leurs fonctionnalités et leurs connexions, tout en garantissant l'orchestration efficace de toutes les différentes unités de traitement entre elles. Cependant, ils doivent toujours être conscients de la technologie actuelle des circuits, afin de fournir des solutions terre-à-terre, c'est-à-dire des conceptions qui peuvent être mises en œuvre sur du vrai silicium. En d'autres termes, les architectes informatique pourraient être considérés comme des architectes traditionnels qui dessinent leurs conceptions en respectant l'état de l'art des techniques de constructions, afin que les fabricants (c'est-à-dire les concepteurs de circuits) puissent les réaliser dans la pratique.

Depuis l'introduction du tout premier processeur informatique largement utilisé, l'Intel 8080 en 1974 [74], les fabricants de matériel tentent traditionnellement de placer de plus en plus de transistors sur une surface finie de silicium. En particulier, les processeurs généralistes actuels comportent des centaines de millions à des milliards de transistors. De manière évidente, plus de transistors (blocs de construction du processeur) signifie plus d'unités disponibles pour le traitement des instructions du programme et peut naturellement conduire à une vitesse de traitement plus élevée. Le nombre croissant de transistors dans les processeurs avait été empiriquement observé par *la loi de Moore*, qui déclarait que "*le nombre de transistors sur une puce doublerait tous les 2 ans*" [Moo98]. Cela signifie qu'environ tous les 24 mois, un transistor ne prendrait plus que la moitié de sa taille actuelle, de sorte que le double du nombre de transistors pourrait tenir dans une puce de taille égale. Bien qu'elle ne soit pas une loi au sens

mathématique, la loi de Moore s'est révélée être un indicateur assez précis de cette tendance depuis plus de 40 ans. Cependant, depuis 2001 environ, le passage à l'échelle de la technologie a ralenti alors que la miniaturisation des transistors atteint ses limites et la courbe de la loi de Moore est désormais difficile à suivre.

De plus, comme des processeurs de plus en plus complexes sont conçus et fabriqués selon des tolérances de l'ordre du nanomètre, les performances des processeurs ont augmenté en raison de l'escalade de la vitesse d'horloge. L'horloge d'un processeur se déclenche périodiquement et définit le temps minimum nécessaire pour terminer une étape de calcul particulière. Ainsi, en augmentant la fréquence d'horloge, plus de tâches peuvent être accomplies dans un certain intervalle de temps. La fréquence d'horloge est liée à la vitesse à laquelle les transistors d'un circuit peuvent être activés et désactivés de manière fiable. Au fur et à mesure que les transistors rétrécissaient, leur vitesse de commutation augmentait suivant une loi inverse, permettant l'émergence de processeurs à hautes fréquences, allant aujourd'hui de 2 GHz à 5 GHz. Pourtant, la fréquence des CPU est intrinsèquement limitée par les contraintes de puissance crête de la puce, également connue sous le nom de Power Wall [Bos11]. En conséquence, les vitesses d'horloge ont stagné pendant plus d'une décennie car, il semblerait qu'une augmentation spectaculaire de la fréquence peut nécessiter un changement dans les lois de la physique.

Néanmoins, le grand nombre de transistors dans les processeurs a permis l'émergence de nouvelles méthodes de conception, telles que *le pipeline*, *l'exécution superscalaire* et *l'exécution dans le désordre* [AA93], qui ont montré que les performances des processeurs peuvent augmenter même si la fréquence d'horloge atteint un plateau. Ces techniques améliorent les performances des processeurs monocœurs, c'est-à-dire les performances séquentielles, en parallélisant l'exécution des instructions, une caractéristique qui est généralement appelée *parallélisme d'instructions* ou *ILP* (Instruction Level Parallelism). Bien que ces mécanismes aient été l'épine dorsale de l'architecture des processeurs jusqu'à aujourd'hui, il était devenu très difficile de continuer à augmenter les performances séquentielles au même rythme que l'augmentation du nombre de transistors. Par conséquent, afin de soutenir l'augmentation des performances, les concepteurs de puces et les fabricants ont adopté le principe de "*diviser pour mieux régner*", en passant progressivement à la mise en œuvre de puces multiprocesseurs ou multicœurs. En 2001, IBM a introduit POWER4 [War+02], la première puce multiprocesseur disponible dans le commerce qui comprenait 2 cœurs sur une seule puce, suivie par le Pentium D d'Intel et l'Athlon 64 x2 d'AMD vers 2005 [Man+06; Lu +07]. Aujourd'hui, le calcul haute performance est essentiellement dominé par des processeurs multicœurs composés de 2, 4, 8 cœurs ou plus.

Dans un processeur multicœur, l'exécution globale d'un programme peut être accélérée en le divisant en plusieurs programmes plus petits qui sont exécutés simultanément dans différents cœurs. Cette "*mentalité de serveur*" améliore les performances en termes de débit

(c'est-à-dire que plus de travail peut être traité dans une unité de temps donnée), contrairement aux monoprocesseurs qui ciblent la latence d'exécution (c'est-à-dire l'achèvement plus rapide d'une tâche particulière). Pourtant, les performances d'un processeur multicœur sont limitées par la nature du programme, car le calcul parallèle ne peut pas s'appliquer aux composants algorithmiques qui doivent être traités séquentiellement. En d'autres termes, certains programmes sont intrinsèquement séquentiels et ne peuvent pas du tout être parallélisés, tandis que d'autres peuvent posséder de longues parties séquentielles et ne peuvent être que partiellement parallèles. Finalement, comme le suggère la loi d'Amdahl [Amd67], le gain de performance qui peut être obtenu en parallélisant un programme est régi par ses aspects séquentiels. En d'autres termes, si l'on pouvait séparer un programme en ses parties parallèle et séquentielle, alors, en supposant un nombre infini de cœurs qui exécutent simultanément la fraction parallèle, l'accélération totale serait limitée par l'exécution des éléments séquentiels. Par exemple, si une application a besoin de 4 heures de calcul et que seulement 2 d'entre elles peuvent être parallélisées, le temps d'exécution final peut être réduit au maximum d'un facteur 2. Par conséquent, même à *l'ère multicœur*, rendre les cœurs plus rapides dans un contexte monocœur est primordial pour obtenir de hautes performances de calculs [HM08].

Les cœurs de processeur haute performance actuels utilisent des techniques de nature spéculative qui s'appuient sur des observations empiriques sur le comportement des programmes afin de devenir "plus intelligents" à l'exécution du programme. La principale observation qui motive à utiliser ces techniques est que les programmes ordinaires, pour la plupart, ne se comportent pas de manière aléatoire. Ils ont plutôt tendance à présenter une certaine régularité dans le flux d'instructions et dans les résultats de leurs opérations. Ainsi, en isolant ces comportements récurrents de programme, il est possible de concevoir des automates capables de deviner des informations d'exécution liées au programme. Ces aspects peuvent être mieux compris en considérant deux techniques de spéculation largement utilisées qui sont renforcées par le comportement des programmes: *la prédiction de branchement* et *la prédiction de dépendances mémoire*. La prédiction de branchement exploite la régularité trouvée dans les chemins de flux de contrôle que les programmes ont tendance à suivre, tandis que la prédiction de dépendances mémoire repose sur la régularité trouvée dans les interactions entre les instructions de lecture mémoire et de stockage. Ces deux techniques sont utilisées par pratiquement tous les cœurs haute performance modernes pour faciliter le flux d'instructions dans le cœur (c'est-à-dire la récupération ou fetch et la planification ou scheduling) et, par conséquent, augmenter les performances séquentielles.

Un autre niveau de spéculation plus avancé exploite la régularité sous-jacente des données brutes produites par les programmes ou des adresses mémoire d'où proviennent des données particulières. Deux techniques principales de ce type sont *la prédiction de valeur* (VP pour value prediction) et *la prédiction d'adresse des lectures mémoire* (LAP pour load-address

prediction). VP et LAP sont deux techniques en développement et très prometteuses de spéculation de données, dans le sens où elles offrent encore beaucoup de potentiel pour l'exploration et l'optimisation.

VP rompt les véritables dépendances de données entre les instructions en permettant aux "consommateurs" de lancer leur exécution de manière spéculative avant que leurs "producteurs" respectifs ne soient exécutés. VP est permis par le fait que différentes instances dynamiques de la même instruction statique produisent des résultats qui suivent souvent un modèle prédictible. Par conséquent, une structure matérielle peut être utilisée pour suivre ces instructions et prédire leur résultat. Lorsqu'une instruction est prédite, celles qui en dépendent peuvent s'exécuter simultanément en utilisant la valeur prédite sans avoir à se bloquer. Ainsi, lorsqu'une prédiction est correcte, le temps d'exécution total diminue. Dans le cas contraire, certaines actions de récupération doivent être mise en place pour garantir la ré-exécution des instructions dépendantes avec la bonne entrée. L'essentiel est que les performances séquentielles augmentent à mesure que le processeur peut extraire plus d'*ILP*.

LAP est assez similaire à VP; plutôt que de prédire les résultats des lectures mémoire, les mécanismes LAP prédisent les adresses mémoire à partir desquelles les instructions de lecture mémoire récupèrent leurs données. Ces mécanismes sont basés sur l'observation que les adresses des lectures mémoire peuvent également tomber dans des modèles prédictibles. La prédiction de l'adresse mémoire d'un lecture mémoire permet d'anticiper l'accès aux données chargées et de générer une valeur spéculative pour cette lecture mémoire. Cette valeur n'est pas nécessairement prédictible par un mécanisme VP. Comme dans VP, les valeurs spéculatives des lectures mémoire sont propagées aux instructions dépendantes et leur permettent de s'exécuter en avance. En tant que tel, LAP initie une forme similaire d'exécution spéculative basée sur les données et nécessite aussi une validation.

Dans l'ensemble, ces deux techniques de spéculation visent à augmenter l'*ILP* et à réduire la latence d'exécution des instructions dans le cœur. En partageant les mêmes objectifs, nous apportons dans cette thèse les principales contributions suivantes.

Premièrement, nous effectuons une exploration inter-jeux d'instructions (ou ISA pour Instruction Set Architecture) de la prédiction de valeur. Depuis la toute première introduction de la prédiction de valeur, il y a eu une pléthore de différents prédicteurs qui prétendent avoir des améliorations particulières indépendamment de l'ISA. Cependant, comme les techniques de VP dépendent essentiellement de la nature et du comportement du code des programmes, il est essentiel de détecter l'existence d'interactions entre les particularité d'une ISA et la prédiction de valeur. Dans notre étude, nous comparons soigneusement l'accélération apportée avec VP dans les binaires d'application compilés symétriquement pour x86_64 et Aarch64 ISA. Notre exploration montre que, en effet, certaines particularités de l'ISA, telles que la quantité de registres disponibles dans l'ISA, peuvent avoir un impact significatif sur le gain de performance

qui peut être obtenu via VP. De plus, étant donné que les prédicteurs de valeur supposent conventionnellement des entrées de 64 bits pour suivre l'exécution, les instructions vectorielles qui produisent des valeurs d'au moins 128 bits ne sont pas couvertes, c'est-à-dire ni suivies ni prédites. Plus précisément, la prédiction d'instructions vectorielles peut entraîner un prédicteur assez grand, car elle nécessite d'agrandir la longueur des entrées en fonction de la longueur maximale de la valeur qui doit être prédite (par exemple, 128 bits par entrée pour couvrir les instructions de 128 bits). À cet égard, dans notre exploration au niveau de l'ISA, nous considérons également l'influence des optimisations vectorielles dans l'accélération réalisée avec VP, montrant leur impact sévère.

Deuxièmement, nous détectons un motif de valeurs inexploré, à savoir l'égalité d'intervalle, qui caractérise cependant une partie substantielle des instructions (plus de 18% en moyenne). Un tel motif de valeur n'est pas efficacement capturé par les prédicteurs précédemment proposés qui conservent généralement les valeurs suivies (c'est-à-dire qui seront potentiellement utilisées pour la prédiction) et leur confiance en la prédiction dans la même entrée. Afin de supprimer cette limitation, nous introduisons le prédicteur VSEP, qui est basé sur un schéma de décision binaire et dissocie fortement les valeurs de leur confiance en la prédiction. VSEP atteint une accélération qui se situe dans la même plage que le VTAGE [PS14b] proposé précédemment, mais en exploitant des motifs de valeur non capturés auparavant. En d'autres termes, VSEP complète efficacement la façon établie dont VP est effectuée en augmentant considérablement la fraction des instructions prédites qui exposent [de] l'égalité d'intervalle (15% supplémentaires en moyenne). Essentiellement, lorsqu'il est combiné avec VTAGE, VSEP améliore l'accélération obtenue de 19% en moyenne. De plus, avec VSEP, nous pouvons obtenir des performances comparables à celles d'un VTAGE à l'état de l'art, en utilisant environ 58% d'entrées en moins (3K entrées de valeur dans VSEP contre 7K entrées de valeur dans VTAGE) avec le même budget de stockage. De cette façon, VSEP peut réduire considérablement le coût de la prise en compte de la prédiction de valeurs pour des valeurs supérieures à 64 bits.

Troisièmement, et enfin, nous proposons un mécanisme basé sur LAP, à savoir LAPELE, qui fait des lectures mémoire anticipés avant la partie out-of-order du cœur. Pour ce faire, nous changeons l'objectif de notre prédicteur hybride VSEP/VTAGE afin d'effectuer LAP. Avec un tel prédicteur, nous réussissons à prédire les adresses mémoire de 48% des lectures mémoire en moyenne. Pour éviter une éventuelle explosion de violations d'ordre de mémoire, nous proposons l'utilisation d'un prédicteur auxiliaire à petite échelle pour "filtrer" les lectures mémoire qui présentent des conflits de mémoire. Finalement, avec ce schéma, nous pouvons exécuter en avance 32% des instructions de lecture mémoire validées en moyenne en les sortant complètement de le partie out-of-order du cœur. Du point de vue des performances, LAPELE atteint une accélération moyenne dans la même plage qu'un prédicteur de valeur de taille égale qui est réglé pour spéculer uniquement sur les valeurs des lectures mémoire. En réalité, les

mécanismes basés sur le LAP ne peuvent pas systématiquement atteindre ou dépasser les performances de la prédiction de valeur généraliste. Cependant, avec LAPELE, nous montrons que LAP peut être un catalyseur nécessaire des futures architectures à grande fenêtre efficaces en complexité pour tolérer la latence de la mémoire.

INTRODUCTION

In his 1937 paper, Alan Turing [Tur37], the father of computer science, stated that *"it is possible to invent a single machine which can be used to compute any computable sequence"*. The *Turing machine* or the *a-machine* (automatic machine) is a mathematical model that defines an abstract machine capable of implementing any given computer algorithm. In automata theory, an "abstract machine" is the theoretical model or the specification of a computing system. In the field of computer science, this abstraction simply represents the model of a processing unit. Indeed, the *Turing principle* is considered to be the general example of a Central Processing Unit (CPU) that carries out all computations within a computer. Traditionally, the term "CPU" refers to the processor, distinguishing it from external components, such as the main memory.

In essence, computers are designed to process, store, and retrieve data. Processors are fairly called the "brains" of computers, since they enable all basic arithmetic, logical, control and input/output(I/O) operations specified by a computer program. As such, they basically set the limits of computers processing capacity and they are reasonably considered the cornerstone of computer science. Image processing, bioinformatics, machine learning and artificial intelligence are only few aspects of computer science that could not simply exist without machines with sufficiently high computational capabilities. Moreover, both the miniaturization and standardization of CPUs has increased the presence of digital devices in modern life, far beyond the limited application of dedicated computing machines. Modern microprocessors appear in electronic devices ranging from automobiles to cellphones and from domestic appliances to critical medical devices. As a result, processor architecture is considered as one of the most important types of hardware design.

The word "*architecture*" typically refers to the design and construction of buildings. Similarly, in the computing world, computer architecture respectively describes the design of computer systems. More specifically, the architecture of a processor chip, known as microarchitecture, is the detailed description of its basic components, their functionalities and their interconnections, as well as the operations that they can effectuate. All these diverse features are fully expressed by the instructions that the processor supports, which all together represent the *Instruction Set Architecture* (ISA) of the processor. Different processor "families", such as Intel, IBM, PowerPC and ARM, can employ different categories of ISAs. For that reason, a program that is constructed by the compiler in a particular type of ISA, can not be executed on a machine that does not implement the same ISA. On the other hand, a single ISA may be implemented by different models of processors that remain compatible at the ISA level. For instance, popular

families, such as x86, have processors supplied by multiple manufacturers. In other words, the ISA provides a conceptual level of abstraction between compiler writers, who only need to know the available machine instructions and their encoding, and processor designers, who build machines that execute those instructions. Overall, the ISA together with the microarchitecture form what is called *computer architecture*.

From a microscopic point of view, *transistors* are the atomic parts of typical microprocessor chips, in the same sense that bricks are the major components of buildings. Transistors are arranged in certain ways in order to form electric circuits that are capable of carrying out all fundamental binary functions, such as OR and AND. Thereafter, the sophisticated combination of these functions can provide devices that enable higher-level operations, such as additions and multiplications. By solely considering the inarguably high complexity of designing such circuits, one can easily infer how perplexing is the actual procedure of "*building*" a processor.

In the modern age of digitalization, where almost everything is processed by computers, the computational demands are in an ascending orbit. In order to sustain the pace in computer-science research and quality-of-life improvement, processors have to keep increasing their performance, regardless of the design complexity. Thus, even if modern general-purpose processors already provide high performance rates, there is a continuous need of improvement, in terms of execution latency and power consumption. In reality, these represent the two constant challenges of processor architecture.

To that end, circuit designers, that act at the transistor level, attempt to provide optimizations that generally aim to shorten the electric critical path for the faster transmission of the information. On the other hand, computer architects operate on a higher level which is still close to the logic gates circuit. An architect's responsibility is to give a blueprint of the processor's organization, which includes the description of all its distinct components. In order to reach the respective performance goals, architects may design new components, remove some others from mainstream models or modify their functionality and their connections, by always guaranteeing the effective orchestration of all the different processing parts together. However, they have to always be aware of the current circuit technology, in order to provide down-to-earth solutions, i.e. designs that can be implemented on a real silicon. In other words, computer architects could be considered regular architects who draw their designs with respect to the state-of-the-art constructing principles, so that manufacturers (i.e., circuit designers) can carry them out in practice.

Since the introduction of the very first widely-used computer processor, the Intel 8080 in 1974 [74], hardware manufacturers traditionally attempt to cram more and more transistors onto a finite silicon surface. In particular, current general-purpose processors feature hundreds of millions to billions of transistors. Evidently, more transistors (processor building blocks) means more units available for processing program instructions and can naturally lead to higher pro-

cessing speed. The increasing number of transistors in processor chips had been empirically observed by *Moore's law*, which stated that "*the transistor count on a chip will double every 2 years*" [Moo98]. This meant that about every 24 months a transistor would be half of its current size so that double number of transistors could fit in an equally-sized chip. While not a law in the mathematical sense, Moore's Law has proved a fairly accurate predictor of this trend for more than 40 years. However, since around 2001, technology scaling has been slowing down as transistors shrinking is reaching its limits and Moore's Law curve is henceforth difficult to follow.

Moreover, as increasingly complex CPUs are designed and manufactured to tolerances on the order of nanometers, processors performance has been growing due to the escalation of the processors clock speed. A processor's clock ticks periodically and defines the minimum time needed for completing a particular computation step. Thus, by increasing the clock frequency more tasks can be completed within a certain interval of time. The clock frequency practically resembles the speed at which the transistors in a circuit can be reliably switched on and off. As transistors have been progressively shrinking, their switching speed has been inversely increasing, allowing the emergence of processors with high frequencies, nowadays ranging from 2GHz to 5GHz. Yet, CPUs frequency is inherently limited by the peak power constraints of the chip, also known as the Power Wall [Bos11]. As a result, clock speeds have stagnated for more than a decade because, as it appears, a dramatic increase in frequency may require a change in the laws of physics.

Nonetheless, the large number of transistors in processor chips enabled the emergence of novel design methods, such as *pipelining*, *superscalar* and *out-of-order* execution [AA93], which showed that processors performance can increase even if the clock frequency plateaus. Such techniques improve the performance of a single or uni-processor, i.e. *sequential performance*, by parallelizing instruction execution, a feature that is typically referred as *Instruction Level Parallelism* or *ILP*. Although these mechanisms have been the backbone of processor architecture till today, it had become very hard to keep increasing sequential performance in the same pace that the number of transistors was growing. Therefore, in order to sustain the augmentation of performance, chipmakers and manufacturers adopted the principle of "*divide and conquer*", by progressively shifting to the implementation of multiprocessor or multicore chips. In 2001 IBM introduced POWER4 [War+02], the first commercially available multiprocessor chip that featured 2 cores on a single die, followed by Intel's Pentium D and AMD's Athlon 64 x2 in around 2005 [Man+06; Lu +07]. Today, high-performance computing is essentially enabled by multicore processors that are composed by 2, 4, 8 or even more cores.

In a multicore processor the overall execution of a program can be sped up by dividing it into several smaller programs that are executed concurrently in different cores. This "*server mentality*" improves performance in terms of throughput (i.e., more work can be processed within

a given unit of time), as opposed to uni-processors that target execution latency (i.e., faster completion of a particular task). Still, the performance of a multicore processor is limited by the program nature, since parallel computing can not apply to algorithmic components that need to be processed sequentially. In other words, some programs are intrinsically sequential and they can not be parallel at all, while some others may possess long sequential parts and they can only be partly parallel. Eventually, as Amdahl's Law [Amd67] suggests, the performance gain that can be obtained by parallelizing a program is governed by its sequential aspects. In other words, if one could separate a program to its parallel and sequential parts, then, by assuming an infinite number of cores that execute concurrently the parallel fraction, the total speedup would be bounded by the execution of the sequential elements. For instance, if an application needs 4 hours of computing and only 2 of them can be parallelized, the final execution time can be reduced at most by a factor of 2. Consequently, it appears that even in the *multicore era*, making single cores faster is paramount to achieve high-performance computing [HM08].

Current leading-edge processor cores employ speculative-nature techniques that rely on empirical observations about programs behavior in order to become effectively "smarter" about program execution. The main observation that works as the enabling incentive of these techniques is that ordinary programs, for the most part, do not behave randomly. Rather, they tend to exhibit a certain regularity in the flow and in the products of their operations. As such, by isolating these program "idiosyncrasies", there is the ability to devise automata capable of guessing program-related execution information. These aspects can be understood better by considering two widely-used speculation techniques that are empowered by programs behavior: *branch prediction* and *memory dependence prediction*. Branch prediction exploits the regularity found in the control-flow paths that programs tend to follow, while memory dependence prediction relies on the regularity found in the interactions between load and store instructions. Both these techniques are employed by virtually all modern high-performance cores to facilitate the flow of instructions in the core (i.e. fetching and scheduling, respectively) and therefore, to increase sequential performance.

Another more advanced level of speculation leverages the underlying regularity in either the raw data that programs produce or the memory addresses from where particular data originate. Two major techniques of this kind are *value prediction* (VP) and *load-address prediction* (LAP). VP and LAP are two developing and highly-promising *data-speculative* techniques, in the sense that they still feature plenty of room for exploration and optimization. To this respect, this thesis proposes certain speculative-nature mechanisms that are based on value and load-address prediction in order to effectively improve the sequential performance of contemporary general-purpose processors. These mechanisms are devised on-top of the architecture of a single-core processor and are totally independent from the higher-level parallelism that multicore designs exploit. Therefore, they can be seamlessly adopted by the different cores of

a multicore processor and potentially scale also its performance. Additionally, when this thesis introduces modifications in the internal core-design, the current technology constraints are faithfully respected. That is, the proposed techniques rely on hardware structures that can be practically implemented on a real processor.

Contributions

The mechanisms that are introduced by this thesis are related with two main classes of advanced speculation: *value prediction* and *load-address prediction*.

Value Prediction is a technique that breaks true data dependencies between instructions by allowing "consumers" to initiate their execution speculatively before their respective "producers" have executed. VP is enabled by the fact that different dynamic instances of the same static instruction produce results that often follow a predictable pattern. Consequently, a hardware structure can be used to track these instructions and predict their result. When an instruction is predicted, its dependents can execute concurrently by using the predicted value without needing to stall. As such, when a prediction is correct, the total execution time decreases. On the opposite case, certain recovery actions must be taken to guarantee the re-execution of dependents with the correct input. The bottom line is: sequential performance increases as the processor can extract more *ILP*.

Load-Address Prediction is fairly similar to VP; rather than predicting the results of load instructions, LAP-mechanisms predict the memory addresses from which loads fetch their data. Such mechanisms are based on the observation that load addresses may also fall into predictable patterns. Predicting a load's memory address allows to anticipate the access to the loaded data and to generate a speculative value for the particular load instruction. That value is not necessarily predictable by a VP-mechanism. Like in VP, the speculative values of load instructions are propagated to dependents and allow them to execute ahead of time. As such, LAP initiates a similar form of data-speculative execution that respectively requires validation.

Overall, both of these speculation techniques aim to boost *ILP* and to reduce the execution latency of instructions in the core. By sharing the same objectives, this dissertation makes the following main contributions:

1. *Analysis of the ISA impact on the performance of value prediction*:

   Our first contribution makes the case that certain ISA particularities have a significant impact on the performance gain that can be obtained with value prediction. In particular, we carefully studied and compared the performance of value prediction in the context of two widely-used 64-bit ISAs in high-end cores, namely the *x86_64* and the *Aarch64* ISA. What we basically discovered from our inter-ISA exploration of value prediction is that the obtained speedup is consistently higher in x86_64, with the speedup-gap being up to

26% in the interest of x86_64. The main reason is that, compared to Aarch64, x86_64 contains less architectural registers that lead to program-code with higher concentration of load instructions that are eventually predicted. In essence, our general observation is that by virtue of this registers-related particularity, in x86_64, the code-regions that induce substantial execution delays due to data dependencies between instructions, are more frequent but at the same time more predictable. In addition, we isolated the influence of vector optimizations in the VP performance. Such optimizations have introduced instructions producing results longer than the 64 bits that the current technology assumes. Conventionally, these instructions are not considered for value prediction, since tracking their fairly wide values scales prohibitively the size of value predictors. In practice, current value predictors are typically limited to speculate on up to 64-bit values. However, our study reveals that this convention limits considerably the corresponding performance benefit. This issue is partly addressed by our second contribution.

2. *VSEP [KS19], a binary-nature VP model*:

   Our second contribution relates to the design of a value predictor that exploits an unexplored value pattern that is, however, substantially followed by instruction results. In essence, the driving force of value prediction is the existence of certain predictable patterns exposed by instruction values. In that sense, the constant challenge of contemporary value predictors is to sufficiently capture these patterns and exploit the predictable execution paths. To do so, existing techniques tightly associate recurring values with instructions and distinctive contexts by building confidence upon them after a plethora of repetitions. What we found is that execution monotony of an instruction can exist in the form of *intervals* and not only uniformly. That is, dynamic instances of the same static instruction may not only constantly produce a particular value, but also in intervals of successive executions. Nonetheless, previous VP-schemes have not been devised with total awareness of this interval-style pattern, but rather of the uniform one. As such, these intervals limit the potential fraction of predicted instructions (coverage), since in a conventional value predictor, confidence is reset at the beginning of each new interval. We address this challenge by introducing the notion of *Equality Prediction* (EP), which represents the binary facet of value prediction. By following a twofold decision-scheme (similar to branch prediction), EP makes use of control-flow history to determine equality between the last committed result read at fetch time, and the result of the fetched occurrence. When equality is predicted with high confidence, the read value is used. VSEP, our value predictor that makes use of EP, obtains the same level of performance as previously proposed state-of-the-art predictors. However, by virtue of exploiting value patterns not previously captured, our design complements the established way that value prediction is performed, and when combined with contemporary prediction models, improves the benefit of value

prediction by 19% on average. Moreover, VSEP mitigates significantly the cost of predicting values that are wider than 64 bits, as it is by construction highly independent from the values width.

3. *LAPELE, a LAP-based design for data-speculative execution*:

   Our third and last contribution focuses particularly on reducing the total execution latency of load instructions. We propose LAPELE, a comprehensive design that leverages load-address prediction for data speculative execution. More specifically, in LAPELE we use a load-address predictor, namely ATAGE-EQ, which is directly derived from the VSEP value predictor of our second contribution and the previously-proposed VTAGE [PS14b]. In this way, we practically explore load-address predictability by capitalizing on recent advances in value prediction, having the intuition that load-addresses follow similar patterns. Indeed, ATAGE-EQ allows us to predict the addresses of 48% of the committed loads on average. The uniqueness of LAPELE is that loads that predict their address, use it to perform their memory access from the early pipeline frontend (early execution) and then bypass the out-of-order execution core. That is, no second repairing execution takes place. To do so, LAPELE employs some additional logic in charge of computing the actual address of address-predicted loads before Commit, similarly to the EOLE architecture [PS14a]. The data-speculative execution is then validated by verifying the predicted address before removing the instruction from the pipeline. However, such a flow is highly sensitive to memory hazards. To avoid a prohibitive increase of memory-order violations, we propose to use MDF, an auxiliary predictor of fairly low size and complexity, in order to "filter" the loads that use their predicted address for early execution. Eventually, LAPELE achieves to execute beyond the out-of-order engine, i.e. in the early frontend, 32% of the committed load instructions, achieving an average speedup in the same range with load-value prediction. This is in contrast with previous schemes where address-predicted loads were normally dispatched for execution in the out-of-order core.

Organization

This dissertation is organized as follows. Chapter 1 serves as a comprehensive analysis of general-purpose processors architecture. We start from describing simple pipelining and we conclude to modern superscalar processors of out-of-order execution. In this way, readers may infer from the beginning the benefits that advanced speculation can have on the sequential performance of modern processors. In Chapter 2, we present the two speculation techniques we consider, namely *value prediction* and *load-address prediction*. We describe their logic and we provide the state-of-the-art models, as well as related work in general. Chapter 3 focuses on our first contribution. We first provide a basic analysis of the two employed ISAs and the tailored

framework we use to enable inter-ISA comparison, then we analyze our findings. In Chapter 4, we introduce our VSEP predictor (i.e. second contribution). First, we present the new value pattern we have detected. Then, we analyze how typical predictor and how VSEP interacts with it, before we provide an experimental analysis and comparison of VSEP with previous recent work. Chapter 5, concerns our third and last contribution. We analyze how we choose to use ATAGE-EQ as our load-address predictor. Then, we analyze the arising issues related with LAPELE and we present its microarchitectural model. We eventually evaluate VSEP considering prior relevant work. Finally, we conclude this dissertation summarizing our findings and offering suggestions on how this work can be extended.

CHAPTER 1

THE ARCHITECTURE OF A MODERN GENERAL-PURPOSE PROCESSOR

> "The term ***architecture*** is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior as distinct from the organization of the dataflow and controls, the logic design, and the physical implementation."
> Gene Amdahl, April 1964 [ABB64]

1.1 Fundamentals of Processor Architecture

1.1.1 General Definition

Processor or computer architecture is the science of designing, selecting, and interconnecting hardware components in order to create computing systems that meet functional, performance, power consumption, cost, and other specific goals. Even more precisely, according to the recognized computer architect Frederick P. Brooks Jr., *"Computer architecture, like other architecture, is the art of determining the needs of the user of a structure and then designing to meet those needs as effectively as possible within economic and technological constraints"* [Buc62]. General-purpose processors form the *core* of such computing systems and as such, terminology-wise, computer architecture typically refers to processors internal design and to their intrinsics.

1.1.2 Instruction Set Architecture - ISA

Essentially, processors are programmable, and thus, they provide a specific interface for the communication between the software and the hardware, known as the *Instruction Set Architecture* or the *ISA*. That is, the ISA includes all those instructions that a processor is designed to "comprehend" and accordingly to execute. In that sense, a processor is said to implement a specific ISA. For instance, Intel and AMD processors implement the x86 ISA, while Qualcomm processors implement the ARM ISA. Although two processors may use the same ISA, the way

they implement it may be different. The term *microarchitecture*[1] is used to describe the specific way that an ISA is implemented by a processor and should not be confused with the ISA. Along with the set of instructions, the ISA also defines a specific collection of registers, i.e. memory locations where the operands of instructions can reside. In particular, registers are the closest and the fastest-to-access memory unit in a processor. These two, *instructions* and *registers*, are considered the primitives of a given ISA and both of them are visible to the software level.

Based on the nature of their instructions, existing ISAs are distinguished into two categories: the *Reduced Instruction Set Computing* (RISC) and the *Complex Instruction Set Computing* ISAs [Col+85]. In the former category, instructions are mostly simple operations (such as simple arithmetics or binary logic), while in the latter category, as its name suggests, instructions can perform more complicated tasks. As it follows, RISC ISAs usually contain more instructions in order to be able to eventually carry out equivalent tasks to CISC ISAs. But on the other hand, CISC ISAs are considered more perplexing in their implementation due to the higher complexity of their instructions, i.e. the microarchitecture of a processor implementing a CISC ISA is relatively more complicated. Furthermore, ISAs are also defined by the number or bits that their instructions use to probe memory. As a note, current technology allows processors to implement up to 64-bit ISAs, meaning that they can actually address 2^{64} memory locations.

1.1.3 Instruction Formats

As already mentioned, instructions describe any single operation that can be performed between values by the processor. A program is a coherent set of instructions that are organized in a sequential way in order to carry out a particular task. The instructions of a program are stored in the processor's memory in a binary representation. Each particular ISA defines a specific way according to which a single instruction is encoded in a binary form. An instruction's encoding basically contains an *opcode* (i.e. the specific operation to be carried out), the *source register(s)* (i.e. the operands) and the *destination register(s)* (i.e. the output). The processor is in charge of retrieving, decoding and executing the instructions in the order that the program has arranged them.

The amount of bits that are eventually used for the representation of instructions can be either equal or different between instructions. As such, existing encoding schemes are characterized as either fixed- or variable-length, respectively. When instructions are encoded with a fixed length are considered to be easier to decode than having variable length. Indeed, in a fixed-length representation, an instruction's special attributes, such as its source registers, are always found in the same offset in the binary word and therefore, during decoding, the processor can deterministically locate them. Conversely, in a variable-length representation,

1. In the rest of this document, the words microarchitecture and architecture will be used interchangeably referring to the design of a processor.

instructions aspects are accordingly in variable locations, complicating more their decoding. Furthermore, when instructions have fixed length it is straightforward to identify where the next instruction begins. Unfortunately, however, memory capacity is wasted because not all the instructions require the same amount of bits to encode their operation. For instance, when an instruction does not feature a destination register (e.g. a branch instruction), some bits are still reserved for its representation but are apparently useless. Therefore, encoding schemes set some ISA-level tradeoffs. Regarding the ISAs we mentioned above, x86 contains variable-length instructions, while ARM uses fixed-length instructions.

1.1.4 Architectural State

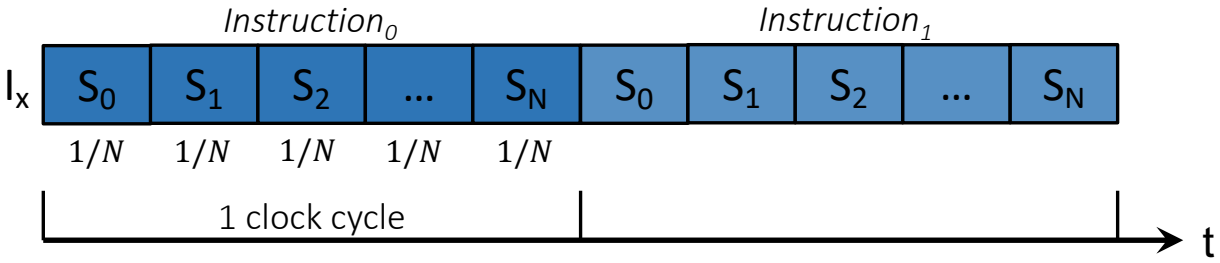
Previously, we mentioned that instructions and registers are two aspects of the ISA that are visible to the software. Together with them, the software is also aware of the memory state, i.e. its content. More specifically, the registers and the memory form together what is called the *Architectural State* of the processor [2], which is by definition the processor's state that is visible to the software. Practically, a program's instructions operate on the current architectural state of the processor (i.e. the contents of its registers and its memory) and continuously change it according to the demands of the relative task. From the software point of view, undergone modifications induced by an instruction are atomic. However, in the hardware level, an instruction may pass through several phases till completion, where each of them affects the content of different hardware components. As such, the software-visible architectural state is the result of various snapshots during instructions execution that are only accessed by the hardware. In the paragraphs that follow we will detail the way that instructions flow and execute in a processor, as long as the different phases from which they pass.

1.2 Simple Pipelining

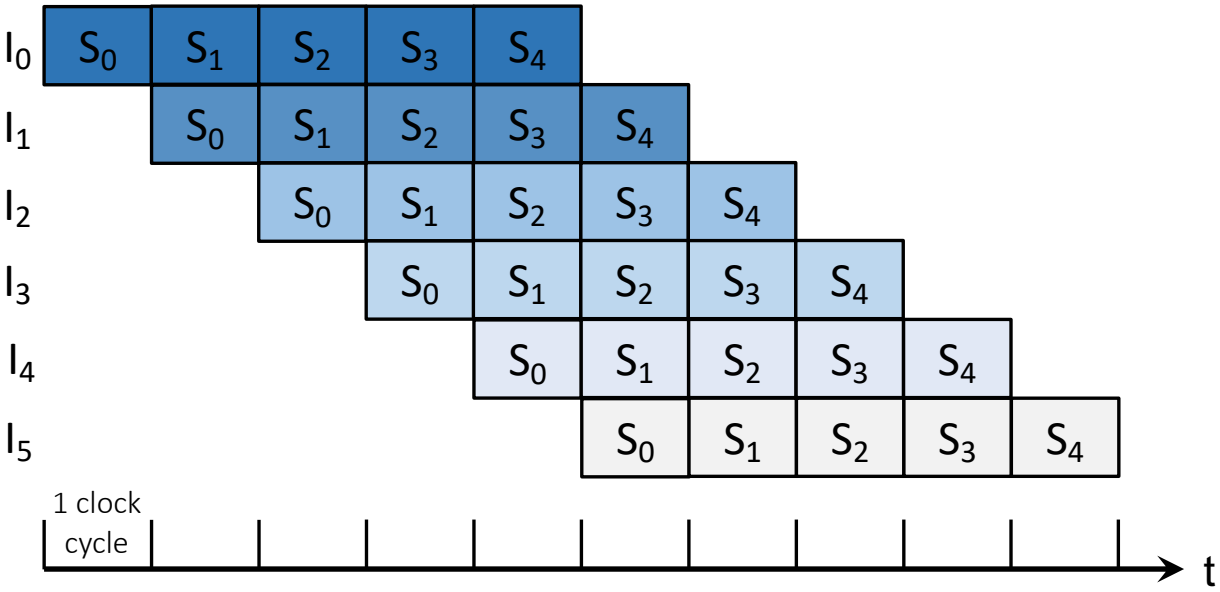
1.2.1 Key Idea

A basic aspect of processors is their *clocked* behavior, in the sense that their operation is driven by a clock that ticks a specific number of times per second. Each time the clock ticks, the execution of some tasks in the processor is triggered. Based on that scale, the primitive type of a microarchitecture is defined as the *1-cycle microarchitecture*, where the entire execution of any instruction fits within the interval of two consecutive clock-ticks, i.e. a clock cycle. To ensure that this is possible, the clock needs to tick moderate enough to permit the slowest (the one requiring the most processing) instruction to execute in a single clock cycle. That is, the

2. Including the *Program Counter* (PC), which is the address in memory from where the next instruction to execute can be fetched.



(a) Single-cycle architecture with instruction execution separated in distinct stages *S*.



(b) Simple five-stages pipeline. The delay of a clock cycle is equal to 1/5 of the single-clock architecture.

Figure 1.1 – Segmentation of instruction execution in abstract and independent stages for pipelined execution.

slowest instruction is the one determining the exact cycle time. During a cycle, the execution of an instruction contains several very different "steps" that require very different hardware modules. Since (roughly) any single instruction needs to flow through all these hardware modules during execution, most of these steps are common to all instructions. Therefore, assuming that instruction execution involves *N* necessary steps, an instruction will be considered a *N-stage* procedure, as shown in Figure 1.1a. With this "slicing" of instructions to stages, it is revealed that each of the corresponding *N* hardware structures is active only during $\frac{1}{N}$ of the total clock cycle, making 1-cycle micro-architectures highly inefficient.

Pipelining was introduced to alleviate this issue by exploiting this "step" style of instruction execution. In particular, with pipelining the several execution-steps that can be identified are embodied in discrete execution-stages. These stages are by definition independent with

respect to the hardware structures they use to process their tasks and therefore can work simultaneously. Consequently, since instructions need to pass through them sequentially, it is plausible to have instruction I_0 carrying out tasks related with stage S_1 while instruction I_1 is concurrently processed in S_0. In this way, instruction execution can effectively overlap, utilizing resources much better. In order to allow this kind of parallelization, the cycle time has to be reduced to the level of the slowest pipeline stage, so that the relevant tasks will be effectively triggered at the beginning of each stage. This results to a *multi-cycle* micro-architecture, where each instruction needs more than one cycle till completion.

Although the number of cycles per instruction increases, their absolute latency remains the same. Nonetheless, performance increases since execution parallelism leads to improved instruction throughput, i.e. more instructions completed within a unit of time. In other words, the global program latency is reduced. Figure 1.1 depicts together the simple 1-cycle architecture and an abstract 5-stage pipeline. As shown, in the pipeline execution any single instruction needs 5 cycles to completion, with the cycle delay being equal to one fifth of the cycle delay of the 1-cycle architecture. However, as long as the pipeline is full, the throughput of *one instruction per cycle* is maintained. As a result, six instructions can finish their execution at the same time that only two of them would have completed in a 1-cycle architecture. Ideally, a pipeline with five stages can be five times faster than a non-pipelined processor (or rather, a pipeline with one stage). Consider that in both of them instruction I_1 finishes one clock cycle after instruction I_0, but five times faster for the pipelined processor since its clock cycle is one fifth of the clock cycle for the non-pipelined processor.

As it appears, the more pipeline stages there are, the faster the pipeline is, because each stage is shorter and instructions are executed at the speed at which each stage is completed. Although it may initially seem tempting to make pipelines as longer as possible, pipeline depth soon reaches its upper boundary. First, as the pipeline becomes deeper and the processor's frequency is increased, power consumption scales proportionally and the design soon becomes impractical. Second, the consecutive processing of instructions coupled with structural hazards natural to the pipeline organization, prevent further performance growth beyond a certain amount of pipeline-cycles (around 15 to 20). Readers should note that those hazards will be detailed in next Sections. Finally, in circumstances that the pipeline becomes empty, re-filling it takes a great amount of time, leading to significant performance loss when the pipeline-depth exceeds a specific threshold.

1.2.2 Stage Breakdown

In the following paragraphs, we detail how the execution of an instruction can be broken down into 5 main stages. We detail each of these stages in the following sections.

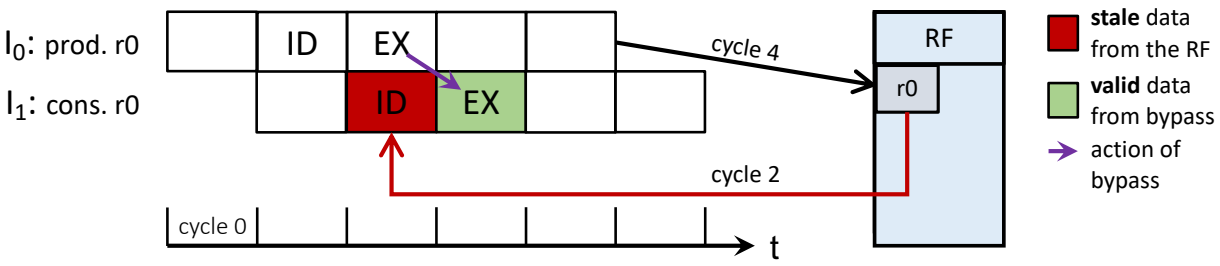


Figure 1.2 – State of the source operands of a dependent instruction based on their origin in a pipeline featuring a bypass network. Irrelevant stage labels are not shown for clarity.

1.2.2.1 Instruction Fetch - IF

The first stage refers to the transfer of the instruction from the global memory where it resides, to the processor. To do this, the memory hierarchy needs to be indexed using the address contained in the Program Counter (PC). The PC is afterwards updated in order to point the correct instruction for the next cycle.

1.2.2.2 Instruction Decode - ID

An instruction fetched for execution is encoded in a binary format. In this stage, all the information that is condensed in this binary form needs to be deciphered or *decoded* into control words that are meaningful to the processor and guide further steps in the pipeline. In particular, these control words mainly specify the instruction's type and accordingly the functional-unit that will be needed for its execution and the register operands, sources and destination. In the abstract pipeline that we consider, register operands are also read during this stage. Moreover, instructions with complex semantics are divided into primitive µ-ops compatible with the relevant ISA. Note that not only CISC architectures feature such complex instructions.

1.2.2.3 Execution - EX

The third stage is that of the actual execution, where the actual result of the instruction is computed or if it is a memory instruction, its memory address is calculated. Depending on the type of instruction, the instruction is directed to the appropriate functional unit, i.e., the processor element that implements the operation required by the instruction. However, an instruction dependent on a value produced by a previous instruction should wait till the produced value reaches the relevant register so that it can be read. Values are written to the register file during the *Write-back* stage, i.e. two cycles after execution. To avoid this delay and allow back-to-back execution of two dependent instructions, produced values are promoted to the *bypass network*. In the processor we consider, the bypass network spans from Execute to Write-back

and communicates the value produced at the output of a functional unit directly to the input of the functional unit that needs it. For instance, Figure 1.2 depicts the execution of two instructions I_0 and I_1, from which the latter needs the result of the former. When I_1 reads the register *r0* at cycle 2, it retrieves stale data, as the valid ones will be updated only after cycle 4 from I_0. However, thanks to the bypass network, I_1 can still execute at cycle 3 with valid data that are promoted from the previous cycle to the equivalent functional unit.

1.2.2.4 Memory - M

Memory instructions calculate their memory address during *Execution* and then probe memory in this stage. Memory accesses happen in the form of *read/write* requests to the *memory hierarchy*. In modern processors, the memory hierarchy is composed by a set of memory caches [Smi82] organized in levels interleaving between the pipeline and the main memory. The reason of this escalated organization of memories is to hide the high latency needed to access the main memory (several hundreds of processor cycles). Ideally, if the register file could be very large, all data would reside in the processor's registers and no memory access would be needed. Evidently, an explosion to the number of architectural registers would increase complexity and eventually register-access latency to an unaffordable level. Caches are used to bridge this gap between the processor and the main memory, as they are the second fastest memory element after registers. Practically, the closer a cache memory is to the pipeline, the smaller and the faster it is.

The rationale of caches is based on the existence of *locality* in memory references. That is, the typical tendency of processors to access the same set of memory locations repetitively over a short period of time. Two different types of locality drive the operation of caches, the *temporal* and *spatial* locality. Based on the former, data that were fetched at a certain point are likely to be fetched again in the near future. While based on the latter, data residing nearby recently referenced memory locations are likely to be used soon.

According to these, any memory instruction accessing data not present in the L1 cache (miss) will bring them from the L2 cache. Whenever said data are not found either in the L2, they will be brought from lower levels of the memory hierarchy, including the main memory, i.e. the lowest one. Since caches have a fixed size, that becomes more moderate the closer they are to the pipeline, embedding new data may require evicting some others that already reside in the cache. Data that are evicted from a certain cache-level are propagated to lower levels in the memory hierarchy, till finding the appropriate location to be stored. A dedicated algorithm e.g., *Least Recently Used* (LRU), is used to coordinate the flow of data from the highest to the lowest memory level. Eventually, a well-orchestrated memory organization effectively reduces memory-access latency by allowing memory instructions to find their needed data in a memory-level as close as possible to the pipeline. An organization with two levels of cache is shown in

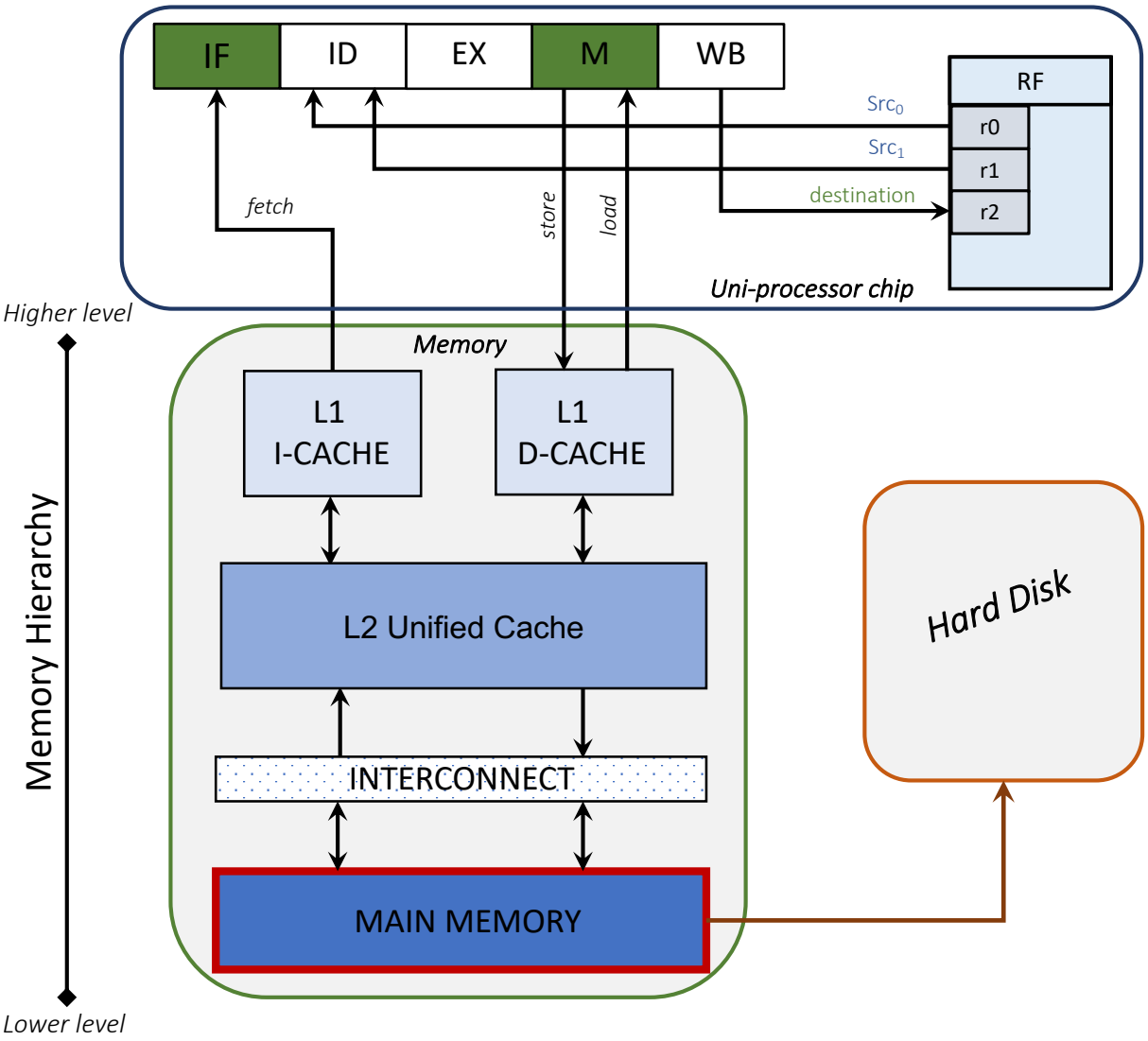


Figure 1.3 – Memory hierarchy of three levels. Access latency increases from higher to lower.

Figure 1.3. The first level of cache is the smallest and the fastest one. Then, by going down the hierarchy, size increases but so does the access latency.

1.2.2.5 Write-back - WB

During this stage, instructions write their result (either loaded from memory or computed) to the correspondent register. The *Write-back* stage brings the end on an instruction's execution. After that point, any changes that the instruction has made in the hardware level are also visible from the software level.

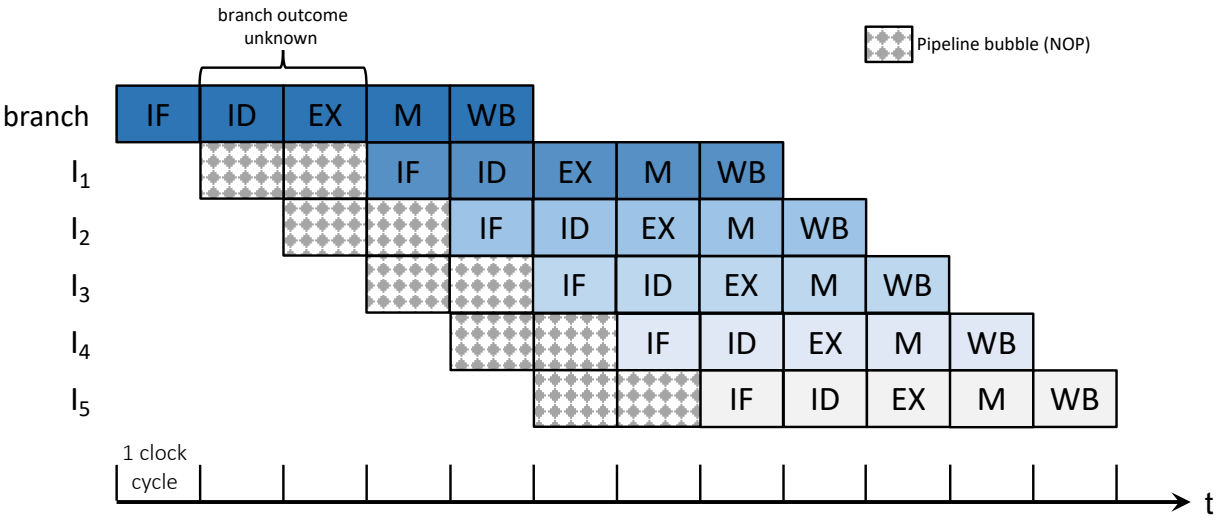


Figure 1.4 – Control-flow hazards induce pipeline "bubbles": instructions fetching stalls while the branch outcome remains unknown.

1.2.3 Limitations

The limitations of a pipeline organization are mainly related with the plausible dependencies between instructions. Dependencies appear when certain instructions influence the execution of later instructions in the pipeline so that the *dependent* ones are not executed in the following cycle because they will lead to incorrect results. In particular, these circumstances are called *pipeline hazards*[3] and are distinguished in two principal categories: *control* and *data* hazards. Below, we describe both of them.

1.2.3.1 Control Hazards

Control hazards are related to control-flow instructions such as branches, that define the next instruction to be executed through the direction they take. After branches leave the Fetch-stage, it is not definite which direction will be followed since the address of the next instruction to be fetched is defined/computed at *Execution*. As a result, the processor cannot continue fetching and processing instructions until the resolved branch clarify its direction. In order to avoid erroneous execution, the processor needs to insert *Bubbles* (no-operation) in the pipeline in order to stall the flow of instructions until the branch outcome is known. In the 5-stage pipeline that we described above, any instruction that follows a branch needs to be stalled for two cycles, i.e. until *EX* where the branch is resolved. We illustrate this in Figure 1.4, assuming the execution of five instructions from which the first one is a branch. In the general case, a

3. The term *hazard* was borrowed from digital electronic design, where it referred to the possible occurrence of an incorrect value for a short period of time [Huf57]

processor's pipeline needs to stall execution for *Fetch-to-Execute* cycles, which amounts to a few ***tens*** of cycles in modern-processors pipelines [4]. Thus, as we mentioned in Section 1.2.1, although it may be possible to increase performance by increasing the pipeline depth, control dependencies soon become the limiting factor [SC02].

1.2.3.2 Data Hazards

As described above, the values produced by instructions are effectively transmitted as soon as possible through the *Bypass network* to any subsequent instruction that potentially requires them as input operands. In these cases, the correct execution of the relevant instruction is at stake, since it may carry out its operation using wrong input data. In official terminology, such an event is referred as a *data hazard*. Indeed, a processor's pipeline suffers from a series of data hazards that are expressed by three types of *register-dependencies*:

1. ***Read-After-Write*** (RAW): They occur when an instruction depends on the result of a previous instruction and are also known as *flow-dependencies*.
2. ***Write-After-Read*** (WAR): Also referred as *anti-dependencies*, they occur when an instruction requires to read the value of a register that is later updated.
3. ***Write-After-Write*** (WAW): They are also called *output-dependencies* and appear when the ordering of instructions affects the final value of the register, since the relevant instructions modify the same register.

The first category represents the **true** data dependencies of instructions. They are referred as true because they are inherent to the program's logic and as a result they cannot be generally circumvented during pipeline execution. RAW dependencies are similar to a *producer-consumer* scheme: the *producer* is an instruction that computes a value and the *consumer* is an instruction that follows and uses this value as its source operand. The correctness of execution is guaranteed by unavoidably imposing delays to dependent instructions (consumers) until their needed values are ready. These delays may be significant when *producer* instructions come with a long execution delay, e.g. an instruction that depends on the value that a load instruction brings from the memory, may stall for even hundreds of cycles if the load instruction needs to reach the main memory in order to retrieve the needed data. Therefore, RAW dependencies are a major performance bottleneck.

On the other hand, WAR and WAW are identified as **false** dependencies because they arise from the reuse of registers by successive instructions that do not have any real data dependency between them. In that sense, if there were abundant or enough registers in the ISA to eliminate any "hazardous" reuse, they would simply not exist. In processors that execute instructions in program order both these dependencies are not an issue, since in that case they

4. The first generation of Intel Pentium 4 had a 20-stage pipeline [Hin+01].

are naturally resolved. However, as we will see later, it is possible to highly eliminate the delays occurred due to RAW hazards by executing instructions out of their sequential order (*out-of-order*). Under these circumstances, WAR and WAW dependencies are meaningful and require special treatment to be avoided. In Section 1.4.2.1 we will detail the established technique to remove false dependencies in out-of-order execution.

1.3 Critical Pipelining Developments

1.3.1 Branch Prediction

As illustrated in the previous section, the processor needs to insert a "bubble" in the pipeline instead of continuing fetching instructions during the time that a branch is unresolved. This action is necessary because the branch outcome (*taken* or *not taken*) defines the program-flow, i.e. the next instruction address (PC) to be fetched. To overcome this obstacle and avoid long pipeline-stalls, processors embody in their design special hardware structures that predict the direction of a branch before its actual execution. With *Branch Prediction* [Smi81], the processor is able to timely derive the PC of the address that should follow and fetching can continue.

Evidently, since branch prediction forms a speculation, it will be necessary to discard some instructions if the prediction is eventually not correct. For instance, in the baseline pipeline of Figure 1.4, when a *branch misprediction* occurs at *Execution* stage, two instructions will have been processed from the wrong path. However, since execution follows the program order, instructions in the wrong path cannot erroneously modify the architectural state. Thus, except from re-fetching instructions from the correct path, restoring the processor state is not required. Fortunately though, current branch predictors can achieve accuracy in the range of 0.9 to 0.99 [SM06; Pre16], yielding a tremendous increase in performance. As such, branch prediction has been an established method of speculative execution in high-performance cores.

This thesis work does not target higher performance through branch prediction and thus, we do not provide here details about state-of-the-art branch predictors [Sezb; Jim]. We will only partially describe the basic algorithm of the TAGE branch predictor [SM06] in Chapter 2, as it is used by the modern VTAGE value predictor [PS14b].

1.3.2 Superscalar Pipelining

Essentially, pipelining improves performance by increasing the number of instructions processed concurrently in the processor. The pipeline organizations depicted in the various Figures of Section 1.2 enable ***scalar*** execution, where at most one instruction per cycle can be fetched, decoded, executed etc. This aspect can be visually observed in Figure 1.1b. Even though the pipeline is shown to be fully-utilized (i.e. no "bubble" and pipeline stall is depicted), only 1

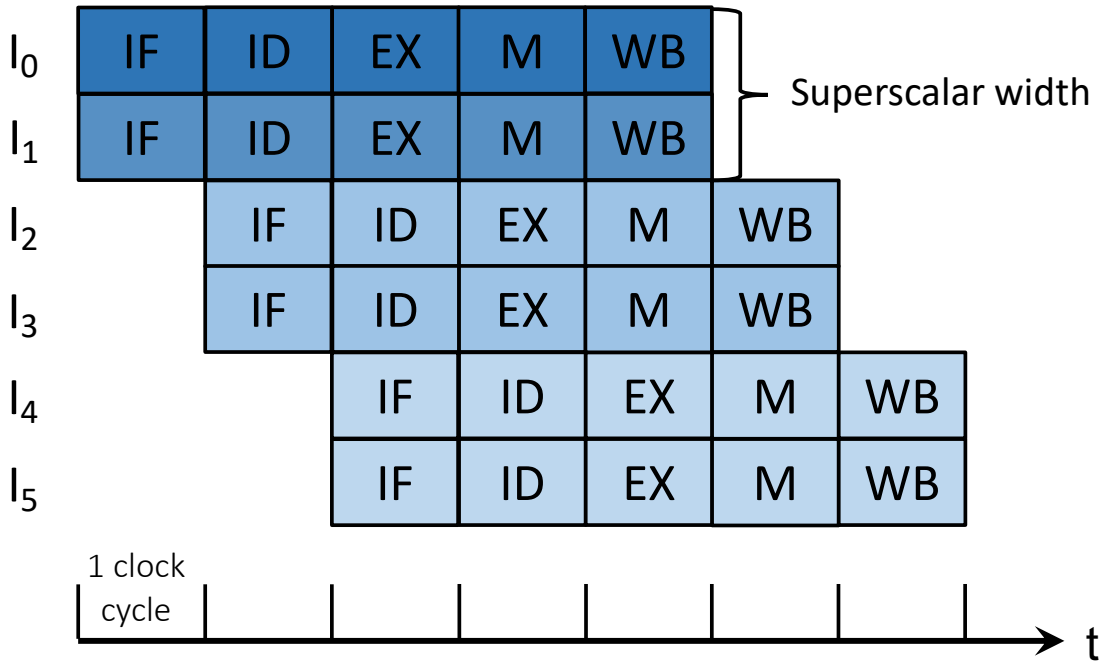


Figure 1.5 – Double width superscalar pipeline: two independent instructions can be processed per cycle.

instruction can be eventually completed per cycle. In the literature, this event is traditionally dubbed the "*Flynn's bottleneck*" [TF70]:

> *If only one instruction is fetched per cycle, no more than one instruction will be executed per cycle.*

Superscalar processors [SS95] were introduced to remedy this issue. The motivating idea is that within an instruction stream, it is often the case that at least two contiguous instructions are independent. As a result, the processor can apparently use multiple "*parallel pipelines*" in order to treat subsequent and independent instructions simultaneously at each stage. For instance, Figure 1.5 depicts a superscalar pipeline with a width equal to 2, able to process six independent instructions in pairs. In this way, all six instructions are completed in 7 cycles, while in the scalar pipeline of Figure 1.1b (equivalently composed of five stages) they would require 10 cycles. Performance-wise speaking, this means that the rate of executed *Instructions Per Cycle* or *IPC* increases from 0.6 to 0.85. Consequently, throughput has the potential to increase proportionally to the freedom degrees that instructions exhibit with their "neighbors", as dependent ones will still need to stall, not exploiting the available superscalar width.

Nonetheless, such a configuration comes inevitably with a great cost, since each stage needs to manipulate *superscalar-width* instructions. More specifically, fetching few more in-

structions per cycle may seem straightforward: more bytes will be fetched from the instruction cache. On the other hand, the decode stage has now to decode several instructions and discover any dependencies between them to determine the ones that can execute concurrently. Furthermore, assuming that the amount of functional units is simply duplicated to enable parallel execution, the complexity of the bypass network will have to grow quadratically to the superscalar-width so that in-flight values can be effectively broadcasted. Accordingly, more read/write ports will have to be provisioned both for the register file and the L1 data-cache to allow more instructions to proceed its cycle in any interaction with these fundamental structures. Considering that only for the register file the additional ports that are needed lead to a super-linear increase of its area and power consumption [ZK98], one can estimate the range of the anticipated explosion in the overall energy consumption.

Additionally, even though such a superscalar organization provides the potential of higher throughput, its perspective is limited only to neighboring instructions, failing to completely exploit the independence between instructions that may be away within the instruction stream. In other words, as instructions are treated in-order, independent instructions need to be consecutive in order to be processed concurrently. As such, regardless of the employed superscalar-width, an instruction will still delay the execution of other instructions that are independent but are found later in the program. Considering also the hardware cost that accompanies a superscalar structure, it becomes inefficient to keep scaling the pipeline when still following the imposed program-order of execution. Out-of-order (OoO) execution [Tom67] was introduced as a means to overcome this limitation and ever since it has massively dominated the design of modern general-purpose processors. Below, a comprehensive analysis of this scheme is provided.

1.4 Out-of-Order Execution

1.4.1 Basic Idea

Despite the fact that a program's low-level instructions have highly sequential semantics, many of them can be carried out in any order because their execution does not require as an input any datum produced by previous (in program order) instructions. This capability is referred to as *Instruction Level Parallelism* or shortly ILP. As we mentioned above, superscalar pipelining already exploits some of the available ILP, but the undergoing parallelism is bound only to adjacent instructions. In order to fully exploit a program's ILP, modern processors employ a custom design that allows them to re-order instructions in hardware and execute them *out-of-order* (OoO). Note that the compiler can not undertake this re-arrange of the instructions because it is driven by their true data dependencies (RAW) that are found in the hardware level

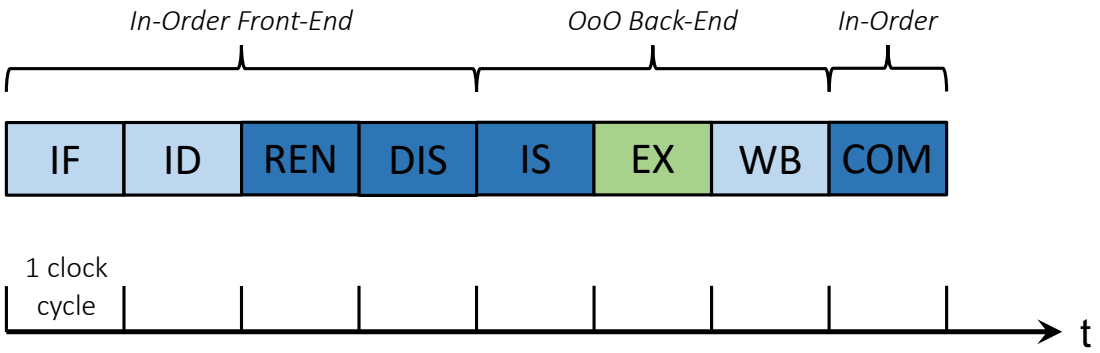


Figure 1.6 – Overview of a pipeline featuring out-of-order execution, marked in light green. New stages that will be described are shown in blue.

and can only be identified by the processor.

Although the processor may eventually execute the instructions in different order, the modifications that are applied on the architectural state should be visible to the software in the imposed program order, so that the program's semantics are preserved. That is, the control and data dependencies between instructions need to be respected. Essentially, instructions are still fetched and decoded sequentially, but then executed out-of-order and at the end, they retire/commit according to the program order. As such, the pipeline is separated in three central parts: the in-order front-end, the OoO back-end and the in-order commit phase. Both the front-end and the back-end of the pipeline are accordingly composed of different pipeline stages that are necessary for the smooth flow of instructions before and after the OoO execution. An overview of this scheme is depicted in Figure 1.6. The additional stages that are inserted in the pipeline compared to the one of Figure 1.5 are denoted in dark blue. Also, the current OoO execution stage is shown now in light green, to differentiate it from its earlier in-order form. In the following paragraphs, we describe these extra stages together with all major structures that processors need to employ in order to enable OoO execution in such a pipeline.

1.4.2 The In-Order Aspects of the Pipeline

1.4.2.1 Register Renaming - REN

In Section 1.2.3.2, we described the three different categories of data dependencies between instructions: RAW, WAR and WAW. As we mentioned, WAR and WAW are dubbed false dependencies because they are solely induced from the limited number of architectural registers rather than any actual data dependency between instructions. Previously, that instructions were executed with respect to their sequence in the program, false dependencies were not an issue by construction. However, when it comes to OoO execution, these dependencies become

a major impediment if they are not handled appropriately. For instance, consider two instructions, namely I_0 and I_5, that perform some independent operation, in the sense that the result of I_0 is not required from I_5 as an input for execution. When these instructions are executed in-order, I_0 will be always processed before I_5. Thus, even if their destination register is the same, no conflict exists. On the other hand, when OoO execution is employed, although the instructions are independent, they will still have to respect their order because their destination registers collide. That is, if only the compiler had more architectural registers to assign to each instruction, this kind of issues would have been avoided. However, extending the number of architectural registers is not a trivial procedure as it requires an essential re-form of the ISA.

In order to eliminate these dependencies, and thus uncover more ILP, without radically modifying the ISA, the amount of registers may be increased in the context of hardware but remain the same in the compiler level. To accomplish such an "artificial" augmentation, the processor needs to dynamically **rename** the *architectural* registers provided by the ISA into *physical* registers that are assembled in the *Physical Register File* (PRF). *Register Renaming* [Sim00] relies on the following principle: a new memory location is allocated, i.e. a physical register, whenever a variable, i.e. an architectural register, is reused. Therefore, the processor must have a large amount of physical registers that will be mapped to the architectural registers. Consequently, by revisiting the aforementioned example of instructions I_0 and I_5, I_0 could have its destination architectural register R0 mapped to the physical register P1. On the other hand, I_5 could also have the same destination register R0 mapped to the physical register P2. As in this way the two instructions will not eventually "write" in the same location, they can be processed in any order, without the hazard of having I_5 overwriting the result of I_0.

Conventionally, renaming is performed by using a *Free List* of the physical registers not mapped yet and a *Rename Map* that keeps the mapping of architectural to physical registers. For every instruction, a new physical register is removed from the *Free List* and linked to its architectural destination register. The *Rename Map* saves this link so that dependent instructions that follow will use this particular physical register as their source register. Consequently, each instruction performs also a table-lookup in order to discover the physical registers that hold the data of its source architectural registers.

However, this mapping is initially *speculative* by default, and therefore, not visible to the software, as the engaged instructions may be in the wrong path of a mispredicted branch. Therefore, a *Commit Rename Map* is also used to hold the non-speculative mappings and it is updated when an instruction completes its execution and is then removed from the pipeline. Note however, that a physical register is re-inserted into the *Free List* only when a younger instruction that has the same architectural destination register, and therefore the latest mapping, leaves the pipeline.

Similarly to *Fetch* and *Decode*, *Rename* needs to happen in-order so that the sequential

semantics of the program are respected. One (or maybe more) extra stage is inserted in the pipeline to cover the total procedure of renaming. Usually, Rename follows the Decode stage, as depicted in the pipeline of Figure 1.6.

1.4.2.2 Instruction Dispatch & Commit - DIS/COM

Once an instruction has finished renaming, it is **dispatched** into two different structures: the *Reorder Buffer* (ROB) and the *Instruction Queue* (IQ). A dedicated phase in the pipeline, namely the *Dispatch* stage (i.e. the *DIS* acronym in the pipeline of Figure 1.6), undertakes to record instructions in these two buffers that are essential to the valid (i.e. faithful to the program's sequential semantics) execution of instructions in an OoO manner.

More specifically, the IQ is a fully associative buffer that defines whether an instruction is ready for execution. Therefore, it is also referred as the processor's *scheduler* and represents the major enabling factor of OoO execution. Shortly, an instruction is marked as *ready for execution* when its source registers are ready to be used. Note that we will fully describe the operation of the IQ in the next Section.

Although an instruction may be executed whenever possible, as defined by the IQ, it can only modify the architectural state that is visible to the software according to the imposed program order. To achieve this, a dedicated *First-In-First-Out* (FIFO) queue is used, namely the ROB, in order to keep the order that instructions flow through the Rename stage, i.e. their sequential order. Then, instructions are allowed to **commit** their modifications to the architectural state (such as to modify the Commit Rename Map), only when they have reached the head of the ROB and are marked as *executed*. The *Commit* stage (i.e. shown as *COM* in our pipeline of Figure 1.6) is charged to remove completed instructions from the head of the ROB in program order. This stage is found at the end of the pipeline and practically represents the completion of an instruction, that is visible to the software. In other words, the ROB holds all the speculative state of execution and only the entry in the head represents the visible architectural state. As it keeps all the in-flight instructions of the pipeline, ROB is also called the *instruction window*. Furthermore, due to its FIFO manner, ROB allows to easily define all younger instructions that are in the wrong path after a branch is resolved. Simply, those instructions are found in the entries after the one that keeps the corresponding branch and can be simply removed, while the older instructions remain intact.

1.4.3 The Out-Of-Order Execution Engine

1.4.3.1 Issue, Execute and Write-back

These three stages compose the OoO execution core of the processor. In Figure 1.6 we mark them as the OoO back-end of the pipeline, where *IS* refers to the new *Issue* stage. As

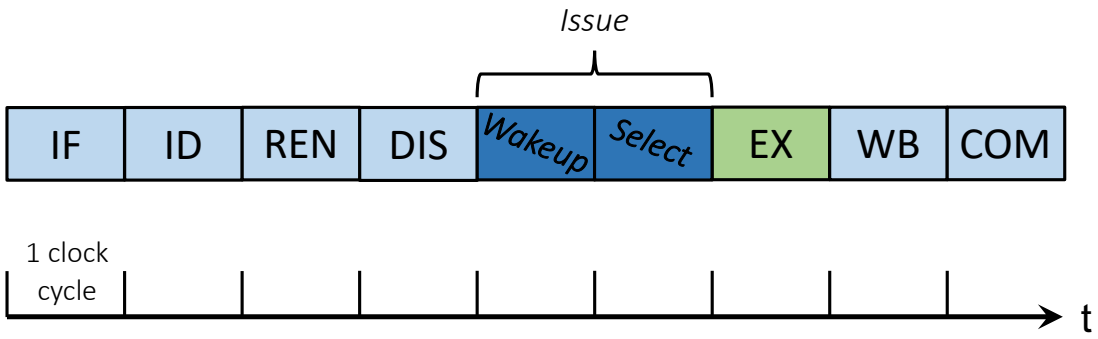


Figure 1.7 – A possible separation of the *Issue* stage into *Wakeup* & *Select*.

we mentioned above, instructions flow sequentially from the *Rename* stage to the IQ. After reaching the IQ, they are waiting for the moment that they will be allowed to perform their operation in an appropriate *Functional Unit* (FU). During the *Issue* stage, the IQ is responsible for scheduling instructions for execution by *issuing* them to an available FU. The number of instructions that can be **issued** per cycle is defined as the *issue-width* of the processor and it is not necessarily identical to the superscalar-width of the pipeline, as the latter practically specifies the amount of instructions that can enter the pipeline (i.e. fetched from memory) within a cycle. Assuming an *issue-width* equal to n, some logic circuit scans the IQ to discover the instructions whose sources are ready either in the PRF or the bypass network. Based on the FU availability, *ready instructions* are finally scheduled for execution.

In particular, the *Issue* stage can be further separated into two different phases of *Wakeup* and *Select*, as shown in the pipeline of Figure 1.7. During the *Select* phase, the IQ uses a heuristic to choose the n ready instructions that will be executed. Initially, the ready instructions that are considered are those that need to execute in a FU that will be available the next cycle. Then, ready instructions are usually selected in age order. Once a ready instruction is selected, its destination identifiers are broadcast to the IQ to inform any potentially dependent instruction about the future availability of the result, based on the instruction latency. As a result, during the *WakeUp* phase, the corresponding instructions are marked as ready for execution and are therefore available for selection. When instructions finish execution, they write their results in the PRF and also forward them to the bypass network. Note that an extra stage for the retrieval of instruction source operands could be also inserted in the pipeline of Figure 1.7 before the *Execution* stage.

As implied in the previous paragraph, instructions are executed with various latencies, taking up to as many cycles as they need, without stalling non-dependent subsequent instructions. This is possible due to the construction of the OoO execution core that decouples instruction sequencing from scheduling/execution, not allowing instruction latency to affect the duration

of a single clock cycle. Conversely, the delay of the *Issue* stage is considered a major ruler of the clock frequency, as the double-phase scanning of the IQ makes issuing relatively slow. Indeed, the IQ size has not substantially increased over processor generations to avoid the inevitable growth of the issue-delay that would accordingly lead to a longer cycle-time for the whole pipeline. Furthermore, although the common logic suggests that increasing the issue-width can lead to higher throughput, the complexity of a fully bypass network scales quadratically with the issue width [PJS97], making it soon impractical [STR02]. Therefore, similarly to the IQ size, the issue-width has undergone marginal increases [5]. As it follows, improving processors performance by directly expanding the OoO execution engine is restricted by construction. Consequently, devising techniques that can harvest higher sequential performance without increasing the complexity of the out-of-order core unaffordably has been a constant challenge of computer architects.

1.4.3.2 Manipulation of Memory Instructions

Memory Disambiguation Mechanisms

In the previous Section we described how instructions flowing into the out-of-order execution engine are dynamically scheduled and eventually executed in different order using two basic regulating structures, the ROB and the IQ. Readers may have already noticed that the *Memory* stage of the superscalar pipeline in Figure 1.5 has been removed from the pipelines for OoO execution in Figures 1.6 and 1.7. The logic behind this fact is that the decoupled OoO execution we have described allows us now to carry out memory accesses during the *Execution* stage, without forcing all instructions to (unnecessarily) pass through the *Memory* stage. Practically, the execution of a memory instruction happens in two steps: first the computation of the memory address and then the read/write access to the memory. The memory-address computation may be treated like any other instruction and is subject to the same issues (i.e. scheduling, operands forwarding etc). However, the actual memory access requires special treatment when one needs to maintain the program's semantics for two specific reasons that we describe below.

The first reason relies on the fact that execution of instructions that follow a predicted branch remains speculative till the branch is resolved at the *Execution* stage. Loads can fearlessly execute speculatively, as they can be seamlessly removed from the pipeline on a branch misprediction without leaving execution remnants that can not be modified in the correct control path. That is, their destination register will be effectively overwritten with the appropriate data when the correct execution take place. On the other hand, the same is not valid for store instructions because when they are executed in the wrong control path, their data are irreversibly written in

5. The Alpha 21264 microprocessor of 1999 [Kes99] could issue 6 instructions per cycle, while the Intel microarchitecture with codename Skylake [Dow+17] of 2017 can respectively issue 8 instructions per cycle.

the memory destroying the program's semantics.

The second reason arises from the equivalent RAW dependencies in the memory-address level between memory instructions. Register RAW dependencies are flawlessly imposed between the engaged instructions by simply not allowing "consumer" instructions to execute before "producer" instructions "prepare" the needed data (see issuing of instructions in the previous Section). When an equivalent producer-consumer pattern appears between a store and a subsequent load instruction due to accessing the same memory address (i.e. a store instruction writes data in the memory that will be afterwards retrieved from a load instruction), executing the load before the corresponding store instruction will make the former to fetch "stale" data from the memory. However, this incorrect re-ordering can not be prevented before both instructions compute their respective memory address, i.e. at the *Execution* stage.

Consequently, the design we have presented so far would be erroneous if no techniques had been employed to perform *memory disambiguation*, i.e. to guarantee the valid execution of memory accessing instructions. In general, to efficiently defeat these issues, two more buffers are also required: the *Load Queue* (LQ) and the *Store Queue* (SQ). During the stage of *Dispatch*, store instructions are buffered in a FIFO fashion (similarly to ROB) in the SQ. Each entry in the SQ has two basic fields, one for the memory address to be accessed and one for the corresponding data to be written in the memory. Store instructions are issued to the out-of-order execution engine when their sources are ready. At the *Execution* stage, they compute their memory address and record it in their corresponding entry in the SQ, together with their data. However, they do not yet perform their write-access to the memory hierarchy. Only after a store instruction has committed, can its data be safely written to the memory, without violating the sequential order. That is, the processor performs a post-commit write-back to the memory for committed stores from the head of the SQ and then it removes their entries [6]. Since the SQ is age-ordered, it is guaranteed that although store instructions may be treated in different order in the execution core, memory writes will be certainly performed in the imposed program order.

Accordingly, the LQ is similar in structure and function with the SQ, buffering in age order the memory address and data of load instructions. The LQ serves two purposes. Firstly, when a load executes, it computes its memory address and probes the SQ in order to find any previous (in program order) store instruction that will write its data in an overpaying memory location. If such a collision exist, the load should stall, because at this moment the data are still stale, preserving execution correctness. However, if their memory addresses are completely identical, the load can retrieve its needed data from the corresponding SQ-entry that tracks its "producer" store. Such an event is called *Store-To-Load Forwarding* (STLF) and allows loads to execute much faster without even accessing the memory system.

Nevertheless, due to the out-of-order nature of execution, a load may have its sources ready

6. Committed stores at this point will have already been removed from the ROB.

before a conflicting store. Since the memory addresses are available only after execution, this load can not capture its dependence and may violate the imposed memory order by executing earlier from the corresponding store. Such a load is then identified as a memory violator. In these cases, the second purpose of the LQ is to allow stores to discover potential memory-order violations. In particular, when a store executes, it uses its memory address to probe the LQ and find these violators. In that event, like on a branch misprediction, the simplest way to recover is to *squash* (i.e. evict) all instructions younger than the load-violator from the pipeline. Then, the *Fetch* stage will have to restart execution from this load instruction. Note that in modern out-of-order processors, both the LQ and the SQ are usually combined in a single structure that holds separately load and store instructions respectively and is called *Load Store Queue* (LSQ). The LSQ is typically a *Content Addressable Memory* (CAM) that is indexed using a memory address.

Memory Dependence Prediction

The memory disambiguation mechanisms that we described above detect true dependencies between memory operations at execution time and allow the processor to recover when a dependence has not been respected. Yet, a final issue lies on the accepted tolerance to these violations, as their penalty appears to be similar to a branch misprediction. Performance-wise speaking, such violations should be limited, but ideally without conservatively stalling a load until all previous store instructions have completed. As it follows, out-of-order processors need to essentially maintain some balance between the cost of stalling loads and the cost of memory-order violations.

Memory dependence prediction (MDP) [Mos+97; Mos98] was introduced to address this challenge, allowing processors to exploit most of the potential of executing memory operations out-of-order. The motivating observation is that the dynamic instances of a specific load instructions tend to depend on the dynamic instances of a specific store instruction, respectively. Therefore, a hardware structure can be used to monitor and eventually to "remember" this repeated behavior, so that dynamic loads will be able to predict their dependencies. That is, the processor **predicts** if allowing a load instruction to execute would cause a memory-order violation and delays its execution only under these circumstances. Although we do not detail the different algorithms that have been proposed, we solely mention here for the readers information the *Store Sets* predictor [CE98], which appears to be one of the most efficient models in the literature.

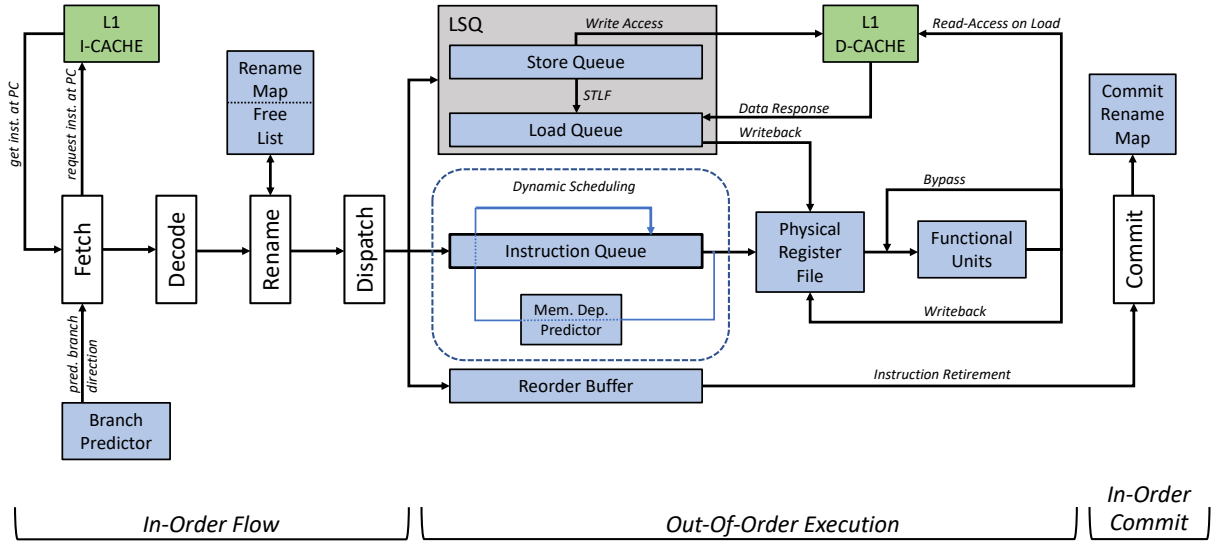


Figure 1.8 – Top-down view of the organization of a modern out-of-order and superscalar processor.

1.4.4 Outline

Figure 1.8 presents the microarchitectural outline of a modern processor that performs out-of-order execution within a superscalar pipeline. Instructions flow in-order from *Fetch* to *Dispatch*, then are executed out-of-order in the core and finally *Commit*, leaving the pipeline according to the program order, i.e. same as they entered. Readers should note that some links are omitted for clarity, such as the scan of the load/store queues during execution or the update of the branch predictor at commit. Nonetheless, the figure contains all vital hardware components, together with their major interactions.

As illustrated, the out-of-order execution engine relies on several structures to allow instructions to be treated dynamically in different order, while preserving the program's semantics, i.e., so that the data and control-flow dependencies between instructions are respected. Although its organization appears to be complex, it remains highly effective in terms of sequential performance. In practice, sequential performance increases due to the higher rates of ILP that the processor can extract, by decoupling instructions from their inherent execution dependencies. In particular, branch prediction allows to overcome the obstacle of control-flow dependencies and prevents instructions fetching from stalling. Then, register renaming eliminates false register-name dependencies between instructions by mapping architectural to physical registers. As such, the pipeline frontend remains full with several instructions flowing in the out-of-order execution engine.

At that critical moment, instructions are scheduled for execution according to their data and

memory dependencies. A memory dependence predictor is typically used to predict conflicting load instructions and to delay the execution only of loads that were predicted dependent on a previous store that is not already finished. Still, data dependencies remain a major barrier of instructions parallelism. Nonetheless, data dependencies can also be mitigated by adding another more advanced level of speculation. *Data speculation* refers to the speculative determination and propagation of instruction results before their execution, with the ultimate goal of collapsing RAW dependencies between instructions. In this thesis, we distinguish two prominent techniques that enable *data speculation*: *value prediction* and *load-address prediction*. In the next Chapter, we will describe these techniques, as well as some related previous work on this domain.

CHAPTER 2

ADVANCED SPECULATION IN OUT-OF-ORDER SUPERSCALAR PROCESSORS

2.1 Introduction

The increasing demand for high-performance computing has led to the emergence of multicore processors and multithreading cores that excel in increasing computation throughput. Unfortunately, many applications can not benefit from these designs because their code features long regions that cannot be parallelized. As argued by Hill and Marty [HM08], Amdahl's Law [Amd67] remains valid even in the multicore. Therefore, improving the *sequential* performance of processors continues to be critical for the development of highly efficient multi-perspective cores.

Modern high-end processors achieve good sequential performance by concurrently executing as many instructions as possible. To do so, they combine wide superscalar pipelines and out-of-order execution, so that they can process several instructions in parallel. As we detailed in the previous Chapter, out-of-order execution coupled with in-order commit allows to remove the unnecessary in-order execution constraints that are not imposed by the program's semantics. However, the instruction-level parallelism (ILP) that processors can exploit is still considerably limited by various bottlenecks, such as control-flow hazards and true data dependencies between instructions. Fortunately, the impact of control-flow dependencies has been tackled with the employment of branch prediction. The introduction of this technique of speculative execution yields a tremendous increase in performance by practically keeping the pipeline full, even in the presence of control hazards.

Yet, although branch prediction allows to effectively exploit the pipeline's capacity, in-flight instructions may experience significant delays due to their dependencies on the results of other instructions (RAW). That is, even by allowing more and more instructions to flow in the out-of-order execution engine, IPC is natively restrained by the presence of RAW dependencies. As it follows, another advanced level of speculation, beyond branch prediction, is highly re-

quired if one wants to bypass this obstacle for further parallelization in instructions execution. In this document, we will be using the term *Advanced Speculation* in order to generally refer to heuristics that processors can adopt in order to *artificially* increase the ILP by *breaking* RAW dependencies. In particular, the contribution of this thesis relies on proposed techniques that fall into two separate categories of advanced speculation: *Value Prediction* and *Load-Address Prediction*. In the following Chapter, we will present the main models of these two categories, that also inspired this thesis work.

2.2 Value Prediction

2.2.1 Key Idea

In mid 90's, Gabbay and Mendelson [Gab96] and Lipasti et al. [LWS96; LS96] independently proposed the technique of value prediction (VP) to attack the obstacle of true data-flow dependencies. VP is driven by the observation that when certain instructions are executed repetitively (e.g. an instruction found in a loop or in a frequently called function), their result stream exposes zero to low variance (i.e. an instruction's result remains always the same) or follows a relatively easy to recognize pattern (e.g. for each dynamic execution the result increases/decreases by a fixed value). This attribute was defined by Lipasti et al. [LWS96] as *value locality*. In these cases, the repeatable behavior of such instructions is said to be *predictable*, as it can be learnt by a finite-state machine.

As it follows, one can augment the microarchitecture of a processor with a dedicated hardware structure that tracks and therefore "remembers" any potential patterns in instruction results. This structure is commonly called *value predictor* and it is added to the frontend of the pipeline. Conventionally, the predictor is accessed for each instruction by using some microarchitecture state information (e.g. the instruction PC). The provided prediction acts temporarily like the result of the instruction, in the sense that it can be sourced by dependent instructions that follow. In this way, *consumer* instructions can execute concurrently to their *producer* counterparts, collapsing completely the underlying RAW dependence and therefore increasing the ILP. Similarly to branch prediction, such a speculative execution needs to be followed by validation. Therefore, when the predicted instructions eventually compute their *actual result*, they use it to validate their prediction and to trigger recovery actions if necessary. As a final step also, the predictor is effectively updated with the actual result, before the predicted instruction leave the pipeline.

In Figure 2.1, we abstractly present the execution of a random instruction stream in the conventional way and in a processor featuring value prediction. As illustrated, the chain of data dependencies between instructions leads roughly to a *ladder-style* of execution, i.e. any

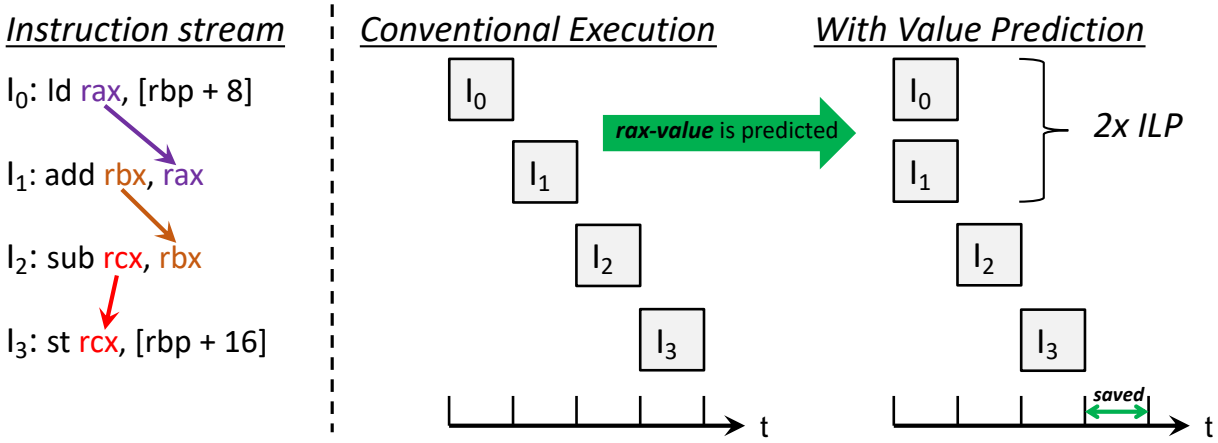


Figure 2.1 – Representation of execution with and without value prediction to show the impact on the ILP.

"consumer" can only execute after its "producer". Assuming that the result of instruction I_0 (value of the *rax* register in Figure 2.1) follows a predictable pattern, a VP-augmented processor could predict this value. In this case, the execution of I_1 can be maintained in parallel with that of I_0, with the former to be using the predicted value of the corresponding source register. Therefore, as it is shown in Figure 2.1, the ILP doubles and thus one unit of time is saved.

2.2.2 Constant Challenges

2.2.2.1 Detection of Predictable Value Patterns

Compared to conditional-branch prediction, value prediction is arguably considered as a more advanced technique for speculative execution, due to the natively unbounded range of the predicted values. That is, as a conditional branch can only have two possible outcomes (i.e. *taken* and *not taken*), its predicted outcome is accordingly binary, while on value prediction, no definite value range can apply to the generated predictions. Nevertheless, prediction algorithms for value prediction can draw from a wealth of studies on branch prediction. Particularly, we will later show in Section 2.2.4.1 how the state-of-the-art value predictor VTAGE [PS14b] is designed by assuming the prediction logic of the also modern TAGE [SM06] branch predictor. Essentially, the majority of improvements in the operation of branch predictors has been obtained from the efficient detection of correlations inside the local and the global branch history [YP92], i.e. the outcome history of either a single branch or a sequence of branches that are subsequently encountered during execution. Similarly, value prediction aims to uncover and exploit potential correlations in the sequence of instruction results.

As we already mentioned above, instruction results can be *predictable* when they fall into

recognizable patterns. This means that in these cases it is possible to devise a prediction algorithm that allows the value predictor to re-produce this pattern. Sazeides and Smith conducted a comprehensive analysis of *value predictability* by distinguishing four predictable categories of value sequences: *constant*, *strided*, *repeated strided* and *repeated non-strided* [SS97]. Indeed, since the very first introduction of value prediction [Gab96; LWS96; LS96], the value predictors that have been proposed aim to capture one or more of these value patterns. We will show this aspect in Section 2.2.4, where we will describe their main representatives. Nonetheless, in Chapter 4 we will show that the analysis of value sequences by Sazeides and Smith [SS97] is not exhaustive and that more potential exists if values are examined strictly sequentially.

2.2.2.2 Increase of Coverage

Predicting the outcome of branch instructions is necessary if one wants the unrestricted flow of instructions in a superscalar pipeline. In that sense, when branch prediction is employed, all generated branch predictions are always used in the pipeline, since naturally there is at least 50% possibility that the prediction will be correct (i.e. taken or not-taken). Nevertheless, in order to maintain high-accuracy, modern branch predictors [Pre16] typically employ saturating counters to track the confidence of their predictions and update their entries accordingly.

On the other hand, value prediction allows more already in-flight instructions to issue and execute concurrently in the core, directly increasing execution parallelism. However, unlike branch prediction, the accuracy of predicting the result of an instruction has not a lower bound, since there is no fixed range in the predicted values. As a value misprediction entails a certain cost that can be at least equal to the cost of a branch misprediction, predictors conventionally discriminate their predictions based on a confidence mechanism. That is, similarly to branch prediction, a saturating confidence counter is associated with each predictor entry that provides a prediction. As such, a predicted value is used in the pipeline only when it flows out of an entry that features a confidence counter that is saturated or has reached a certain threshold. Thereafter, at retire time, the corresponding counter is updated according to the prediction's correctness: it is incremented (if not saturated) when the prediction is correct and it is decremented or even reset (depending on the model) on the opposite case. Particularly, the recent state-of-the-art VTAGE [PS14b] value predictor (see section 2.2.4.1) has established the use of *Forward Probabilistic Counters* (FPC) [RZ06] for a highly-efficient confidence tracking.

As it appears, confidence mechanisms in value prediction are necessary to maintain an efficient balance of *accuracy* and *coverage*. The former term refers to the amount of correct predictions over the ones that were used by the pipeline. The second term is defined as the number of dynamic instructions that were eventually predicted divided by the total number of dynamic instructions eligible for value prediction (i.e. data-producing instructions). Both of them are usually two essential performance metrics of value predictors. In essence, high accuracy

is critical to avoid performance degradation. High coverage is always desirable but it is practically irrelevant in the presence of low accuracy. Thus, inevitably, value predictors set stringent, yet necessary, high-confidence criteria to safely preserve high accuracy. In Chapter 4 we will argue that the regular way of tightly keeping in the same predictor entry values and confidence counters, limits erroneously the prediction coverage, i.e. not in a way that it is necessary for preserving high accuracy.

2.2.3 Principles of a Realistic Implementation

Value prediction had initially gained lots of interest from the research community that led to the emergence of several prediction algorithms. We will describe the main models that have been devised in Section 2.2.4. However, in the late 90's, VP lost its large attractiveness due to implementation concerns. In particular, it was assumed that the integration of a value predictor in the pipeline would induce high complexity in critical parts of the out-of-order engine, that are already complex by construction. Even though predicting is relatively simple because it can happen in-order (e.g. in the pipeline frontend), both prediction validation and recovery would require meticulous tracking of the instructions during their out-of-order execution. First, the predicted value of any predicted instruction would need to be broadcasted to the functional units, so that right after execution, the corresponding instruction would be able to validate it. Second, the pipeline would need to track any instruction that actively used a predicted value, and on a value misprediction, would need to selectively re-execute only those instructions. Note that this is not the case in branch prediction, where any instruction that follows a mispredicted branch is definitely in the wrong control path and should be evicted.

However, recent studies [PS14b; PS15b] have introduced some specific design standards that allow to overcome these obstacles. These standards mainly concern the validation of predictions and the cost-effective use of the *physical register file* (PRF) for the needed communication of predictions to dependent instructions.

More specifically, Perais and Seznec showed in [PS14b] that by enforcing high-confidence requirements (discussed in Section 2.2.2.2), prediction validation can be performed in-order at commit time without impairing performance. By not performing validation at execution, no complex circuity is needed in the out-of-order engine for the propagation of prediction. A simple FIFO queue can be used to maintain predictions till commit, where an additional ALU can verify them simply in commit order. Furthermore, the pressure in the core can be further relaxed by employing a flush-based recovery. That is, on a misprediction, the pipeline squashes (i.e. evicts) all in-flight instructions that are younger than the mispredicted instruction from the ROB, regardless if they used the wrongly predicted value. Overall, prediction validation and pipeline squashing at commit increases the minimum misprediction penalty but it is much easier to implement, as it leaves the out-of-order engine intact.

Moreover, in the most common approach, predicted values are written in the PRF and then they are replaced after the normal execution of the relevant instruction. Therefore, some extra read/write ports should be added to the PRF to enable prediction writing and validation, respectively. Unfortunately, as the number of these ports is proportional to the pipeline width (i.e. the number of instructions that can be fetched and committed per cycle) it may lead to a prohibitive increase of PRF's complexity and power consumption. Nonetheless, Perais and Seznec examined these constraints in [PS15b] and proposed PRF banking as a solution. In a nutshell, since prediction and validation happen in-order, instructions of the same dispatch group can be forced to allocate their registers from the same register bank. With this equal distribution of instructions, the additional ports that are needed per register bank are reduced and therefore, complexity and power remain controllable. We refer readers to [PS15b] for a more detailed explanation of the benefits of PRF banking.

2.2.4 Main Prediction Models

Since the very first introduction of value prediction [Gab96; LWS96; LS96], there has been a plethora of works proposing prediction schemes that essentially aim to capture different value patterns (see Section 2.2.2.1). According to that, Sazeides and Smith categorize value predictors into two broad classes: *context-based* and *computation-based* [SS97]. Also, *hybrid* predictors that combine both these types are also considered in a separate category. In reality, this classification results from the value pattern(s) that the predictors intend to capture. Below, we will detail in a scale from the earliest and most fundamental to the latest and most efficient, the main representatives of these categories, by focusing on the respective advancements they introduced.

2.2.4.1 Context-Based Predictors

Context-based predictors identify patterns in the value history of a static instruction by leveraging a distinctive processor context. If the given context matches the one that is "recorded" in the predictor, the predictor has "learned" the value that the instruction produces following that context and provides it as the prediction. Thus, by construction, context-based value predictors can only predict values that have been already encountered during execution and the predictor effectively associated them with the corresponding context in its structure. We demystify the operation of context-based value predictors by describing the fundamental model of *last-value prediction* [LS96], the more advanced FCM and finally the state-of-the-art VTAGE [PS14b].

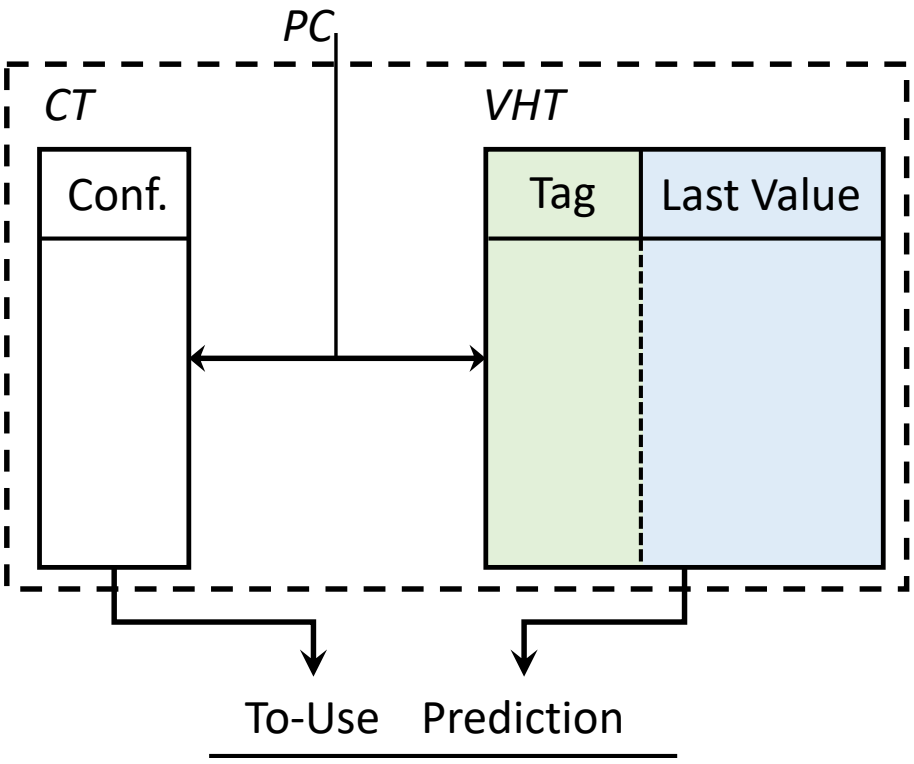


Figure 2.2 – The Last Value Predictor

Last-Value Predictors

As its name reveals, a *Last Value Predictor* (LVP) predicts that the result of the current execution of a static instruction is equal to the one produced by its previous execution. To do so, LVP records the value produced the last time a static instruction was executed and predicts this value in a future re-execution of the instruction. In practice, this heuristic forms the main principle of value prediction: track past results of an instruction's instances and then based on them, predict the result of future instances. We will see later that this logic is fundamental for many other schemes. Initially, Lipasti et al. introduced this model considering only load instructions [LWS96], and then, Lipasti and Shen generalized it for any data-producing instruction [LS96].

Figure 2.2 shows a general representation of a *last-value predictor*. Its main part is a *Value History Table* (VHT), where the predictor maps the *last results* produced by instructions. Each entry of the VHT holds the *last values* together with a tag. A second table, the *Classification Table* (CT), associates each entry of the VHT with a saturating confidence counter. At prediction time, a hash of the instruction's PC is used to index both of them. The result of a particular instruction is predicted when there is a tag-match in the VHT and the corresponding confidence counter is high enough. At retire time, the actual result is send to the VHT and the

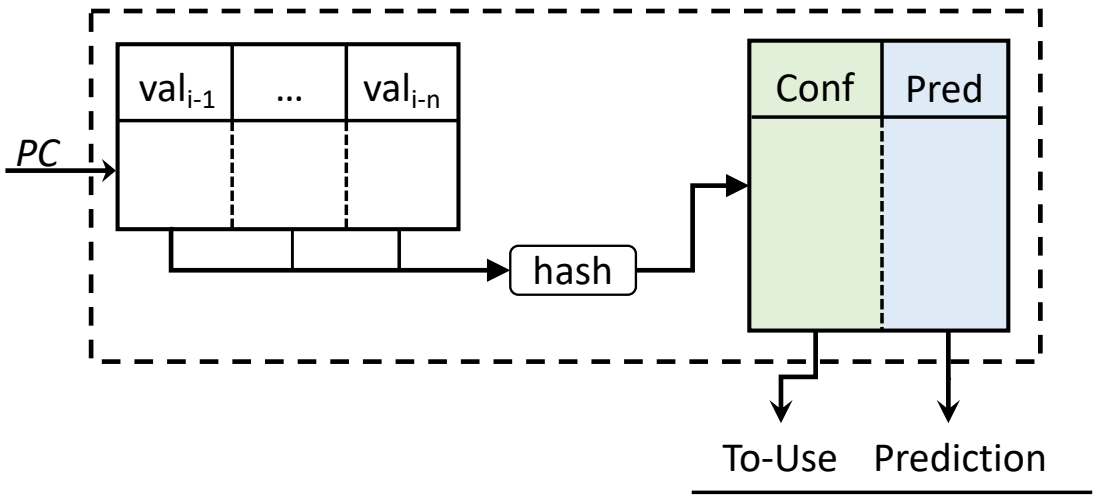


Figure 2.3 – A representation of the Finite Context Method Predictors

confidence counter is updated based on the correctness of the prediction. Consider that despite its apparent simplicity, the authors reported that LVP achieves 4.5% of average performance improvement in their framework [LS96].

A *last-value predictor* that follows this scheme relies its prediction upon a simple context: the last-seen value [1]. Therefore, it can practically predict only instructions with a *constant* value sequence. However, in the same study [LS96], the authors evaluated also the theoretical upper limit of the obtained performance when storing the last four results of each static instruction and assuming a perfect selector to choose during prediction. Certainly, such configuration achieve higher performance rates as it partially covers more value patterns. We will show how this aspect is addressed in the next two prediction models we describe below.

Finite Context Method Predictors

Predictors of this class typically use two structures. The first one is used to capture the local value history of an instruction. Then, this history is used to index the second structure that virtually provides the prediction. Therefore, unlike simple LVP, *Finite Context Method* (FCM) predictors consider more than one recent value in order to cover several value patterns, depending on the number of last values that they track. Sazeides and Smith detailed FCM predictors in two different studies [SS97; SS98]. In Figure 2.3 we depict the operation of n^{th}-order FCM predictors, i.e. generally tracking n recent values per static instruction.

Briefly, the structure that holds the local value history is accessed with the instruction PC

1. According to Sazeides and Smith *last-value predictors* perform the trivial "identity" computation to generate a prediction and thus they can be even considered computation-based [SS97].

to provide the values hat will be used for the computation of a hash. Said hash is then used to index the prediction table that eventually provides the predicted value. The prediction table may associate the predictions with a confidence mechanism [SS98] in order to filter out uncertain predictions. The same entry that eventually provided the prediction will be finally updated with the computed value after execution. Practically, each value-hash corresponds to an encountered value pattern for a particular instruction. Using this hash as a context, FCM predictors can provide the value that follows in the local value stream. However, consider that in a superscalar and multi-issue out-of-order processor, it is not guaranteed that at prediction time the previous instance of the instruction will have committed. Therefore, the local value history in the predictor may contain stale values that hinder the effective recognition of value patterns.

The Value Tagged Geometric Predictor

More recently, Perais and Seznec introduced the Value Tagged Geometric (VTAGE) predictor [PS14b], showing that leveraging control-flow history to correlate instruction results appears to be highly efficient. In particular, VTAGE is a direct adaptation of the indirect-branch predictor ITTAGE [SM06] to value prediction. That is, instead of predicting targets of indirect branches, that are also values, one can use the model of ITTAGE to actively predict instruction results. Since the ITTAGE itself is accordingly derived from the TAGE branch predictor [SM06], the VTAGE predictor is known as a *TAGE-like* value predictor.

In Figure 2.4 we show the general structure of a VTAGE predictor. It consists of a tagless table (VT0) that is accompanied by several other tagged tables or components. Assuming *N* tagged tables (VT1 to VTN), VTAGE is referred as a (1+N)-component predictor. The tagless table is accessed using the instruction PC and basically represents a simple LVP, as we described it above. On the other hand, the tagged tables are indexed by using a hash of the instruction PC with a certain number of bits from the global branch history. The history-length that is used in the hash computation for each table-rank follows a *geometric series*, i.e. two for VT1, four for VT2, eight for VT3 and so on.

At prediction time, all tagged tables are accessed in parallel in search of the matching entry. A tag-match may appear in more than a table, but eventually the table that provides the prediction is the one that was accessed using the largest branch-history length. If there is no match, then the predicted value is provided by the tagless table. All table-entries (either tagged or not) feature a saturating confidence counter, so that only high-confidence predictions are used in the pipeline. In particular, on a correct prediction, the confidence counter of the corresponding provider entry is increment but only under a certain probability, while on a misprediction the confidence is reset. Such a scheme allows VTAGE to achieve accuracy higher than 99,5% in the framework used in [PS14b]. At update time, except from the confidence update we described, a misprediction triggers two more actions. First, the entry's value is replaced by the

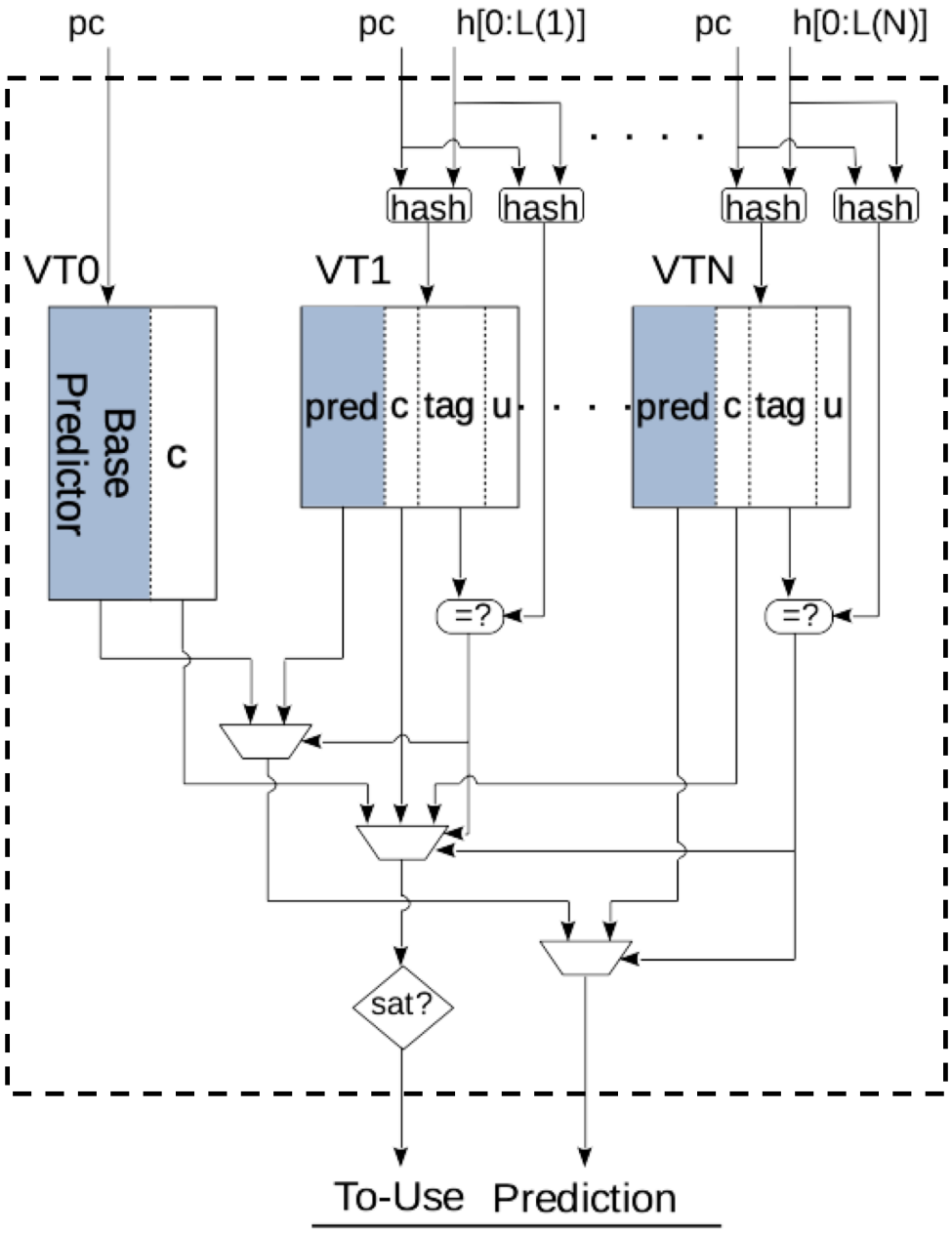


Figure 2.4 – The Value Tagged Geometric Predictor.

computed/committed one when the confidence counter is equal to zero. Second, a new entry is allocated in a higher-ranked table, i.e. that is indexed with a hash of longer branch history. The selection of the new entry is driven by a single auxiliary bit, the *u* bit, which is set when a prediction is correct and unset otherwise, revealing the usefulness of the prediction.

Essentially, the geometric-style use of the global branch history by VTAGE is an inher-

ited feature from the TAGE branch predictor [SM06] and allows VTAGE to identify correlation of an instruction's value with both close and distant branches. VTAGE can efficiently capture constant values but most importantly, it may also recognize repeated strided or non-strided value sequences when they can fit in the maximum history length that is employed. Indeed, let us assume the following value sequences, where comma-separated values represent the value/result of subsequent dynamic executions of the same static instruction:

Repeatable & Strided: 1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, ...
Repeatable & Non-Strided: 1, -8, 6, -4, 1, -8, 6, -4, 1, ...

If the values interchange with respect to different branch history lengths that are always shorter than the maximum used, potentially, the predictor will map them in different entries and will be gradually able to predict them. For instance, in the first case, the *value 5* is always followed by the the *value 1*. If this value switch correlates consistently with a branch, the corresponding instruction instances will repeatedly match with two distinct entries. Thus, the predictor will eventually establish their confidence (i.e. the confidence counter will be saturated) and future instances will be able to predict them. The predictor's behavior will be also similar for the second sequence. Nonetheless, as we will detail in Chapter 4, there exist another major value pattern consisting of interval-style repeated values that VTAGE can not efficiently capture.

2.2.4.2 Computation-Based Predictors

This class refers to predictors that *compute* the prediction by applying a specific function on the results produced by previous instances of a static instruction. According to the utilized function, computational predictors are therefore capable of predicting values that are not yet even encountered in previous executions.

Stride predictors [Gab96; NGS99; ZFC03] are good examples of this class. These predictors aim to capture value sequences that follow a strided pattern, i.e. instruction results that differ by a constant delta (i.e. a *stride*). Practically, the predictor "learns" the constant stride by which each dynamic value differs with its preceding counterpart. At prediction time, the predictor predicts this stride and *computes* the potential instruction value in function with the last value produced by the instruction. Therefore, stride predictors are generally based on two parts: one that tracks last values of instructions (basically an LVP-like structure) and the main one that provides the predicted stride.

The general scheme of a simple stride predictor is depicted in Figure 2.5. As illustrated, at prediction time, the predicted value is generated by using the obtained stride and the corresponding last value of the instruction. At update time, the stride is calculated using the computed and the stored last value. Then, the predictor updates both the stride and the last value

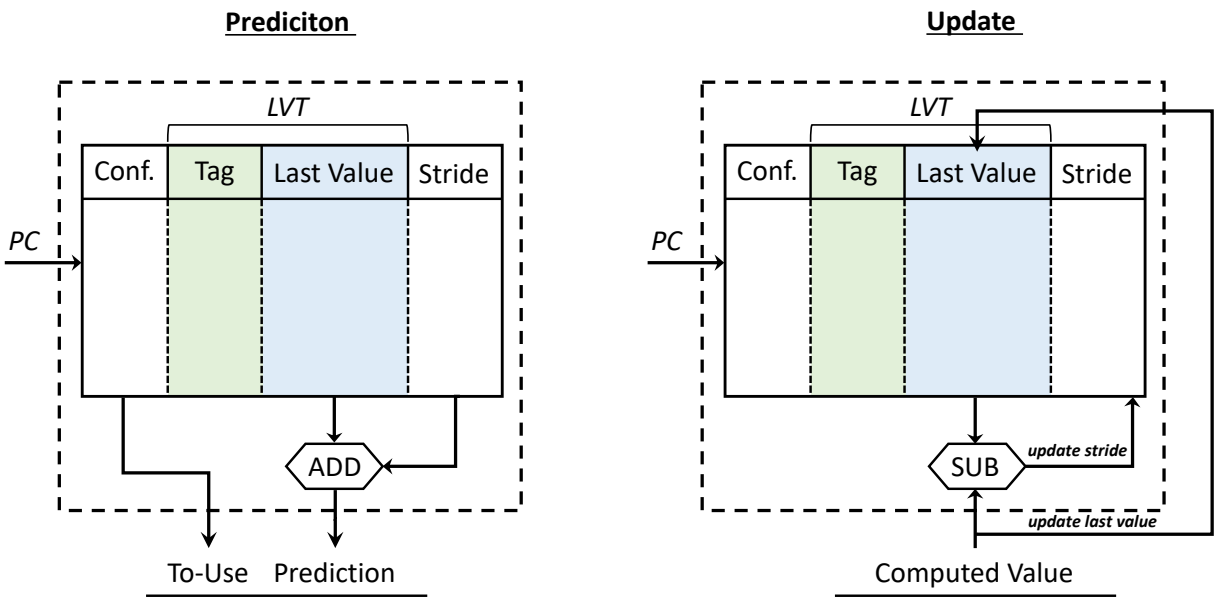


Figure 2.5 – Simple stride-based value prediction (one the left) and training/update on the right. LVT is used as an acronym for the *Last Value Table*.

of the respective fields. Stride-based predictors are able to predict only values that follow stride patterns. Note that constant values can be also considered strided, as they can be generated with a stride equal to zero. Therefore, these predictors may eventually overlap (in terms of value patterns that can be covered) significantly with last-value predictors.

Most, if not all, *computation-based* predictors rely on the value produced by the most recent instance of a particular static instruction in order to *compute* the value of the currently processed instance. However, as we already mentioned above, multiple instances of the same static instruction may co-exist within the instruction window. As it follows, the last value that is recorded in the predictor may not be apparently the most recent, as another one may have been computed but remains speculative because the relevant instance of the instruction is still in-flight. Therefore, to efficiently enable computation-based prediction, the hardware needs to provide a mechanism to supply the predictor with the most recent value which is not yet committed. This mechanism is commonly known as the *speculative window*. It practically is a fully-associative structure on-par with the *Instruction Queue* that tracks the values of all dynamic instructions that have not yet retired. However, as detailed in [PS15a], simply adding an additional associative structure of that scale in the pipeline may eventually render the implementation of the relevant microarchitecture complexity-ineffective.

2.2.4.3 Hybrid Predictors

As it appears from the above analysis, context-based and computation-based predictors can be considered partially complementary to each other. Sazeides and Smith had supported this argument in their study and proposed the development of *hybrid predictors* [SS97]. In general, the critical aspect of a hybrid scheme revolves around the policy that is used to enable the effective synergy of its counterparts. Early hybrid schemes of value prediction were mostly simple combinations that were based on accordingly trivial arbitration policies. For instance, in the *2-level + Stride* predictor of Wang and Franklin [WF97], both parts are equally accessed and updated. When predicting, priority is given to the 2-level predictor, meaning that Stride predicts only when the former has not a high-confidence prediction. Rychlik et al. examined a more efficient policy by proposing a dynamic classification mechanism that distributes instructions only to one predictor component [Ryc+98].

Nonetheless, the strict use of an arbitration policy may not be necessary if hybridization is "naturally" integrated in the predictor. The D-FCM predictor of Goeman et al. is a good example of this case [GVD01]. Unlike the simple FCM that we described above, D-FCM augments its structure with a table to hold the last values and adapts into recording strides rather than values to its basic components. In this way, any predicted value, that either follows a stride pattern or not, is practically expressed and therefore computed by using a stride (recall that even constant values can be computed using the zero-stride). Similarly, the more recent *D-VTAGE* predictor [PS15a], transforms the VTAGE predictor [PS14b] into a hybrid scheme by adding a last-value table in its structure and storing strides in its main components.

This refined style of hybridization potentially results in more space-efficient predictors. Indeed, consider that both in D-FCM and D-VTAGE, the components that were previously storing full values in their entries now need to record fairly shorter strides. Yet, complexity-wise speaking, hybrid models need also to address the issue that arises by the maintenance of the *speculative window* in computation-based value prediction. In the study introducing the D-VTAGE predictor [PS15a], the authors propose to limit this complexity by performing VP per fetch block rather than per one single instruction. As such, each entry in the predictor will now compact more than one stride together, for the prediction generation of also more than one instruction. Such a reform may not be straightforward, but it allows to highly reduce the design complexity of the needed speculative window by practically reducing the required entries and ports in its construction. Lately, Seznec proposed the EVES predictor [Seza] that requires to track only the number of the inflight instances of instructions, rather than their speculative values. Such information could be extracted for instance from regular structures of the out-of-order engine (e.g. the ROB). Therefore, in any case, a relatively complex associative search in the instruction window is required.

2.3 Address Prediction

2.3.1 Overview of Possible Use Cases

Value prediction mechanisms focus on predicting instruction results by detecting their repeatable behavior. Similarly to the results of instructions, value patters may also exist in the effective addresses of memory instructions (i.e. load and store instructions), making them accordingly *predictable*. As such, in the same period that value prediction was initially proposed [Gab96; LWS96; LS96], Sazeides et al. introduced the notion of *address prediction for the speculative execution of load instructions* [SVS96]. In their study, a stride-based mechanism is used to predict the effective address of load instructions to allow them to issue, and accordingly execute, speculatively. To our knowledge, this is the first scheme to repurpose a value-prediction algorithm for address prediction in order to enable speculative execution. Although, *address prediction* had already been used by some previous studies [EV93; AS95], but not in the same scope. For instance, Eickemeyer and Vassiliadis had proposed an address-prediction mechanism in order to enable data prefetching [EV93].

Following [SVS96], González and González examined the *predictability of load/store effective address* and proposed to use a *memory address predictor* (stride-based) to enable both load and store instructions to execute speculatively [GG97b]. Interestingly, in the framework they used in their study, the fraction of load/store instructions that exhibit strided references (i.e. effective addresses) is almost double from the respective fraction of load/store instructions that feature strided values. Therefore, the critical inference was that memory addresses appear to be more predictable than memory values.

Framing the anticipated benefit from employing a value predictor for speculative execution is fairly straightforward: when an instruction's result is predicted, dependents can use it to start execution speculatively ahead of time. In this way, sequential performance can be improved due to the increase of the ILP. On the contrary, the utility and therefore the benefit of *address prediction* is subject to the scope wherein it is considered. Essentially, studies that make use of *address prediction* can be mainly classified based on their central context into the following three types:

— Dynamic memory disambiguation [GG97a; Sat98]

— Data prefetching [GG97b]

— Speculative execution of load instructions [SVS96; MLO98; Bla+98; Bek+99; SCD17]

The general objective of this thesis work is to improve sequential performance by harvesting more ILP. In particular to address prediction, focus is given on the speculative execution of load instructions. Therefore, in the following paragraphs we will describe the main aspects of *load-address prediction*.

2.3.2 Data Speculation through Load-Address Prediction

2.3.2.1 Prediction Schemes

Load-address predictors can also be either *computation-based* [SVS96] or *context-based* [Bek+99]. Similarly to value predictors, load-address predictors aim to respectively capture patterns in the load addresses (i.e. the memory addresses accessed by load instructions). Practically, these patterns are similar to those we described above for value prediction. Therefore, the commonly followed approach is to leverage the already devised value prediction mechanisms in order to perform *load-address prediction* (LAP). This includes also other specific features of value predictors, such as the use of prediction confidence. In this respect, a scheme of *last-address predictor* was refined in [MLO98], a *stride predictor* was considered in [GG97b], while a hybrid of the aforementioned schemes together with an additional FCM-like predictor [SS97; SS98] (see Section 2.2.4.1) was proposed in [Bla+98].

On the other hand, Bekerman et al. studied closer the correlation of load addresses and proposed a more advanced predictor, called *Correlated Address Predictor* (CAP) [Bek+99]. CAP is a context-based predictor that is composed by a table that keeps past recent addresses of static load instructions and a table that holds the potential prediction. The first table is accessed with the load PC and provides the *load-address context* used to access the prediction table, in order to eventually retrieve the predicted load-address. In this way, CAP can capture more complex patterns, such as *repeated non-strides*. Still, its logic remains arguably very similar to the family of FCM value predictors [SS97; SS98], adapted for load-address prediction. More recently, Sheikh et al. refined this scheme proposing the PAP load-address predictor [SCD17]. PAP does not use per static instruction history like CAP, but a global program context built from the least significant bits of loads PC. Thus, its update and recovery is simplified, compared to CAP. Yet, this load-address context needs to be explicitly tracked to be reverted on a misprediction.

In reality, load-address predictors have been developing on-par with value predictors, from fundamental schemes that can only detect trivial patterns, to more elaborate that can accordingly capture more complex patterns. As in value prediction, the constant challenge that remains the same is the coverage of more and more cases. However, while the way to exploit a value predictor in a processor's pipeline in order to enable data-speculative execution is relatively simple (or even trivial), the same is not equally valid for a load-address predictor. The main reason for that relies on the fact that a load-address predictor does not directly provide the predicted data like a value predictor. To enable these speculative data paths, there exist certain design aspects that can be followed. These aspects are mainly related with the basic execution flow when a load-address predictor is inserted into a processor's pipeline, i.e. how a load that predicted its address is treated, how this load is executed or how dependents on this

load benefit from its predicted address. We discuss these aspects below.

2.3.2.2 Critical Design Aspects

Load-Address Prediction for data speculation is also some form of value prediction; rather than predicting the result of a load instruction, one predicts its effective address, allowing the relevant memory access to begin earlier. Following that, potential dependents on the corresponding load will accordingly retrieve their sources earlier and issue speculatively. In this way, similarly to value prediction, more instructions may execute concurrently (i.e. larger ILP) leading to higher instruction throughput. Unlike value prediction, though, speculatively-loaded (i.e. retrieved using a predicted memory address) values are not "atomically" available in the pipeline due to the interleaving memory-access delay. In that sense, what seems to be more challenging in load-address prediction is its effective exploitation during pipeline execution. Some of the initial studies [MLO98; Bek+99] were only focused in providing rigorous load-address predictors, mainly considering address-prediction accuracy, by assuming an abstract implementation. Fortunately, some other studies [SVS96; Bla+98; SCD17; Mor02] have provided more insights on vital implementation aspects that we describe below.

Treatment of Address-Predicted Load Instructions

In a conventional design, a load-address predictor is inserted into the frontend of the pipeline and is indexed in-order by any load instruction. Therefore, it is typically assumed that a load instruction can retrieve its prediction during the first stages of the pipeline. Following that, there are two possible directions that an *address-predicted* load can follow:

— Issue to the out-of-order execution engine using the predicted address.

— Initiate an address-speculative memory access early in the pipeline.

In the first case, the address-predicted load follows normally the pipeline flow in order to reach the out-of-order execution engine. Then, since it has predicted its memory address, it can issue its memory access in parallel to the address generation [SVS96]. Therefore, such design can only decouple load instructions from scheduling delays related with the computation of the memory address. In the second case, the load instruction starts its memory access as soon as it gets its predicted address, without needing to reach the out-of-order core [Bla+98; Mor02; SCD17]. Arguably, in modern general-processors that employ deep pipelines (see for instance Intel's recent core code-named *Sunny Cove* that implements a 19-stage pipeline), such design can entail more speedup.

In both of these designs, the memory access of an address-predicted load can issue speculatively with respect to the memory address, but possible memory dependencies need to be

always respected. In the first case, since the load is already in the out-of-order engine, the employed memory dependence predictor (see Section 1.4.3.2) can be seamlessly used to figure out underlying memory conflicts. An arbitration mechanism is also required for the second case that address-predicted loads launch their memory access before arriving in the core. A modern representative of this scheme is the study of Sheikh et. al [SCD17], where the PC of an address-predicted load that exhibit memory conflicts is stored in a filter to prevent its future instances from violating the memory order. Nonetheless, in Chapter 5 we will show that an auxiliary predictor is highly required in order to efficiently limit the amount of memory-order violations that can be induced by address-predicted loads.

Validation Techniques

In any of the above schemes, an address-predicted load may eventually retrieve its data earlier. When these speculatively-loaded data reach the pipeline, the execution becomes accordingly data-speculative, as any instruction that follows may use them in its operation [2]. Therefore, the correctness of the speculative execution needs to be validated. Apparently, such speculative execution includes two speculative aspects: the predicted load-address and the speculatively-loaded data. As such, two different validation policies have been followed: validation of the predicted address or validation of the loaded-data [SCD17]. Note that in case of a misprediction, execution recovery can be performed by employing pipeline squashing, similarly to branch/value prediction.

The first approach appears to be relatively natural, since the speculative execution originates by the prediction of the load-address. Commonly, in studies that this scheme is employed [SVS96; Bla+98; Mor02], loads validate their predicted address against the computed one. To do so, they perform their memory access speculatively, but they compute the actual memory address in order to validate the predicted one. As the memory access happens on the predicted address, in case of a misprediction (i.e. predicted address is different from the actual address) the loaded data are not correct and the address-predicted load should also be re-executed. On the other hand, the second approach induces two memory accesses per address-predicted load: firstly using the predicted address and then using the computed address. As such, the speculatively-loaded value is validated against the value that has been retrieved using the actual address. A representative example of this model can be found in [SCD17]. Arguably, such validation model can be employed if one needs to easily adapt a core featuring LAP in a multicore system.

2. Speculatively-loaded data may be propagated through the PRF (as typically happens in value prediction) or by using some dedicated structure, like in [SCD17].

CHAPTER 3

ON THE INTERACTIONS BETWEEN VALUE PREDICTION AND ISA-LEVEL INTRINSICS

3.1 Introduction

As mentioned in the previous Chapter, value prediction has been actively re-gaining interest in the research community, after some period of recess since its initial appearance in the 90's. The reason why processor designers and researchers had shifted away from VP was the ambient skepticism regarding the feasibility of the actual construction of a core that features a value predictor. That is, inserting a value predictor in a typical processor was considered complexity-ineffective. However, recent studies [PS14b; PS14a; PS15a; SCD17] have established practical solutions for the hardware implementation of value predictors that provide meaningful performance gains. Currently, a championship on value prediction (CVP [1]) is held in an all-year-round basis in order to coordinate the plethora of new proposals and maintain the research momentum. Till today, however, VP has not officially been implemented in any commercially available processor.

In general, studies on VP are ISA-agnostic. That is, value predictors are developed in order to be functional with any ISA that includes classes of instructions with explicitly mentioned operands that are either architectural registers or memory locations. In this respect, the effectiveness of already proposed VP schemes has been evaluated over various familiar ISAs, e.g. Power, Alpha, x86 and ARM ISA. Nonetheless, on a realistic implementation, the intrinsics of each different ISA can impose different limitations and offer specific freedoms on the operation of the value predictor, in the sense of affecting the range of value-predictable instructions. Also, as code orchestration depends on the ISA, instruction data-dependencies that can be collapsed through VP may have a different contribution to the awaited performance gain. All things considered, the impact of VP is arguably affected by the recruited ISA and as such, it may be more/less meaningful to implement VP in some ISAs.

1. https://www.microarch.org/cvp1/

To illustrate this aspect, in this Chapter, we characterize some chief figures of merit of VP on an ISA-aware basis in the context of the state-of-the-art VTAGE [PS14b]. Our ultimate objective is to reveal the ISA-effect on VP performance. To accomplish that, without loss of generality on VP portability, in our exploration we leverage two of the most dominant ISAs in leading-edge high-performance microprocessors, the *x86_64* and *Aarch64*. Besides, the exhaustive evaluation of VP performance over any existing ISA, except from being out of reach, is not our intention. To that extent, we first provide an outline of the basic aspects of the two ISAs. In particular, we focus on register-producing instructions that are meaningful to VP. In this way, we uncover the inherent ISA-related limitations and explicitly define the classes of value-predictable instructions per ISA. As a second step, we detail our customized experimental infrastructure that enables our exploration. Our experimental approach allows us to capture a wide variety of ISA-level particularities that contemporary CPUs may have and to examine their interaction with VP. We present our experimental results and delve into possible reasons for the variations that we observe in the VP-performance per ISA. Additionally, we briefly illustrate how vector optimizations, that are currently widely used, significantly limit the potential of VP.

3.2 Basic ISA Properties Meaningful to Value Prediction

This section briefly overviews the examined instruction sets by focusing on the register-producing and therefore, value-predictable instructions. In order to avoid a repetitive reference to the same sources of information, we mention here that the abridged overviews of the two sets are faithfully extracted from their relevant manuals [6M19; Man19; Arm19].

3.2.1 x86_64

x86_64 is the 64-bit version of the x86 instruction set originally developed by Intel. Historically speaking, it is the second extension of the x86 instruction set to a larger word size with emphasis on backward compatibility. Essentially, the x86 family of CPUs started in the 1970s with the 16-bit Intel 8086 processor [MPR78; Mor+80]. The 32-bit version of the ISA (IA-32) was released by Intel in 1985 [Bre95]. Then, this 32-bit architecture was extended to 64 bits by AMD in the early 2000s, initially with the name *x86_64* and as *AMD64* upon its release. As its initial specification was created by AMD, it is the first significant addition to the x86 architecture designed by a company other than Intel. However, the architectural extensions that x86_64 brought were soon adopted by Intel under the name *Intel 64*. Since *AMD64* and *Intel 64* are substantially similar, in this text, we will keep using the vendor-neutral term x86_64 in order to refer to the 64-bit mode of the x86 architecture.

3.2.1.1 Primary Data Types of Operations

In the primitive 16-bit version of the x86 architecture a *word* was defined as 16 bits (two bytes). Although, the *word size* in a CPU is typically based on the size of the memory data bus (i.e., the number of bits that can be loaded from memory or stored into memory within a cycle), for compatibility reasons the same word definition has been retained through any generation of the x86 architecture, including the x86_64 variant. According to that, a *dword* (double-word) denotes 32 bits (two words, or 4 bytes), a *qword* (quad-word) 64 bits (four words, or 8 bytes) and a double quad-word 128 bits (eight words, or 16 bytes). Also half-precision, single-precision, double-precision and double-extended precision floating points are defined as 16, 32, 64 and 80 bits respectively (IEEE 754 Standard).

3.2.1.2 Basic Program Execution Registers

Architectural registers can be roughly classified in three wide classes: *general-purpose registers* (GPRs), *floating-point* (FP) and *single-instruction-multiple-data* (SIMD) registers. The primitive *Floating Point Unit* (FPU) of x86-based architectures had been the *x87* superset, but it has been highly to totally superseded with the introduction of the *Streaming SIMD Extensions* (SSE). However, x87 FP instructions are still supported, as each architectural generation of the ISA extends the existing architectural structures without eliminating them. The architectural registers that are manipulated by data-processing and load instructions can be organized in the following categories:

1. **GPRs:** There are 16 GPRs of 64 bits that are used for integer arithmetic and logic, and to hold both data and pointers to memory. In practice, R0 to R7 are the extended version of the already existing 32-bits wide registers from previous generations and therefore they possess different aliases in order to be adequately manipulated from legacy instructions. GPRs support operations on byte, word, double-word and quad-word integers. The lower 32, 16 and 8 bits of each register are accessed using the names R0D to R15D, R0W to R15W and R0L to R15L respectively.
2. **x87-FPU Registers:** There are eight x87-FPU data registers of 80 bits for operations on single-precision, double-precision and double extended-precision floating-point values. The lower 64 bits are used to hold the *significand*, the following 15 bits for the *exponent* and the remaining highest bit for the *sign*.
3. **MMX SIMD Registers:** The eight MMX vector registers of 64 bits (MM0 through MM7) support execution of single-instruction-multiple-data (SIMD) operations on 64-bit packed byte, word, and double-word integers.
4. **SSE/AVX SIMD Registers:** This class includes 32 vector registers that were introduced gradually, after a series of SIMD ISA-extensions and support a diverse range of SIMD

operations of both integer and floating-point values. Their length is pre-fixed to 128 (SSE) or 256(AVX) or 512 ($AVX-512$) bits, depending on the SIMD-extension generation. The registers retain adequate aliases in order to remain forward/retro-compatible with the undergone SIMD extensions. These are ZMM0 through ZMM31, YMM0 through YMM31 and XMM0 through XMM31 for the width of 512, 256 and 128 bits, respectively.

5. **RFLAGS Register:** The 64-bit RFLAGS register contains a group of status flags, a control flag and a group of system flags, all of them mapped to the lower 32 bits (the higher 32 bits are reserved). From them, only the status flags and the control flag can be modified during a program's execution by the instruction stream. The status flags indicate the results of arithmetic instructions, such as ADD, SUB and MUL. The 6 different status flags are: the *carry flag* (**CF**, bit 0), *parity flag* (**PF**, bit 2), *auxiliary carry flag* (**AF**, bit 4), *zero flag* (**ZF**, bit 6), *sign flag* (**SF**, bit 7), *overflow flag* (**0F**, bit 11). Conditional instructions use one or more of the status flags as condition codes to specify their execution (e.g. jump instructions on condition code or conditional moves test the status flags for branch and data transfer respectively). The control flag, called *direction flag* (**DF**, bit 10), solely concerns instructions manipulating strings, defining the way that strings are processed, i.e. from high addresses to low addresses or the opposite.

3.2.1.3 Instruction Classes

Similarly to its predecessors, x86_64 is a highly micro-coded ISA, which means that a single instruction (*macro-op*) may consist of several low-level operations (*micro-ops*). Generally, x86-based processors are typically characterized as quasi-CISC. Instruction operands can be *immediates* (direct values), *registers* or direct *memory locations*. However, a single instruction may only have one memory operand, as memory-to-memory operations are not feasible. Instructions encoding allows them to have at most two and in rare cases three operands. Therefore, arithmetic or logical instructions with two operands are characterized as "destructive", in the sense that one of their source operands is replaced by the execution outcome, i.e. acting also as the destination operand. For instance, an ADD instruction would follow the form *a = a + b*. An exception to that are the SSE/AVX SIMD instructions (described below) that are expressed in an at least three-operand format, having always separate destination and source registers. According to the architectural registers they process, all data-processing and load instructions of the x86_64 set can be generally distributed in the following classes:

1. **General-Purpose:** This class includes any instruction that performs basic data movement, memory addressing, arithmetic and logical operations, program flow control, input/output, and string operations on integer data types of at most quad-word size (64 bits) using any of the GPRs. Quad-word results take the whole destination register, while

double-word results get zero/sign-extended to fill all the 64 bits. Byte and word outputs are only sign-extended when the corresponding operation concerns address calculation. Otherwise, the upper bound of the equivalent destination register is not modified by the operation.

2. **Primitive x87-FPU:** The x87-FPU instructions operate on floating-point operands by treating the eight x87-FPU data registers as a register stack where all addressing is relative to the register on the top. The double-extended-precision floating-point format is only supported by this class, while operations on single and double-precision floating point data happen also by SSE FP instructions that we discuss later.
3. **MMX (SIMD INT):** MMX is a set of highly optimized SIMD instructions that perform parallel computations on packed integer data contained in the 64-bit MMX registers. Instructions in this category always produce a 64-bit result that is stored in an MMX register. The packed addition of 16-bit integers is described with the following pseudocode:

Packed SIMD addition of word INT

```
1. FOR j = 0 to 3
2.      i = j*16
3.      dest[i+15:i] = a[i+15:i] + b[i+15:i]
4. ENDFOR
```

4. **SSE/AVX (SIMD INT/FP):** SIMD instructions of this category originate from a series of *Streaming SIMD Extensions* (SSE, SSE2, SSE3, SSE4) and *Advanced Vector Extensions* (AVX, AVX2 and AVX512). Unlike MMX SIMD-instructions, they operate on both integer and floating-point (single and double-precision) values and thus, they cover most of the floating-point computations. They are distinguished in two principal categories: those that operate on packed SIMD data (either integer or floating point) and those that operate on explicitly scalar floating-point data. In the second one, only the lowest XMM (128 bits) element is engaged, regardless of the supported vector length. Hence, they produce 128-bit results that are zero-extended if necessary. For instance, the operation of a scalar single-precision floating-point addition is expressed by the following algorithm (*MAX_VL* denotes the supported *maximum vector length* in any subsequent example):

Scalar SIMD addition of single-precision FP

```
1. dest[31:0] = a[31:0] + b[31:0]
2. dest[127:32] = a[127:32]
3. dest[MAX_VL:128] = 0
```

Similar is also the operation of a scalar double-precision floating-point addition:

Scalar SIMD addition of double-precision FP

```
1. dest[63:0] = a[63:0] + b[63:0]
2. dest[127:64] = a[127:64]
3. dest[MAX_VL:128] = 0
```

On the other hand, packed SIMD instructions operate on packed integer or floating-point data and can produce at least XMM-wide (128 bits) and at most ZMM-wide (512 bits) results, depending on the supported vector length. Below we selectively present the algorithm of the operation of a packed SIMD addition of double-word integers and the operation of a packed SIMD addition of single-precision floating-point data:

Packed SIMD addition of double-word INT

```
1. FOR j = 0 to MAX_VL/32
2.      i = j*32
3.      dest[i+31:i] = a[i+31:i] + b[i+31:i]
4. ENDFOR
```

Packed SIMD addition of single-precision FP

```
1. FOR j = 0 to MAX_VL/32
2.      i = j*32
3.      dest[i+31:i] = a[i+31:i] + b[i+31:i]
4. ENDFOR
```

Beyond the above categories, regular instructions may also modify some of the status flags in the RFLAGS register, regardless if they already produce a result that is written in one of the available registers we described (e.g. ADD, MUL or DIV instructions). Moreover, some other instructions exist that do not generate a value-result but only modify the RFLAGS register, e.g CMP, STD and CLD, from which the last two set and clear the DF flag, respectively.

3.2.2 Aarch64

ARMv8 is the latest generation of the ARM instruction set architecture and the first one to introduce the 64-bit execution state Aarch64. Aarch64 encapsulates all the 64-bit operating capabilities of an ARM-based processor, representing the 64-bit version of the instruction set. As

the ARM ISA is related with a series of CPUs since the early 80's, ARMv8 features also a 32-bit execution state, the Aarch32, in order to maintain backward compatibility. ARM-based architectures are representative examples of RISC machines. Yet, instructions are often composed by smaller *micro-ops*, since even RISC-like general-purpose processors can have instructions that perform complicated operations.

3.2.2.1 Supported Data Types

The Aarch64 ISA has a fixed word size of 32 bits, like any other version of the ARM architecture. Based on that, operations on integer data can happen in the byte (8 bits), word (32 bits), half-word (16 bits), double-word (64 bits) and quad-word (128 bits) level. On the other hand, floating-point values can be expressed and processed as half-precision (16 bits), single-precision (32 bits) and double-precision (64 bits) data (IEEE 754 Standard).

3.2.2.2 Register & Instruction Classes

The architectural registers that are available to any data-processing and load instruction are generally divided into the class of *general purpose (integer) registers* (GPRs) and *single-instruction-multiple-data & floating-point* (SIMD&FP) registers, coupled with the *condition-flags* register:

1. **GPRs:** The general-purpose register bank contains 32 64-bit architectural registers (R0 to R31 [2]) that enable any arithmetic or logical operation on integer values.
2. **SIMD&FP Registers:** The SIMD&FP register bank contains 32 128-bit [3] vector registers (V0 to V31) for SIMD vector and scalar floating-point operations. Essentially, the integer quad-word and all the floating-point data types apply only to the SIMD&FP registers. A vector register can hold one or more packed elements of the same size and type or a single/scalar element starting from the least significant bits.
3. **NZCV Register:** This special-purpose register is employed to hold the four condition flags: *negative* (N), *zero* (Z), *carry* (C) and *overflow* (V). Conditional instructions test these flags in order to define their execution based on the corresponding condition code. On the other hand, the condition flags are updated either by flag-setting data-processing instructions, such as ADDS, ANDS, SUBS etc., or by conditional compare instructions, like CCMN and CCMP. In this way, execution of the instruction is conditional on the result of a previous operation.

2. In the 64-bit context of Aarch64, R31 represents *"read zero"* or *"discard result"* (aka the *"zero register"*) under the name *XZR* and *WZR* for 32-bit and 64-bit operand size, respectively.

3. Latest *Scalable Vector Extensions* (SVE) on Aarch64 ISA allow variable length of the vector registers, ranging from 128 to 2048 bits. In this general description, we consider the basic case of 128 bits.

Accordingly, data-processing and load instructions can be classified into the following categories:

1. **General-purpose (INT):** Integer instructions perform their operations by using any of the GPRs. The operand's data-size and hence the instruction's data-size (either word or double-word) is explicitly defined by the instruction mnemonics. The letter W is shorthand for a 32-bit word, and the X for a 64-bit extended word [4]. Instruction results of word size are zero/sign-extended to 64 bits. For any register number "n" between 0 and 30, the qualified names have the following form:

Size (bits)	word (32 bits)	double-word (64 bits)
Name	Wn	Xn

where Wn and Xn refer to the same architectural register

2. **Advanced SIMD Scalar (INT/FP):** Scalar SIMD integer or floating-point instructions process single-element vector registers similarly to the main general-purpose integer registers, i.e only the lower and relevant bits are accessed, with the unused high ones ignored on a read and set to zero/sign on a write. Consequently, they produce integer or floating-point values of up to 128 bits (quad-word). For instance, the scalar SIMD addition of two double-word integer values is described below:

Scalar 64-bit SIMD addition of double-word INT

1. dest[63:0] = a[63:0] + b[63:0]
2. dest[127:64] = 0

3. **Advanced SIMD Vector (INT/FP):** Vector SIMD integer or floating-point instructions use the multi-element vector registers in a parallel fashion. While 128-bit vectors are supported, the effective vector length can be also 64 bits. The element size is derived from the instruction mnemonics [5] and the number of elements or "lanes" is computed based on the vector length. Thus, the vector shape can have one of the following formats, expressed in bits per lane: *8b×8, 8b×16, 16b×4, 16b×8, 32b×2, 32b×4, 64b×1, 64b×2*. Similarly to the scalar SIMD instructions, when the vector length does not equal 128, the upper 64 bits of the register are ignored on a read and set to zero/sign on a write. For instance, a vector integer addition is processed in the following way:

4. The letter X (extended) is used rather than D (double), since D conflicts with its use for floating point and SIMD "double-precision" registers.

5. B, H, S, D for 8, 16, 32 and 64 bits respectively.

Scalar 64-bit SIMD addition of double-word INT

```
1. dataSize = if Q == '1' then 128 else 64
2. elements = dataSize DIV elementSize
3. FOR e = 0 to elements-1
4.       operand1 = Vn[e+elementSize:e]
5.       operand2 = Vm[e+elementSize:e]
6.       Vdest[e+elementSize:e] = operand1 + operand2
5. ENDFOR
6. Vdest[127:dataSize] = 0
```

Of particular interest are *load-pair* (LDP) instructions which load a pair of independent registers with values from consecutive memory locations. These instructions can manipulate registers from both the GPR and the SIMD&FP bank and therefore are not explicitly classified to one of the above categories. Their loaded-values can be at most equal to a quad-word and thus zero/sign extension is applied when necessary. The operation of a load-pair that loads two GPRs with two double-words from memory can be described from the following algorithm:

LDP X1, X2, addr

```
1. X1[63:0] = mem[addr:addr+64]
2. X2[63:0] = mem[addr+64:addr+128]
```

3.3 Value Prediction in the Studied ISAs

This section demystifies practical issues on the implementation of value prediction over the two architectures we described above. As already mentioned, x86_64 is a highly micro-coded ISA while in Aarch64 the use of micro-ops is more limited. However, an effective implementation of value prediction in a processor built with any of these sets would most likely need to target instructions in the micro-op level, as this is where all *RAW* dependencies reside. As a result, from now on, the term instruction will refer to the low-level operations of a potentially multi-level instruction. Note that the instruction classification described in the previous section is naturally applied in micro-ops, since it is guided by the type of register(s) that instructions use for their output(s) (INT,FP and SIMD).

3.3.1 ISA-aware Definition of Value-Predictable Instructions

Value predictors are individual processor components that monitor execution by hosting the results of instructions in dedicated array structures. More specifically, the entries of a predictor's table(s) hold information necessary to enable its prediction logic (such as tags for indexing, and confidence counters of predictions) and store encountered values (instructions results) in the payload. According to that, a predictor's width equals the number of bits that are mapped to the payload of its entries. In latest studies [PS14b; PS15a; SCD17; SH19], the newly introduced VP models had been evaluated over one of the two 64-bit ISAs that we consider in our exploration by employing 64-bit wide predictors.

However, as detailed in Section 3.2, in both ISAs, instructions do not flatly produce 64-bit values, as SIMD results are in the range of 128 to 512 bits. Predicting such wide values of n bits using predictors with the same width is impractical, as it requires the selective allocation of $\frac{n}{64}$ instead of one single entry. As a quick refresher, modern context-based value predictors are developed over sophisticated indexing schemes, attempting to map the entries of fixed-size prediction tables to instructions, with as less aliasing as possible. The most fashionable way to accomplish that is to probe those prediction table(s) by using a mixture of the instruction's PC address with a distinctive context (e.g. branch outcome history [PS14b]). Therefore, allowing the non-uniform allocation of more entries for some of the instructions disorganizes significantly the underlying mapping and can impair its effectiveness. On the other hand, by uniformly provisioning n (i.e 128 to 512) bits per entry, scales un-affordably the required storage budget of the predictor. The bottom line is: being in line with the current processor generation, modern value predictors are 64-bit wide and instructions are predictable only when their result can fit in 64 bits.

As it follows, a modern 64-bit wide value predictor has to be explicitly tuned in an ISA-aware manner in order to consider only the appropriate instructions. By definition, value-predictable instructions regroup data-processing and load instructions that produce a value that is written to the register file. The INT and FP alias distinguishes instructions that produce integer and floating-point values, respectively. This characterization does not necessarily reflect the ISA definition of INT or FP instructions, since there are instructions that are pre-defined as FP and eventually produce an integer value, e.g. data type conversions and move instructions. Based on the classification of Section 3.2, for any CPU-design that implements one of the studied ISAs, the eligible instructions for value prediction are:

— **In x86_64:**

1. General-Purpose INT
2. MMX SIMD INT
3. Instructions solely writing to the RFLAGS register

— **In Aarch64:**

1. General-Purpose INT
2. Scalar/Vector SIMD INT/FP of up to 64 bits
3. Instructions solely writing to the NZCV register

Overall, in both sets, as GPRs are natively 64 bits long, instructions writing to any GPR are suitable for value prediction. So are instructions that only modify the corresponding *flags-register*, since the last never exceeds the size of 64 bits. For any of the two instruction sets, an SIMD instruction can be value-predictable when either it naturally writes to a 64-bit register (MMX SIMD of x86_64) or its result is up to 64 bits and then zero/sign-extended to a wider format (part of the vector/scalar SIMD instructions of Aarch64). As a consequence, packed SIMD instructions of x86_64 are non value-predictable since their outputs are at least 128 bits. Scalar SIMD instructions of x86_64 are not value-predictable either, as in their implementation (see Section 3.2.1.3, even though they generate at most 64-bit results, the final value that the destination register receives is firstly extended with the upper part of the source register and then, if necessary, zero/sign-extended. Note that for both ISAs, solely the above categories of instructions access the corresponding value prediction tables. Non value-predictable instructions such as branches are executed as usual.

3.3.2 Deriving Flags from Predicted Results

Instructions that generate a result but also modify the flags-register, have by default a twofold contribution to *RAW* dependencies of following instructions: one directly from their generated value and one from the flags that they modify. Only predicting their produced result is not enough to utterly eliminate the dependencies they trigger, since modifications on the flags-register will be still pending. To radically defeat both dependency sources, the related flags need to be derived from the predicted results, when feasible. In x86_64, the flags *ZF*, *SF* and *PF* can be easily extracted from the predicted result, as they only relate to it. The remaining ones cannot be equally guessed, since they depend on the instruction's operands. Accordingly, in Aarch64 only the flags *N* and *Z* can be directly inferred from the predicted result. Derivation of flags is assumed to be the last phase of value prediction [PS14a], as it needs to happen after the predicted value has been specified.

3.3.3 Arbitration on Instructions with Multiple Destination Registers

Although not explicitly mentioned before, both sets that are studied in this work are equipped with instructions that can have more than one destination registers. In x86_64, these cases are limited only to instructions like MUL and DIV, that capture potentially overflown values. On

the other hand, Aarch64 contains load-pair instructions, that load values into two separate registers. In both cases, these instructions initiate two different dependency chains of instructions, as their destination registers can be subsequently used in other operations of any kind (e.g. arithmetics or move operations). Ideally, by predicting both of them, these dependency chains would be collapsed and extract more performance. However, similarly to the case of SIMD instructions, predicting two different values selectively for some instructions would be impractical. Authors in [SCD17] mentioned that when they attempted to use similar heuristics to predict multiple values for load-pair instructions, they observed destructive aliasing in the prediction tables. Therefore, the general consensus has it that only one of the destination registers can be predicted and conventionally the first one [EPS17; SCD17]. As it follows, two subclasses of predicted instructions can be distinguished:

— **Fully Predicted**, that can possess by construction only one predicted destination register.

— **Partially Predicted**, that modify more than one registers, from which only the first one is conventionally predicted.

Note here that only *fully predicted* instructions can derive the flags they affect from their prediction, in the way we described in the previous Section.

3.4 Experimental Setup

3.4.1 Simulator

In our experiments, we employed the *gem5* cycle-level simulator [Bin+11] implementing both the x86_64 and Aarch64 ISAs. In the gem5-x86 version we use in our study, we have implemented branches in a single rather than three µ-ops, to eliminate false dependencies that would not exist in a realistic processor.

We model a relatively aggressive 4GHz, 8-issue superscalar baseline with a fetch-to-commit latency of 19 cycles. The in-order frontend, the out-of-order backend, as well as the in-order Commit stage of the pipeline, they are all properly dimensioned to be able to treat up to 8 µ-ops per cycle. We model a deep frontend that spans in 15 cycles, accompanied by a fast backend of 4 cycles, in order to obtain realistic branch/value misprediction penalties. The parameters of our baseline core are configured as close as possible to those of Intel's Skylake core, one of the latest commercially available Intel microarchitectures. Table 3.1 describes the details of the baseline pipeline structure we use. Since µ-ops are known at Fetch in *gem5*, all the widths given in Table 3.1 are in µ-ops, including the fetch stage. Independent memory instructions (as predicted by the Store Set predictor [CE98]) are allowed to issue out-of-order and the entries in the IQ are released upon issue.

Table 3.1 – Simulator configuration overview. *not pipelined.

Front-End	**Fetch through Rename width**: 8 insts/cycle **L1I**: 8-way, 32KB, 1 cycle, 128-entry ITLB, 64B fetch buffer **Branch Pred.**: State-of-the-art TAGE-SC-L [Sezb] 64KB, 2-way 8K-entry BTB, 32-entry RAS, 20 cycles min. mis penalty.
Execution	**Issue through Commit width**: 8 insts/cycle **ROB/IQ/LQ/SQ**: 224/97/72/56 (modeled after Intel Skylake) Store To Load Forwarding (STLF) latency: 4 cycles 256/256 INT/FP physical registers (4-bank PRF) 2K-SSID/1K-LFST Store Sets [CE98], not rolled-back on squash and cleared every 30K access 4ALU(1c), 1Mul/Div(3c/25c*), 2FP(3c), 2FPMulDiv(5c/10c*), 2Ld/Str Ports, 1Str Port, Full bypass
Memory Hierarchy	**L1D**: 4-way, 32KB, 4 cycles load-to-use, 64 MSHRs, 2 reads & 2 writes/cycle, 64-entry DTLB **L2**: Unified private, 8-way 256KB, 11 cycles, 64 MSHRs, no port constrains, Stream Prefetcher, (degree 1) **L3**: Unified shared, 16-way 2MB, 34 cycles, 64 MSHRs, no port constrains, Stream Prefetcher, (degree 1) All caches have 64B lines and LRU Replacement Policy **Memory**: Dual Channel DDR4-2400 (17-17-17) 2 ranks, 8banks/rank, 8K row-buffer, tREFI 7.8us, Across a 64B bus Min Read lat.: 75 cycles, Max.: 185 cycles.

Table 3.2 – Layout summary of the employed value predictor VTAGE, where $rank$ varies from 1 to 6, being the position of the tagged component from lowest to highest history length used.

Predictor	Number of Entries	Tag-Width (Bits)	Size (KB)
VTAGE [PS14b]	4096 (Base)	-	33,5
	6 x 512 (Tagged)	12 + $rank$	30,5

3.4.2 Value Prediction Setup

3.4.2.1 Predictor Considered in this Study

The inter-ISA exploration of value prediction of this study has been performed in the context of the value predictor VTAGE [PS14b], an arguably worthy representative of modern context-based value predictors, i.e. one of the major classes of value predictors, as detailed in Chapter 2. Nevertheless, other models could have been also considered, either some more fundamental like LVP [LS96], or some other modern equivalents like DLVP [SCD17].

The configuration of VTAGE that we use has been carefully chosen to operate well (no slowdowns) in the simulation of both ISAs but without necessarily reaching the upper limit of obtainable speedup. The VTAGE-structure we consider is composed of a 4K-entry last-value table (LVT or base component) and 6 512-entry partially tagged components. The internal characteristics of VTAGE are meticulously extracted from [PS14b], as overviewed in the previous

Chapter. Its storage budget reaches strictly the 64KB, corresponding to the storage budget used in branch predictors of high-end processors. Adopting such a reasonably sized design allows us to draw highly realistic conclusions from our simulations. Table 3.2 summarizes the specific sizes used in the internal modules of VTAGE. In its implementation, Forward Probabilistic Counters (FPCs) [RZ06] are employed for the confidence estimation of generated predictions. Particularly, the probability vector V = $\{1, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}\}$ is used to control the forward transitions of the FPCs. In essence, by discriminating predictions based on their confidence some prediction coverage is sacrificed in trade of admissible accuracy, i.e. embodiment of VP does not lead to execution slowdowns. With the use of FPCs, we keep value-prediction accuracy above 99,5%, guaranteeing deeply unbiased exposure of the underlying potential of value prediction.

3.4.2.2 Value Predictor Operation

For each simulated ISA, the predictor makes a prediction at *Fetch* time for any eligible µ-op from those detailed in Section 3.3.1. In particular for the x86_64 ISA, *gem5* does not support AVX instructions but it does implement the instructions of the SSE category. That is, the length of its implemented vector registers can be either 64 (MMX SIMD registers) or 128 bits (SSE SIMD registers). This is conveniently in agreement with the basic vector length (128 bits) of the Aarch64 ISA that is adopted in the implementation of *gem5*[6], allowing comparable inter-ISA simulations. Note, however, that in terms of simulation, *gem5-x86* splits any 128-bit SSE instruction writing to an *xmm* register into two 64-bit µ-ops. Although these "cracked" µ-ops are allegedly predictable, in order to apply the value-predictable conditions of Section 3.3.1, we explicitly tune the value predictor not to consider them, since instructions "cracking" is not representative of a high-performance implementation.

To index the predictor, we XOR the instruction's PC address left-shifted by one with the µ-op number inside the instruction. With this mechanism we ensure that (especially in x86_64) more than one µ-ops of the same instruction will generate a different index and therefore will have their own entry in the predictor. We assume that the predictor can deliver fetch-width predictions by the *Fetch* stage. Predictions with high confidence are written in the PRF at Rename time and replaced by their non-speculative counterparts, when they are regularly computed by the OoO execution engine. Like other recent studies [PS14b; PS14a], predictions are inserted in a FIFO-queue equal to the length of the instruction window, to allow in-order prediction validation and update of the predictor at commit time. To do so, a pre-commit stage responsible for validation/training (VT) is added to the pipeline. The VT stage lengthens the pipeline by one cycle, resulting in a value misprediction penalty of at least 21 cycles (based on the pipeline length

6. SVE feature, which is also implemented in gem5, is not considered in this study, for the sake of analogy with the implemented vector-related features for x86_64.

of our framework described above), while minimum branch misprediction latency remains unchanged. The validation is performed after retrieving the architectural value from the PRF and comparing it with the value from the head of the previously-mentioned queue.

3.4.2.3 Simulating Fully Derivation of Flags

As we stated in Section 3.3.2, in both ISAs that we consider in our study, the corresponding flags that a predicted instruction sets can only be partially derived by its predicted value (3 out of 7 in x86_64 and 2 out of 4 in Aarch64). In terms of simulation though, in order to wholly gauge the impact of inserting a value predictor in a processor, we apply a special workaround in our *gem5* implementation. Execution correctness is guaranteed as a prediction is considered incorrect if a single one of the derived flags (i.e. the corresponding flags-register) is eventually wrong. For x86_64, we adopted the approach found in [PS14a]. That is, the OF is always unset (0) and the CF mimics the SF. Additionally, the AF is ignored when validating the correctness of the flags-register, as its use is totally deprecated. Accordingly, in Aarch64, we approximate the remaining two flags similarly to the way found in [EPS17], i.e., both C and the V flags are always assumed to be 0, except from the case of a SUB instruction where C is set (1).

3.4.3 Benchmark Slicing for Inter-ISA Comparable Simulations

We consider workloads from both SPEC2K6 [Staa] and SPEC2K17 [Stab] suites. Simulating these applications entirely is inarguably out of reach. On the other hand, typical techniques like Simpoints [Per+03] that discover representative "*slices*" of a benchmark use a single binary as input. Therefore, these techniques are not applicable in our inter-ISA exploration where we need to isolate the same program-region in two different binaries, one compiled for x86_64 and one for Aarch64. In order to guarantee that the simulated slices of an application are identical in both cases, we employ a special method motivated by the study of Endo et al. [EPS17], that equivalently analyzed the impact of compiler optimizations on VP. Our method has as follows:

1. **Profile:** Use the *Linux Perf Tools* to profile the program's execution on a machine that natively implements one of the studied ISAs.
2. **Select:** Identify a function from the program that both represents an outstanding part of its execution time and is called relatively close to the beginning.
3. **Fast-Forward:** Using the performance counters of the relevant CPU, count the **F** function calls that are needed to fast-forward the program to 2% of its retired instructions.
4. **Map:** Using the same setup, specify the application slice that covers **W** and **R** calls to the selected function that roughly represent 50 and 500 million contiguous instructions, respectively.

Table 3.3 – Benchmark regions. Top: SPEC2K6, Bottom: SPEC2K17

Benchmark	Function	W Calls	R Calls
soplex	`CLUFactor::vSolveUright`	7	59
povray	`All_Plane_Intersections`	34878	337187
sjeng	`std_eval`	4595	44802
libquantum	`quantum_toffoli`	2	24
h264	`FastPelY_14`	398770	2 263 067
omnetpp	`cMessageHeap::shiftup`	9470	98471
mcf	`primal_bea_mpp`	19	187
lbm	`LBM_performStreamCollideTRT`	121935	329682
xalancbmk	`ValueStore::contains`	73	630
deepsjeng	`ProbeTT`	5924	61506
imagick	`SyncCacheViewAuthenticPixels`	1	2
leela	`FastBoard::update_board_fast`	6967	67447
nab	`heapsort_pairs`	182	1266
exchange2	`digits_2`	9574	76206
fotonik3d	`mat_updateE`	17	20

5. **Slice:** Carefully configure the application's code using special *gem5-commands* so that when it is simulated on *gem5*, the simulation statistics will be reset after **W** calls (warmup part) and the program will *exit* after **R** calls (effective simulation of the region of interest).

All the applications are compiled with gcc -O3 in both architectures. For each of them, we carried out the first four steps of our technique on an Intel Core i7 and then we applied the necessary changes to get the same slice in both binaries (step 5). For every application, we avoid potential redundant program-initializations by fast-forwarding simulation to **F** calls. As a result, we consider the same code-region represented by **W+R** calls of the selected function in application binaries of either ISA simulated on *gem5*. Note, however, that the resulting program slices may not represent the program execution in the same scale as the slices that a Simpoints analysis can provide [Per+03]. In other words, although these slices seamlessly enable our unique inter-ISA exploration of value prediction, they should not be considered for the absolute evaluation of a value prediction algorithm/model.

Table 3.3 presents the applications that we eventually use in our evaluation, together with their selected function and their slicing parameters (function calls). We chose a set of applications from both SPEC2K6 [Staa] and SPEC2K17 [Stab] suites, on which our slicing technique could be effectively applied. That is, we did not consider applications that the representative function was located un-affordably [7] away from the beginning of execution or applications that

7. Fast-forwarding till the first call of the selected function would need a prohibitive amount of simulation time, which was more than a month.

slicing using a function was not possible [8]. Nonetheless, our experimental analysis shows that the subset we used serves adequately the purposes of this study, as it allows to capture a very wide variety of the outstanding features that the two examined ISAs possess.

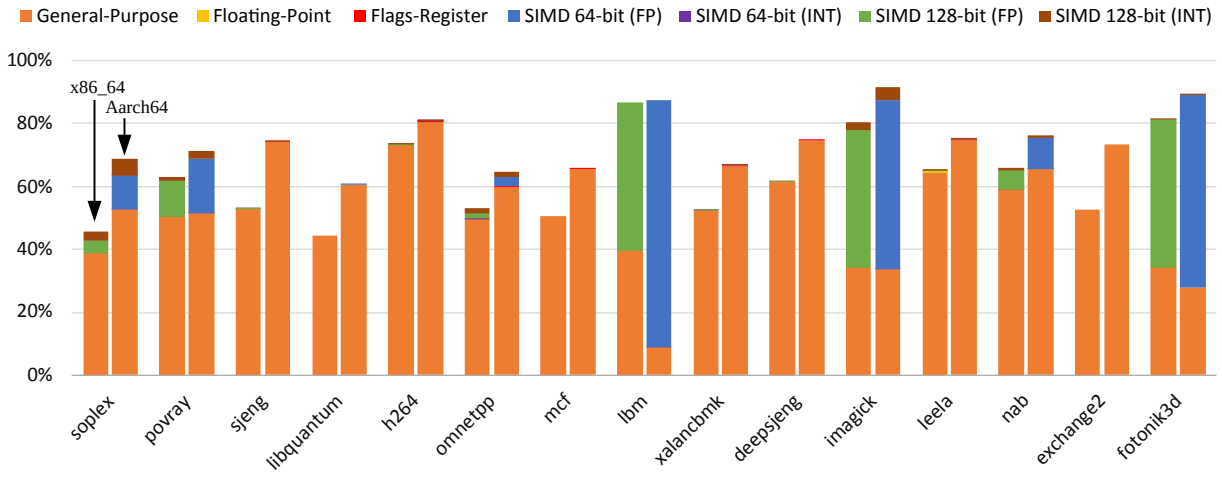
3.5 Simulation Results

3.5.1 ISA Effect on Value Predictability

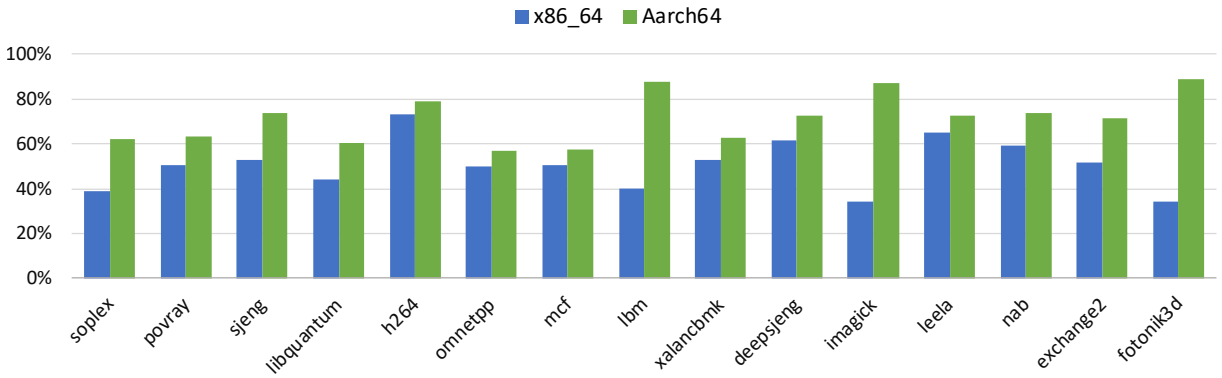
The potential of value prediction can be generally approximated by the proportion of value-predictable instructions that are found within the relative instruction stream that is executed. As we described in Section 3.3.1, value predictable instructions need to be specifically defined per ISA based on the width of their destination register(s) and the width of their effective result(s). Therefore, as a first step in our exploration, we have profiled the register-producing instructions in our workloads, categorizing them according to the type of their destination register. Figure 3.1a presents in a stacked diagram their distribution in the seven distinct categories that can be identified per ISA, based on the analysis of Section 3.2. The numbers presented in the graph are relevant to the total amount of committed instructions. For each benchmark, two bars are depicted, referring to x86_64 and Aarch64 (from left to right), respectively. Note that the *Floating-Point* category contains only instructions that write in registers that are defined as floating-point by the hardware, i.e. not instructions that use the SIMD registers to perform floating-point operations. That is, they can only exist in ISAs that feature a FPU, such as the x87-FPU of x86_64.

In both ISAs, the overwhelming majority of instructions are writing on a general-purpose (i.e. integer) register. Thereafter, in x86_64, benchmarks contain hardly any floating-point instruction, since floating-point operations are practically performed by 128-bit scalar SSE instructions. These are basically included in the class of 128-bit FP SIMD instructions. Integer SIMD 64-bit instructions are negligible, as 64-bit packed MMX instructions have been highly superseded by 128-bit packed integer SSE instructions. Also, instructions that solely write the flags-register are not that frequent. On the other hand, benchmarks in Aarch64 appear to contain both FP 64-bit and integer 128-bit SIMD instructions, but very few that modify only the flags-register. In Aarch64 floating-point operations are processed as scalar SIMD instructions that write at most the lower 64 bits of the vector register and always zero the rest upper bound. Therefore, these are counted in the class of 64-bit FP SIMD, although they modify a 128-bit vector register. As a result, in Aarch64 several are the benchmarks that feature large portions of 64-bit FP SIMD instructions. On the contrary, in x86_64, the same benchmarks expose higher rates of 128-bit

8. It was not possible to cut the application in the reasonably-sized slices we defined because the function was covering a very big portion of the execution time but it was called very few times.



(a) Register-based classification of register-producing instructions in relevance with total amount of committed instructions.



(b) Percentage of committed instructions suitable for value prediction.

Figure 3.1 – Value predictability.

FP SIMD instructions, since 128-bit FP SSE instructions modify entirely the vector register, as we explained in Section 3.2.1.3.

According to the ISA-aware definition of value-predictable instructions of Section 3.3.1, the first five categories of Figure 3.1a, excluding the floating-point class, regroup value-predictable instructions in both ISAs. This is expressed in Figure 3.1b that follows and eventually presents the portion of value predictable instructions out of the total amount of simulated instructions (register-producing or not). Overall, Aarch64 exhibits higher rates of value-predictable instructions. In many benchmarks, the two ISAs possess comparable rates, but in some of them e.g., *soplex, lbm, imagick* and *fotonik3d*, the supremacy of Aarch64 is fairly clear. By combining the two Figures, we observe that most of these benchmarks are those that feature large portions of 64-bit FP SIMD instructions in Aarch64 and an accordingly great amount of 128-bit FP SIMD

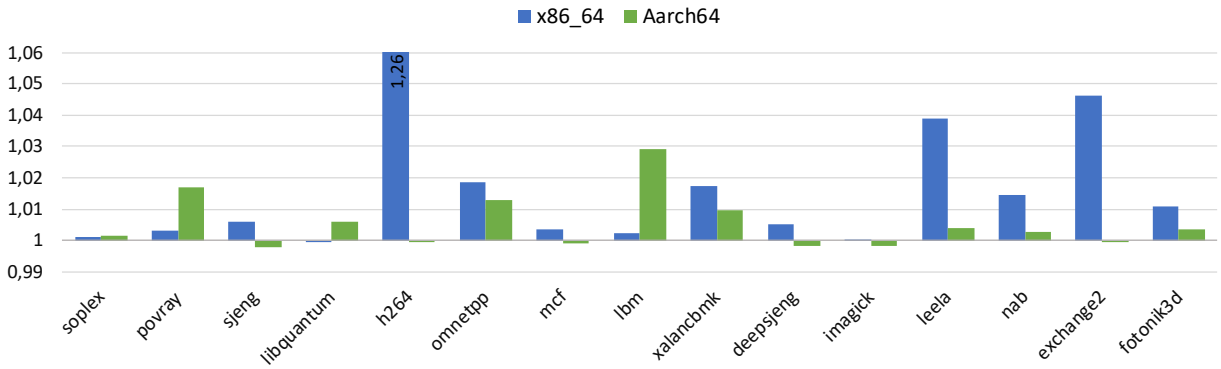


Figure 3.2 – Performance improvement brought by value prediction using VTAGE.

instructions in x86_64. But since 128-bit instructions do not fit in the conditions of value prediction, x86_64 appears to exhibit less predictable instructions. Consequently, we observe that the 128-bit processing style that x86_64 has adopted for floating-point instructions, can be a considerable impediment in the potential of value prediction. Nevertheless, value-predictability is an indication for potential performance improvements but not a guarantee. As such, the next paragraphs examine in detail the value-prediction impact over the two ISAs.

3.5.2 Analysis of Value-Prediction Impact per ISA

As we mentioned before, VTAGE [PS14b] is employed for the needs of this study due to being a well-established model of value prediction in the literature. However, our objective is to capture the general impact of value prediction on the examined ISAs rather than meticulously evaluate the efficiency of VTAGE. Therefore, the following analysis focuses on the inter-ISA comparison of typical VP-metrics and not on their absolute evaluation. Note again that the reported results are representative in terms of an inter-ISA comparison, but maybe not optimal. Nonetheless, a comprehensive evaluation and analysis of VTAGE is provided in Chapter 4.

Figure 3.2 illustrates the obtained speedup through value prediction with VTAGE in both ISAs, by reporting the relative IPC normalized to the baseline. As the results indicate, most of the simulated benchmarks benefit from value prediction more when implementing the x86_64 rather than the Aarch64 ISA. Only in two cases, namely *povray* and *lbm*, the performance gain in Aarch64 is higher. Of particular interest is the case of *h264*, where no gain appears when the Aarch64 ISA is used, but around 26% of speedup is observed in the x86_64 ISA.

To gain some insight on why we observe such behavior, initially, we consider the fraction of register-producing instructions of each ISA that are eventually predicted. In particular, we discriminate them as *fully* and *partially* predicted instructions, as defined in Section 3.3.3, and we depict them in Figure 3.3. The first observation is that the total number of predicted in-

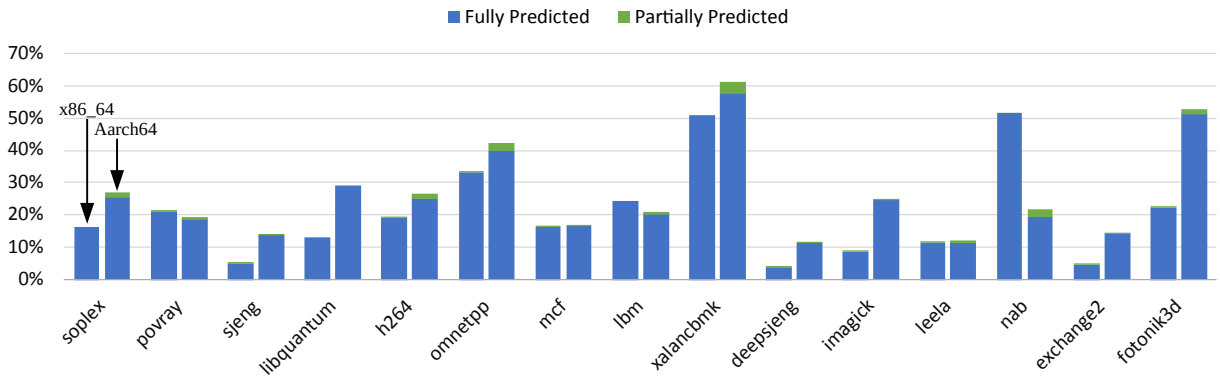


Figure 3.3 – Proportion of *fully* and *partially* predicted register-producing instructions.

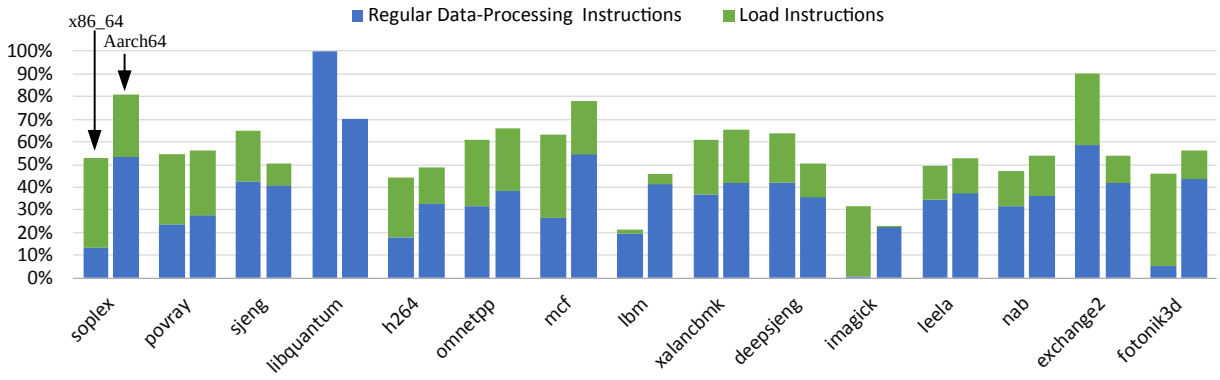


Figure 3.4 – Proportion of predicted instructions that were *useful*, classified in regular data-processing and load instructions.

structions is higher almost for each benchmark in Aarch64 (except from *nab*), in opposition to the respective speedup shown in Figure 3.2. The second one is that in general, the amount of partially predicted instructions is negligible in x86_64 and small but still sizable in Aarch64 (e.g., *soplex, povray, h264, omnetpp, xalancbmk, nab* and *fotonik3d*). Since partially predicted instructions leave unresolved dependencies, their contribution to the expected speedup should be less. Yet, their concentration in applications compiled for Aarch64 does not look enough to justify the lower speedup observed.

To further uncover the reason behind this event, we explore the *usefulness* of predicted instructions, a metric that firstly appeared in the study of Endo et al. [EPS17]. A predicted instruction is deemed *useful* when its predicted result is read from dependent instructions before the actual result is computed. Load instructions, that typically have higher execution latency than regular instructions, are relatively more *useful* to predict in most of the cases. In Figure 3.4 we present the proportion of predicted instructions in both ISAs that are characterized use-

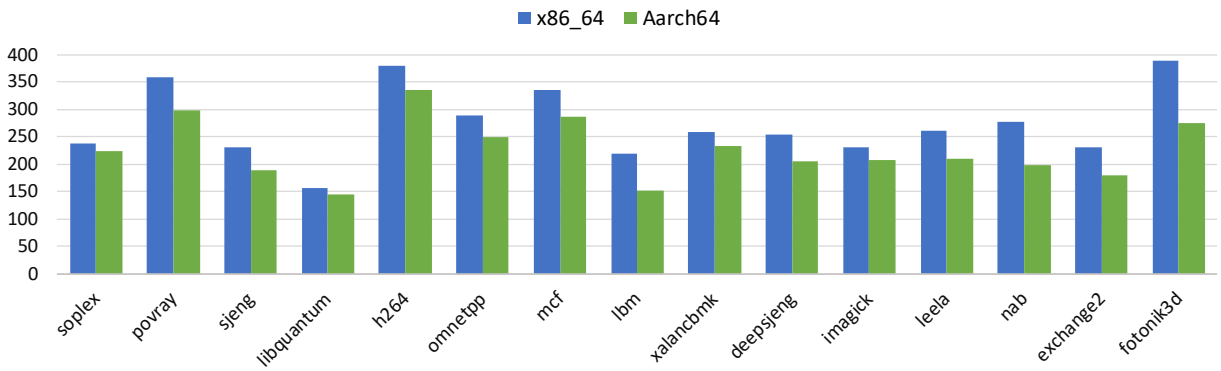


Figure 3.5 – Loads per kilo instructions (LPKI).

ful, separating loads from regular data-processing instructions.

Usefulness appears to be more correlated with speedup, but still not a "golden" rule, in the sense that high usefulness is not strictly followed by high speedup. For instance, in *libquantum* almost 100% of the predicted instructions in x86_64 are useful but speedup is negligible. However, usefulness serves adequately the purposes of this study as it essentially indicates the variant with the higher speedup (if any). For instance, in benchmarks like *h264* and *exchange2* with a substantial speedup-gap in the interest of x86_64, the portion of useful loads is accordingly larger in x86_64. Also, in the two cases that applications achieve higher speedup in Aarch64, either the fraction of useful loads or the total number of useful instructions is respectively higher.

Moreover, as we observe, in x86_64 the fraction of useful loads is always larger. Given that loads contribute more in pipeline stalls, predicting more loads than other instructions eventually leads to higher performance improvements. In other words, applications in x86_64 benefit more from value prediction because they feature more significant dependency chains (i.e. initiated by load instructions) that can be resolved.

The above allegation can be reasonably explained by the arithmetical difference in architectural registers between the two ISAs. In particular, x86_64 possess half the number of GPRs from Aarch64, resulting in code with higher concentration of load instructions. We illustrate this in Figure 3.5, characterizing the rate of *loads per kilo instructions* (LPKI) of our applications. Indeed, LPKI is higher for any application in x86_64. In essence, due to the less amount of available registers in x86_64, more instruction results are transferred to the memory in order to allow the reuse of registers. Then, when instructions need these values as their input, load instructions are in charge to fetch them. Since these loads have an implied pattern, it is very likely that they can be predicted. As such, value prediction in x86_64 appears to be less "*phase-dependent*" than typical, meaning that the code-regions that induce considerable pipeline stalls due to data dependencies are more frequent and predictable.

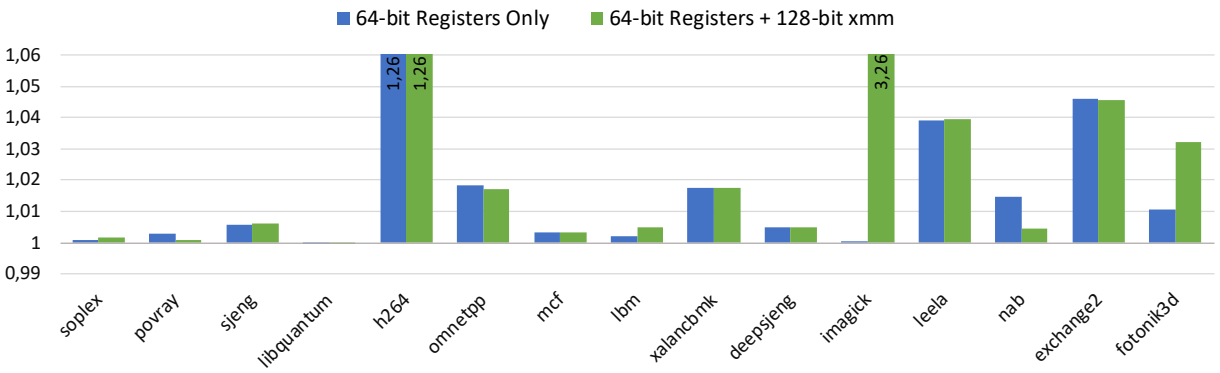


Figure 3.6 – Speedup brought by value prediction using VTAGE. Instructions eligible for VP are either those writing to a 64-bit register, or all register-producing instructions, including those writing to the 128-bit xmm registers.

3.5.3 Influence of Vector Optimizations

SIMD instructions have been traditionally introduced as extensions that can boost performance of floating-point intensive calculations in multimedia and other scientific applications. According to Figure 3.1a, there are roughly three applications in our set that contain a considerable amount of 64-/128-bit SIMD floating-point instructions, namely *lbm, fotonik3d* and *imagick*. Indeed, the first two programs are related with scientific computations for fluid dynamics and electromagnetics, respectively, and the third one with image manipulation.

As we explained above, in x86_64 ISA there are large portions of 128-bit SIMD FP instructions that are realistically not value-predictable and are excluded from value prediction in the previous experiments. Such exclusion limits considerably the amount of value predictable instructions in the above applications. Our intuition is that this reduction can influence negatively the potential performance gains through value prediction. In order to demystify this influence, in this section, we entertain the thought of quasi-predicting wider results of up to 128 bits by including 128-bit SIMD instructions in the value-predictable class. To do so, we consider solely the x86_64 ISA, the one including the largest amounts of non-predictable SIMD instructions, and in our implementation we authorize VTAGE to speculate on the results of instructions writing to 128-bit xmm registers, both integer and floating-point. As previously mentioned, *gem5-x86* apparently splits 128-bit instructions into two 64-bit ones, which makes it convenient to predict these wide instructions by simply considering the "chopped" µ-op as value predictable. In the following experiment, we actively embody this approach.

Figure 3.6 compares the relative speedup obtained through value prediction with VTAGE in x86_64 for two cases: the first blue bar represents a predictor that only considers 64-bit instructions, while the second one (i.e. the green bar) refers to a predictor that predicts also the results of 128-bit instructions. As expected, only the three SIMD-intensive applications that

we mentioned above are affected by this configuration, with the exception being *nab*. The last, although it has a relatively low amount of 128-bit SIMD instructions (representing around 7% of its register-producing instructions) shows some speedup reduction due to higher contention and aliasing in the predictor's table. When it comes to the rest applications that are affected, *lbm* shows some marginal improvement but interestingly *imagick* has a significant performance increase of more than 3x, while speedup in *fotonik3d* is also improved by 2%. From these results, we infer that vector optimizations limit considerably the potential of value prediction if one is limited to predict only 64-bit values.

3.6 Conclusion

In this Chapter, we explored value prediction from an ISA perspective, in the context of the x86_64 and the Aarch64 ISA. We first studied the internal characteristics of these two variants and then the performance of value prediction over them, using VTAGE [PS14b]. Demystifying the rationale behind the performance of value prediction is not a trivial procedure. It initially includes framing the potential of value prediction and then discovering its contribution to the obtained speedup (if any).

The raw potential of value prediction on each ISA can be naively expressed by the proportion of value predictable instructions in applications. Thereafter, the exploited portion of this "potential" or the coverage (i.e. how many instructions were finally *predicted*) gives some insight on how possible it is to improve performance. Note that although it may seem similar, this metric is not identical to the prediction coverage which is usually employed to show a predictor's effectiveness. A predictor's coverage denotes the portion of predictable instructions that were predicted, while the coverage we employed in our study declares the portion of register-producing instructions (predictable or not) that were predicted. However, our first observation is that neither the amount of predictable instructions nor the achieved coverage are tightly connected to the obtained performance gain. In particular, our analysis showed that even though applications in Aarch64 contain both more predictable and predicted instructions, speedup is consistently higher in x86_64. For that reason, we attempted to reply the following question: "*Are there any ISA-related aspects that affect the impact of value prediction?*".

To do so, we explored when a predicted instruction can be *useful* and which instructions are more *useful* to predict. Value prediction breaks true data dependencies between instructions, and therefore, allows *consumer-instructions* to potentially execute even ahead of their *producers*. In this way, a useful prediction cannot be other than the one that breaks a dependency chain that would stall notably the pipeline. Based on that, loads are almost always more useful to predict than short-latency instructions. Our experiments showed that in x86_64 VP usefulness stems more from the prediction of loads rather than any other instruction. In this

way, we found the reasoning behind the speedup variation between the two examined ISAs but more interestingly, we explained that this is a reasonable aftermath of the lower amount of general-purpose registers that x86_64 contains. Therefore, we showed that, indeed, there are ISA particularities that are highly connected to the performance of VP. Finally, we further showed the ISA effect on the VP performance by studying the influence of vector optimizations.

To put it all together, with this analysis we exposed the correlation between ISA-intrinsics and VP performance. Our findings show that it is not necessary for a value predictor to thrive in all ISAs. Already conducted research has focused on proposing novel designs that account for meaningful speedups regardless the ISA. As the VP performance appears to be sensitive to the underlying ISA, chip makers may need to study deeper its interaction with their hardware. Nevertheless, the achievable VP performance primarily depends on the efficiency of the used prediction algorithm to detect and exploit patterns in instruction results. In the next Chapter, we will show that applications exhibit patterns that can only be uncovered by the consecutive examination of instruction results. As these patterns have not been completely exploited yet, we will introduce a value prediction model that is able to capture these cases and to further increase performance. This particular value predictor is also highly independent from the registers length and as such, it can enable the prediction of instructions that produce values wider than 64 bits with an affordable storage budget.

CHAPTER 4

INTRODUCING THE BINARY FACET OF VALUE PREDICTION: VSEP

4.1 Introduction & Motivations

Current flagship processors tend to employ large instruction windows in order to efficiently extract more instruction-level-parallelism (ILP). But even then, potential performance improvements are inherently limited by true data dependencies. Value Prediction (VP) [LWS96], [LS96], [Gab96] was introduced as a technique to collapse these true dependencies by allowing dependent instructions to execute ahead of time using speculative sources. Initially proposed in the late 90's, VP has been a primary factor of contention for over a decade due to the high complexity that its implementation can impose. However, recent studies [PS14b], [PS14a], [PS15a] disclose that such complexity can be highly mitigated.

As already mentioned in Chapter 2, value predictors can be classified into two main families: *context-based* and *computation-based* [SS97]. Context-based predictors aim to generally capture the repetition of the same value, while computation-based predictors compute the predicted value from the last results of an instruction, e.g. stride predictors [GM98; ZFC03]. In essence, computation-based predictors require the actual or predicted result of previous instruction occurrence(s) in order to generate a fresh prediction. Predicting an instruction's result with a computation-based predictor requires searching the speculative window [PS15a] for potential inflight occurrences of the instruction to predict.

On the other hand, the state-of-the art context-based predictor VTAGE [PS14b] does not require such a complex integration in the out-of-order (OoO) execution engine, by simply relying on a set of tables read at fetch time. VTAGE aims to capture the correlation of instruction results with the global branch history, i.e., the same value encountered with the same global branch history. To achieve this functionality, VTAGE is implemented with a plurality of tables indexed with different branch history lengths. Each entry in these tables features a full 64-bit value, a partial tag and a confidence counter, i.e., around 80 bits in total.

In this study, we aim to capture another form of this regularity: *equal values on consecutive occurrences of the same instruction*. Two different kinds of *value equality* can be discriminated:

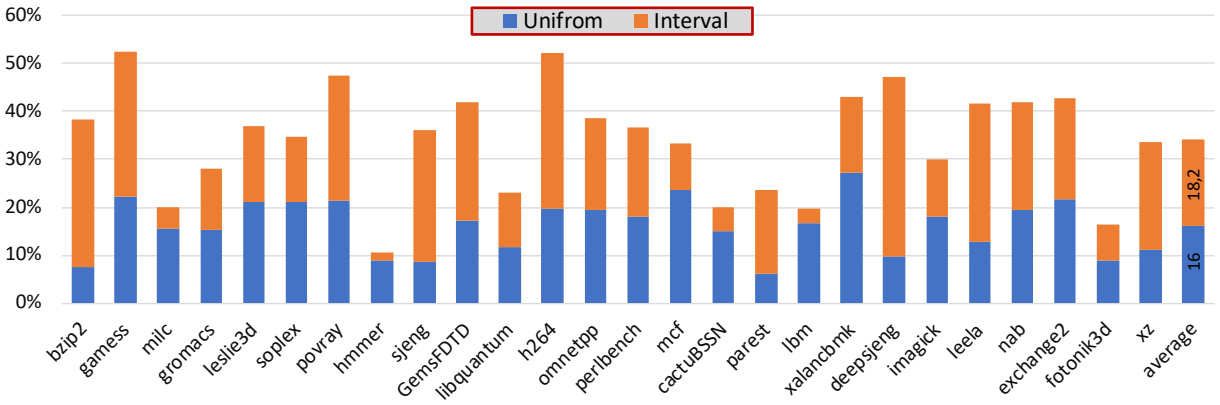


Figure 4.1 – Classification of instructions according to the form of consecutive value equality they expose. The two classes are mutually exclusive.

1. **Uniform**: All dynamic instances of a static instruction always produce the same result (e.g. a constant value sequence [SS97]).
2. **Interval-style**: Within an interval of repetitive executions, a static instruction produces the same result. But for each interval the result is different. Practically, it resembles the uniform pattern limited within an interval.

In other words, *value equality* is not always related with a single recurrent value because consecutive executions might expose equality on intervals for different values.

Equality of the results of two consecutive occurrences of an instruction is very frequent. We illustrate this in Figure 4.1, characterizing *uniform equality* and *interval equality* of our benchmarks (discussed in Section 4.3.2). As reported, interval equality is prevalent across the applications considered in our study, representing on average more than 18% of the instructions and plainly dominating the rates of uniform equality in 12 out of 26 cases. However, to our knowledge, no previous context-based value prediction scheme has been designed with total awareness of interval equality, but rather of uniform equality.

In this Chapter, we introduce the notion of *Equality Prediction* (EP), which represents the *binary facet of value prediction*. Following a twofold decision scheme (similar to branch prediction), EP makes use of control-flow history to determine *equality* between the last committed result read at fetch time, and the result of the fetched occurrence. When equality is predicted with high confidence, the read value is used. Altogether, we refer to this scheme as the technique of *Value Speculation through Equality Prediction* (VSEP). VSEP is practically a context-based value predictor that can exploit both types of *consecutive equality* that instruction results may exhibit (i.e. *uniform* & *interval*). Similar to VTAGE, VSEP does not require a complex integration in the out-of-order execution engine. Though, contrarily to VTAGE, VSEP does not require wide entries in all the predictor tables.

To allow performance gains through VP, reaching a high confidence value generally necessitates many successive correct predictions (typically more than 100) [PS14b]. Commonly, in context-based predictors (from the oldest to the most current [LS96; PS14b]), the prediction confidence is associated with the value to be predicted in the same predictor entry. As a result, on instructions exhibiting interval equality, prediction reaches high confidence only on long intervals. In VSEP, we do not associate the confidence with the value in the same predictor entry, but rather predict some binary information, i.e., whether the result of the currently-processed occurrence of instruction *Z* is very likely to be equal to the last committed value for instruction *Z*. More precisely, VSEP consists of a TAGE-like *equality predictor* that we call *ETAGE*, and a table that tracks the last **committed** values of instructions, namely LCVT. ETAGE detects (likely-)equality without explicitly defining the value to be predicted by using the value provided from the LCVT. As a consequence, when the VSEP predictor is able to identify the beginning of intervals, it may resume useful (i.e. high confidence) predictions as soon as it can anticipate with high confidence that the first instruction occurrence in the interval has been committed.

Our simulations show that VTAGE and VSEP performance improvements are in the same range, but their prediction coverage is partially orthogonal.

This study makes the following contributions:

— We identify two different forms of *consecutive value equality*, the **uniform** and the **interval**, and show how the latter diminishes value predictability.

— We propose ETAGE, an equality predictor that solely identifies equality between the current execution of a static instruction and its last committed result.

— We propose VSEP, a value predictor that leverages the ETAGE equality predictor to perform value prediction.

— We present a comprehensive analysis of our scheme and compare it against prior work in value prediction, revealing how our solution complements existing techniques by increasing the fraction of predicted instructions that expose interval equality.

— Finally, we propose a practical integration of the state-of-the-art VTAGE with VSEP that delivers approximately 7% of average speedup, i.e., 19% higher than VTAGE/VSEP alone.

4.2 Value Speculation through Equality Prediction - VSEP

In this Section, we describe VSEP, our proposed scheme for leveraging *Equality Prediction* (EP) in order to perform value prediction. Then, we also present a synopsis of its prediction logic with a comparison to already established methods of value prediction.

4.2.1 Value Prediction using ETAGE

VSEP consists in two distinct components, ETAGE, the equality predictor, and LCVT, the Last Committed Value Table.

ETAGE is a context-based equality predictor that essentially copies the TAGE branch predictor structure [SM06]. That is, it employs a series of prediction tables that are indexed by a different number of bits of the global branch history, hashed with the instruction PC. These main tables are also backed up by a tag-less base table that is directly indexed with the instruction PC. The entries of the tagged tables contain the 1-bit equality information, a partial tag, a 1-bit usefulness indicator and a 3-bit confidence counter. Since we leverage EP to guide the speculation of instruction results, high accuracy is essential to avoid performance degradation. Hence, as already established by modern VP-schemes, we use confidence counters to discriminate the predictions that are promoted to the pipeline. Therefore, a predicted value is used by the pipeline only if *equality* is predicted with high confidence. To ensure that high confidence is reached only after a large (i.e. sufficiently safe) number of equality occurrences, ETAGE-entries feature *forward probabilistic counters* (FPCs) [RZ06] of 3 bits. Experimentally, we found that using the probability vector V = $\{1, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}\}$ to control forward transitions provides a good trade-off.

On the other hand, LCVT records the last committed value of instructions. The particular LCVT that we employ is a 3-way set associative table of no more than 3K entries, as we found that this size is sufficient to track execution in our framework (discussed in Section 4.3.3). Along with the full 64-bit committed value, each entry includes a 13-bit tag, which is a folded hash of the instruction PC. In the paragraphs that follow, we detail the prediction and training logic that VSEP follows.

4.2.1.1 Prediction

ETAGE delivers a prediction after accessing all its main tables in parallel in a search of the tagged entry. When matching entries are present in multiple tables, the prediction arrives from the one accessed with the longest history. In the absence of any matching entry, the base table provides the prediction. Also, concurrently to the prediction process, the LCVT is accessed to provide the last committed value of the relevant instruction. On a miss in the LCVT, *inequality* is assumed. Recall that the predicted value, if any, is used in the pipeline only if *equality* is predicted with high confidence.

4.2.1.2 Training

At update time, the entry of ETAGE that provided the prediction (i.e. equality or inequality) is updated with the execution outcome, i.e. the actual equality/inequality of the current instruction

result with the speculative value retrieved from the LCVT at prediction time. Their agreement triggers the increment of the entry's confidence counter, according to the probability vector described above. Otherwise, on a misprediction, the entry's equality bit is replaced if its confidence is already equal to zero. If not, the confidence counter is reset, so that the entry's equality bit will be corrected in the next execution. Moreover, a new entry is allocated in a "higher" table (i.e accessed with a longer history), following a similar allocation policy with the TAGE [SM06] branch predictor. In addition, the usefulness bit that leads the replacement policy, is set when the produced prediction is correct and the prediction that could have been alternatively derived from a table with shorter history is inverse. In the TAGE algorithm, this alternative prediction-option is called *alternate prediction* [SM06]. In the opposite case (i.e. when the alternate prediction is actually correct since it is the inverse of the processed one), the usefulness bit is reset. In any case, the usefulness bits of all the tagged entries are periodically reset to avoid the possible endless protection of certain entry replacements. Finally, regardless of the execution outcome, the LCVT is updated with the committed result.

4.2.1.3 Pipeline Details

In VSEP, validations are done in-order at commit time, since this allows reduced complexity of the OoO core by banking the physical register file [PS14b]. To enable validation at commit, a FIFO queue with size equal to that of the instruction window is employed. Each entry holds the 1-bit equality prediction of ETAGE and the value retrieved from the LCVT at prediction time (around 2KB of storage space is required considering the 256-instructions window of our framework). We assume the following process:

— At Fetch, leverage the ETAGE equality predictor to generate a high accuracy prediction that specifies the equality of the current instruction result with the last committed value. In parallel, index the LCVT to acquire the instruction's last committed value. Also, place both the equality prediction and the potentially predicted value in the validation queue.

— At Rename, if *equality* is predicted with high confidence, and a hit on the LCVT is encountered, the predicted value is written to the physical register file. Conversely, when *inequality* is predicted or *equality* does not have sufficiently high confidence, the predicted value is not used.

— At execution, overwrite the predicted value with the computed one.

— Before Commit, use the validation queue to validate the predicted value against the computed result. Flush the pipeline on a misprediction if the predicted value was inserted in the pipeline. Similarly, validate the equality prediction in order to adequately update the ETAGE predictor. Finally, update the LCVT with the just committed result.

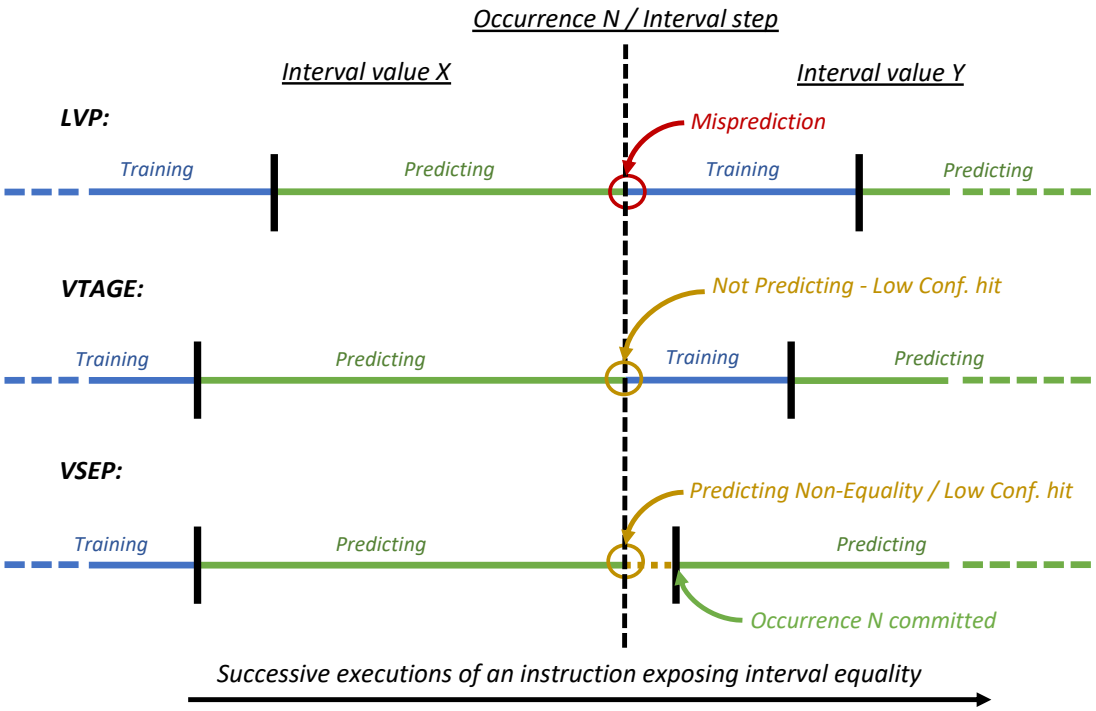


Figure 4.2 – Prediction scenarios in presence of interval equality.

4.2.2 Dissecting the Prediction scenario of VSEP

In previous context-based value predictors, the predicted value is precisely defined by the predictor's entries that tightly associate the predicted value with its confidence. With VSEP, the predicted value is not directly specified by the equality predictor ETAGE. ETAGE solely defines the potential equality/inequality of the current instruction-instance result with the last committed value of the same static instruction. When high-confidence equality is predicted, the corresponding last committed value constitutes the predicted value. In these cases, the value that is used to perform the prediction of the instruction's result is provided from a cache-like table of committed values, the LCVT. Overall, the unique feature of VSEP prediction scheme is that ETAGE makes a *de facto* decision of whether value speculation can take place independently from the eventually predicted value.

Being a context-based predictor that leverages the global branch history as a context, ETAGE can exploit recent and less recent control-flow history, similarly to VTAGE. Both predictors implement the TAGE algorithm [SM06]; therefore they are both able to identify precise positions in the control flow path. Below we describe the general prediction scenario for LVP, VTAGE and VSEP on instructions featuring interval value equality. This is illustrated in Figure 4.2. We assume that the examined instruction Z returns the value X on the first interval, then

flips to value Y on the second interval.

LVP learns the equality in the first interval and reaches high confidence. At this point, the prediction from LVP can be used in the pipeline. When the result flips to Y, a misprediction is encountered and a misprediction penalty must be paid. Thereafter, LVP has to re-train on the second interval before re-reaching high confidence and becoming used in the pipeline again.

ETAGE also learns the equality in the first interval. Moreover, when the value change is correlated with the control flow path, ETAGE learns also to detect it and therefore it will not suffer a misprediction on the flip to Y. When fetching after the *Occurrence N* of instruction that flips to Y, the value read from the LCVT remains X till the *Occurrence N* has retired. ETAGE is able to identify this in the control-flow history. In practice, the entries associated with the first occurrences do not reach high-confidence levels and the predicted values are not used. When the Occurrence N has retired, or more precisely, is very likely to have retired (i.e., the distance between the currently fetched and the *Occurrence N* is larger than the instruction window size), the LCVT provides the value Y and ETAGE identifies the presence of equality. Since ETAGE entries for the first occurrences and further occurrences are distinct, prediction from ETAGE exhibits high confidence as soon as the first occurrences have been fetched. At an interval change, the number of occurrences that are not predicted is proportional to the size of the instruction window divided by the distance (in number of instructions) between two consecutive occurrences of the instruction.

VTAGE will also learn of equality in the first interval, and may also learn of a value change. However, the new value Y has to be re-learned by the predictor entries, and therefore the prediction does not reach high confidence before a large number of occurrences of the instruction Z (generally more than 100).

In summary, for instructions that exhibit interval equality, VTAGE and LVP need very long intervals to learn the equality (in the sense of reaching high confidence), and as a result, the predicted data are not sent to the pipeline. VTAGE is often able to learn of interval changes. Although VSEP also learns of interval changes, it learns as well the paths leading to the first instruction occurrences in the interval, and when the value from the LCVT is safe to use in the pipeline. As it follows, VSEP generally predicts more occurrences than LVP and VTAGE on instructions featuring interval equality. Nonetheless, it should be pointed out that VTAGE can learn some other scenarios that VSEP is not able to track, even on instructions featuring interval equality, as will be shown in the performance analysis of Section 4.4. One of these scenarios is the case of repeatable short intervals (for instance of 5 occurrences) with distinct values (X0,..,Xk). Contrary to VSEP, VTAGE will be able to track these cases.

Table 4.1 – Basic layout and size of the examined predictors.

LVP [LS96]	4K-entry LVT, 13bit-tags, *40KB*
VTAGE [PS14b]	4K-entry Base Component, *33,5KB* 6 x 512-entry tagged tables, tags 12+rank bits, *30,5KB*
VSEP	**ETAGE**: 8K-entry Base Component, *4 KB* 13 x 1024-entry tagged tables, tags 8+rank bits, *32KB* **LCVT**: 3K entries, 3-way associative, 13bit-tags, *28KB*
Hybrid VSEP-VTAGE	**ETAGE**: 8K-entry Base Component, *4 KB* 13 x 512-entry tagged tables, tags 8+rank bits, *16KB* **LCVT**: 3K entries, 3-way associative, 13bit-tags, *28KB* **VTAGE Tagged tables**: 6 x 256-entry, tags 12+rank bits, *15KB*

4.3 Evaluation Methodology

4.3.1 Examined Predictors

Together with VSEP, we study the behavior of two context-based value predictors, namely, the classic LVP and the state-of-the-art VTAGE predictor. Our intention is to compare and evaluate our scheme against a trivial and a modern model of VP. In their implementation we use FPC of 3 bits that are controlled by the probability vector V = $\{1, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}\}$, i.e. the same that was employed in the study that introduced VTAGE [PS14b].

We use predictors with a storage budget in the 64KB range, i.e. a storage budget in the same range as the one of branch predictors in high-end processors. More precisely our LVP predictor employs 4K entries for a total budget of 40 KB. Doubling this size does not bring any significant benefit in our experiments. Then, we derive a proportional design of VTAGE with a 4K-entry base component, since this component is practically a tag-less LVP. VTAGE also includes 6 tagged tables of 512 entries each, amounting to a total budget of 64KB.

Accordingly, we design our ETAGE predictor to reach the same storage budget as VTAGE. The shortest and longest history lengths are 2 and 64 respectively, following a geometric series. We consider a predictor featuring 13 history components together with an 8K-entry tag-less table (i.e. a base component that basically holds *last equality* of instructions), that accounts for 36KB of needed space. This, added with a 3K-entries LCVT (i.e 28KB,) completes the storage requirements of our design. Table 4.1 summarizes the design parameters and the size of all the predictors that we consider in our analysis. Note that the *Hybrid VSEP-VTAGE* predictor that is shown in Table 4.1 will be presented later in Section 4.4.2.

4.3.2 Benchmarks

We cast a wide range of workloads from the suites of SPEC2K6 [Staa] and SPEC2K17 [Stab] in order to expose as many value patterns as possible. For the workloads duplicated in

Table 4.2 – Applications used in our evaluation.

Benchmark Suite	Applications
SPEC2K6 INT/FP	bzip2, gamess, milc, gromacs, leslie3d, soplex, povray, hmmer, sjeng, GemsFDTD, libquantum, h264, omnetpp
SPEC2K17 INT/FP Rate	perlbench, mcf, cactuBSSN, parest, lbm, xalancbmk, deepsjeng, imagick, leela, nab, exchange2, fotonik3d, xz

both suites, we consider the ones from SPEC2K17. Table 4.2 lists the benchmarks considered in our study. To get relevant numbers, we identify a region of interest in the benchmark using Simpoints 3.2 [Per+03], as value prediction is highly sensitive to phase behavior. We simulate the resulting slice of 150M instructions by warming up the processor (caches, predictors) for 50M instructions and then collecting statistics for 100M instructions. For comfort of reading, an extra bar was added to present the average value of the evaluation metrics we use. For this average, we present results as the arithmetic average across all workloads, except from the relative speedup where we calculate the geometric mean of results.

4.3.3 Simulator

In our experiments, we reuse the framework described in the previous Chapter. That is, a modified [1] version of the *gem5* cycle-level simulator [Bin+11] implementing the x86_64 ISA. We prefer to use the x86 ISA as it recruits instructions that can have at most one INT or FP destination register, allowing to effectively capture the benefit of value prediction when couples of *producer-consumer* instructions appear in applications (see Chapter 3). However, it should be noted that contrarily to modern x86 implementations, *gem5* does not support optimizations like *move elimination* [Jou+98; PSR05] and *µ-op fusion* [Goc+03].

The characteristics of the superscalar baseline we consider are summarized in Table 3.1 and detailed in Section 3.4.1. In particular, we employ a superscalar pipeline with a fetch-to-commit latency of 19 cycles, where 15 cycles correspond to the frontend and 4 cycles to the backend. When value prediction is used, a pre-commit stage responsible for validation/training (VT) is added to the pipeline. The VT stage lengthens the pipeline by one cycle, resulting in a value misprediction penalty of at least 21 cycles, while minimum branch misprediction latency remains unchanged. Note that predictors are effectively trained after commit, but values are read from the PRF during the VT stage.

The operation of the value predictors in the pipeline is similar to the one described in Section 3.4.2.2 of the previous Chapter for x86_64. That is, the predictor makes a prediction at *Fetch* stage for any eligible instruction (i.e. producing a register of at most 64 bits, as defined by the ISA implementation in the simulator). To index the predictor, we XOR the instruction's PC

1. Branches are implemented in a single µ-op instead of three.

address left-shifted by one with the µ-op number inside the instruction. With this mechanism we ensure that more than one µ-ops of the same instruction will generate a different index and therefore, they will have their own entry in the predictor. We assume that the predictor can deliver fetch-width predictions by the Fetch stage.

4.4 Experimental Results and Analysis

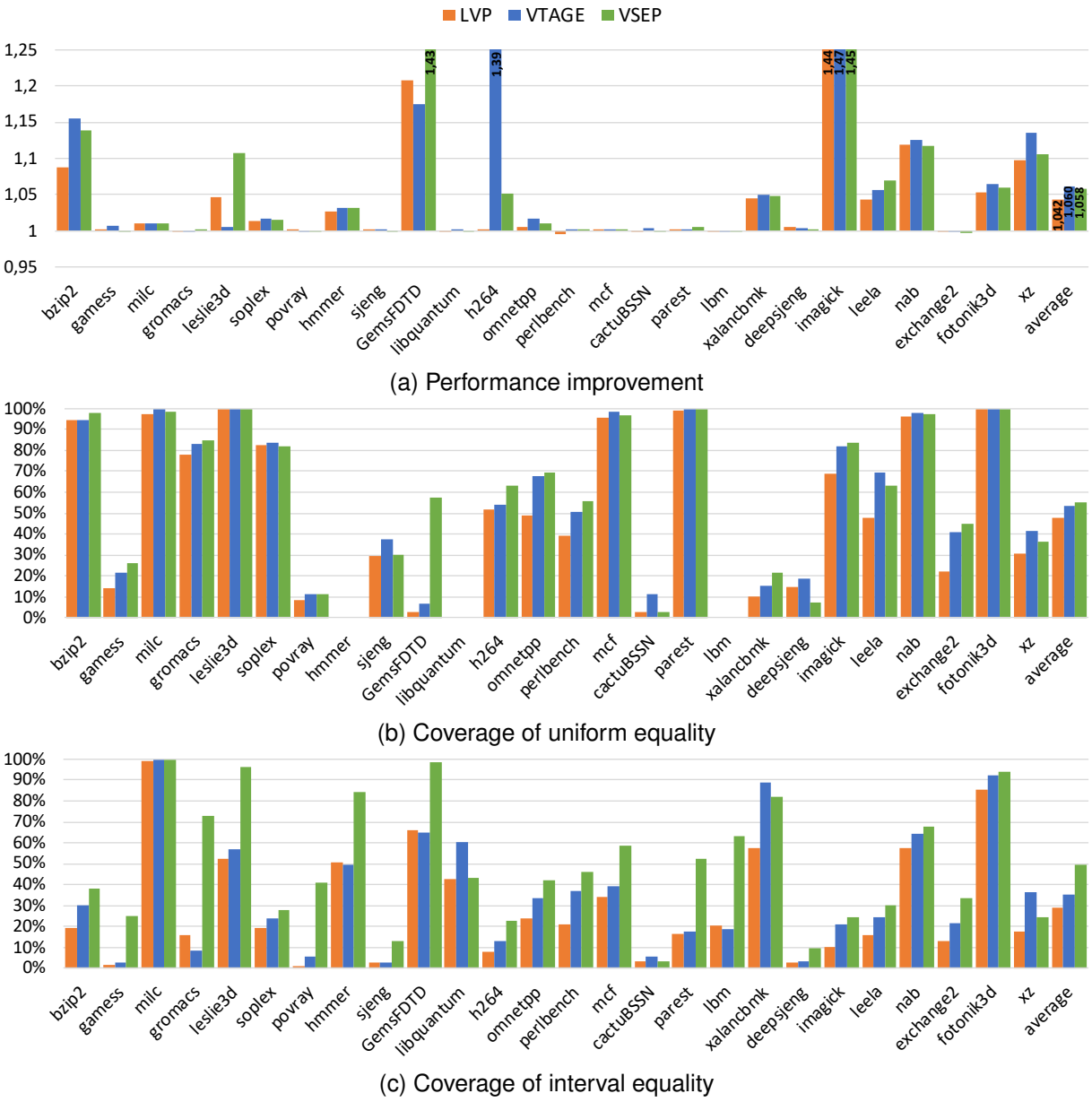
In this section, we demonstrate the strengths of VSEP. We compare our model to related prior work, the fundamental LVP and the state-of-the-art VTAGE, both implemented as described in Section 4.3.1. Overall, our evaluation shows that VSEP is partly orthogonal to traditional context-based VP techniques. This leads us to define a hybrid predictor, that of VSEP and VTAGE as a unified design. We propose a practical implementation of this combination, and finally compare the hybrid predictor with VTAGE and VSEP alone.

4.4.1 VSEP vs VTAGE and LVP

Figure 4.3 reports our simulation results by comparing the three different value predictors that we consider: LVP, VTAGE and VSEP. In particular, Figure 4.3a shows the relative IPC of the three variants normalized to the baseline. Figures 4.3b and 4.3c show the fraction of predicted instructions from those exposing either uniform or interval value equality. That is, *uniform coverage* (Figure 4.3b) describes the portion of instructions that exhibit uniform equality and their result was predicted, while *interval coverage* (Figure 4.3c) similarly expresses the predicted portion of the instructions that exhibit interval equality. Note that a prediction is used in the pipeline only in high confidence. These two metrics should not be confused with the general prediction coverage that is defined as the number of predicted dynamic instructions divided by the total number of dynamic instructions that are eligible for value prediction.

4.4.1.1 Performance Analysis

Our evaluation indicates that VSEP essentially benefits the same benchmarks with respect to typical context-based predictors and heavily competes with the state-of-the-art VTAGE. Specifically, it improves 14 out of 26 workloads by more than 1%, with a maximum speedup of 45% on *imagick* and an average speedup of 5.8%. In addition, VSEP succeeds in always obtaining equal or higher performance to the fundamental LVP. On several benchmarks, VSEP outperforms VTAGE e.g on *leslie3d* and on *GemsFDTD*. On the other hand, VTAGE outperforms VSEP on some other benchmarks e.g. on *h264*, and *xz*. Finally, the two predictors achieve average performance in the same range.



(a) Performance improvement

(b) Coverage of uniform equality

(c) Coverage of interval equality

Figure 4.3 – Comparison of the three prediction schemes: LVP, VTAGE and VSEP.

Readers will notice that the reported speedup of VTAGE is lower than that of LVP for two applications, namely for *leslie3d* and for *GemsFDTD*, even though their two-fold coverage is in the same range. After a closer examination, we found that the potential performance gain of VTAGE for these benchmarks is limited by bursts of mispredictions associated with the interval transitions. This peculiar behavior may happen in VTAGE when applications expose very long value equality intervals. In Figure 4.4 we display a thorough classification of all the instructions exposing interval equality depending on the relevant interval length. The length of an interval is

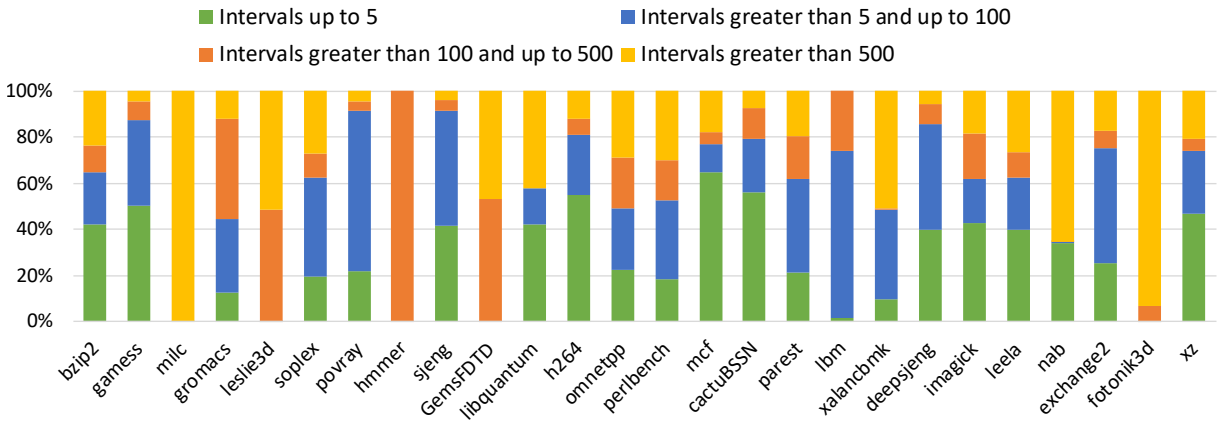


Figure 4.4 – Segmentation of instructions that exhibit interval equality in very short, moderate and very long intervals.

defined as the number of successive executions of a static instruction that produce the same result.

As we mentioned in Section 4.2.2, VTAGE can identify value switches without mispredicting when the correlation with the control-flow path is high enough. However, during very long intervals VTAGE may establish the confidence of the same recurrent value in several of its tagged-entries. Hence, upon an interval step, a burst of mispredictions will occur when these entries will be successively hit. In our simulations, both *leslie3d* and *GemsFDTD* suffer from this phenomenon, as the overwhelming majority of their *interval equalities* occur in intervals of more than 100 occurrences. Seznec experienced a similar anomaly in his VTAGE-like value predictor EVES [Seza], and proposed the use of a prediction filter in order to defeat such bursts of mispredictions (no prediction, high-confidence or not, is promoted to the pipeline for 128 µ-ops after a misprediction).

On the contrary, such a prediction filter is not required by VSEP. Similarly to VTAGE, ETAGE may also establish the confidence of *equality* in multiple entries during long intervals. When instruction results correlate with the control-flow history, ETAGE identifies also value switches by matching with entries that either possess low confidence or indeed predict *inequality*. However, when ETAGE starts respectively hitting to entries with already high-confidence *equality* from the previous interval, it allows VSEP to immediately re-enter prediction mode using the correct value retrieved from the LCVT. Therefore, due to its construction, VSEP not only captures instructions that expose this behavior, but it also benefits highly from that by delivering significantly larger speedup in both aforementioned applications, i.e. in *leslie3d* and *GemsFDTD*.

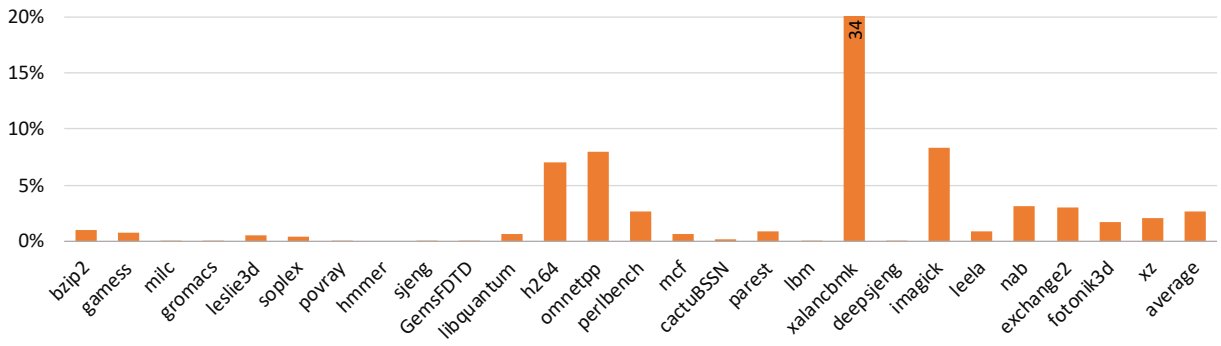


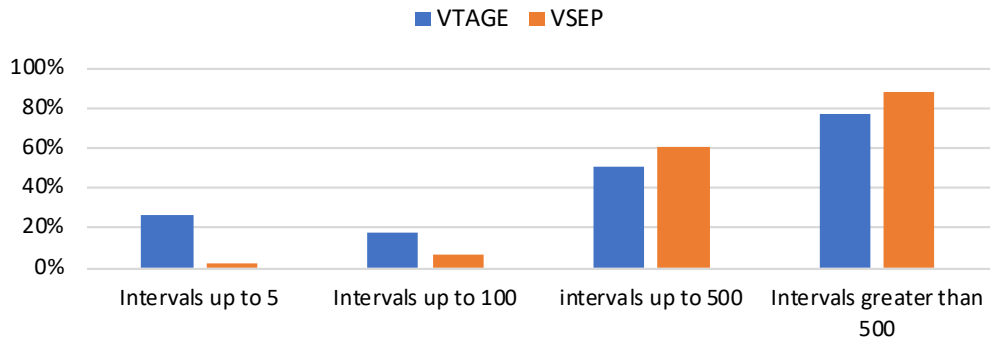
Figure 4.5 – Proportion of instructions that were predicted by VTAGE and do not exhibit either *uniform* or *interval* equality.

4.4.1.2 Coverage Analysis

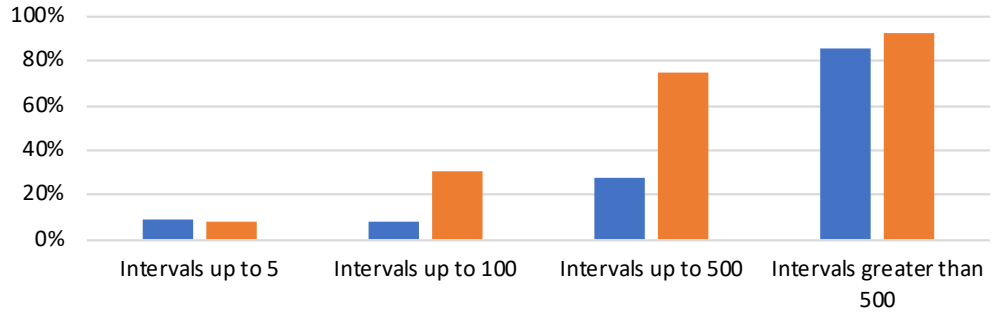
Figure 4.3b indicates that both VSEP and VTAGE achieve similar coverage of uniform equality of around 54% on average, which is marginally better than LVP that attains 48%. This behavior is expected since *uniform equality* is relatively simple to capture, even from predictors that do not leverage an elaborate context to distinct dynamic instructions (e.g. the global branch history used both by VTAGE and VSEP). Although, when it comes to the coverage of interval equality, Figure 4.3c illustrates that VSEP successfully surpasses both LVP and VTAGE on 22 out of 26 workloads, achieving 50% on average, versus 35% for VTAGE and 29% for LVP. As a matter of fact, the behavior of VSEP that we have detailed in Section 4.2 is verified by our experimental results, i.e., that VSEP is able to generally cover cases of interval equality that both LVP and VTAGE miss. Thus, VSEP is capable of complementing the established way that VP is performed.

Yet, higher interval/uniform coverage for VSEP on many benchmarks does not always mean higher speedup. As in the case of *h264*, VTAGE significantly outperforms VSEP when it only has half of its interval coverage and similar uniform. In fact, the general prediction coverage of VTAGE is not composed only by the instructions that expose one of the two flavors of value equality that we have identified. In reality, the prediction range of VTAGE also includes independent patterns, e.g. sequences of repeatable strided values (see the detailed explanation in Section 2.2.4.1), that VSEP barely captures. Figure 4.5 presents exclusively this proportion of instructions that VTAGE predicts, revealing that on average 3% of its global prediction coverage corresponds to the prediction of instructions that exhibit neither form of consecutive value equality (i.e. *uniform* or *interval*). As it follows, this additional coverage explains why VTAGE can dominate over VSEP in cases of similar or even lower interval/uniform coverage.

Moreover, VTAGE is occasionally able to capture some interval equality that is missed by VSEP e.g. on *libquantum* and *xz*. By selectively focusing on the case of *xz*, where there is



(a) Analysis of interval coverage explicitly for application *xz*



(b) Analysis of total interval coverage across all workloads

Figure 4.6 – Anatomy of interval coverage for VTAGE and VSEP, explicitly for application *xz* and totally for all the examined applications.

substantial performance gain from VP, we present in Figure 4.6a the synthesis of the interval coverage for VSEP and VTAGE on this specific benchmark. As we mentioned in Section 4.2.2, VTAGE can obtain higher interval coverage than VSEP when instructions rotate between repeatable short intervals. Unlike the binary-content VSEP, VTAGE is able to exploit the reappearance of the same value even in non-consecutive intervals and continue enhancing the confidence of corresponding value-entries. Indeed, as illustrated in Figure 4.4, a significant portion of *consecutive value equalities* found in *xz* occur in intervals that are shorter than 5 occurrences. As reported by Figure 4.6a, the coverage superiority of VTAGE is concentrated in these short intervals, where VTAGE achieves around 26% while VSEP can only reach 2%.

Finally, Figure 4.6b displays the segmented representation of the total interval coverage of VTAGE and VSEP, considering our entire benchmark pool. Both schemes accomplish roughly equal levels of coverage for very short intervals, but VSEP far outweighs VTAGE in all the rest situations. In particular, VSEP achieves very high rates of coverage in both moderate (i.e. up to 100 and up to 500 occurrences) and very long intervals (i.e. more than 500 occurrences), which correspond to around 75% and more than 90%, respectively. On the other hand, VTAGE can merely approximate these rates only for very long intervals. Above all, this observation exposes the operational asset of VSEP over VTAGE under the state of interval value equality. More

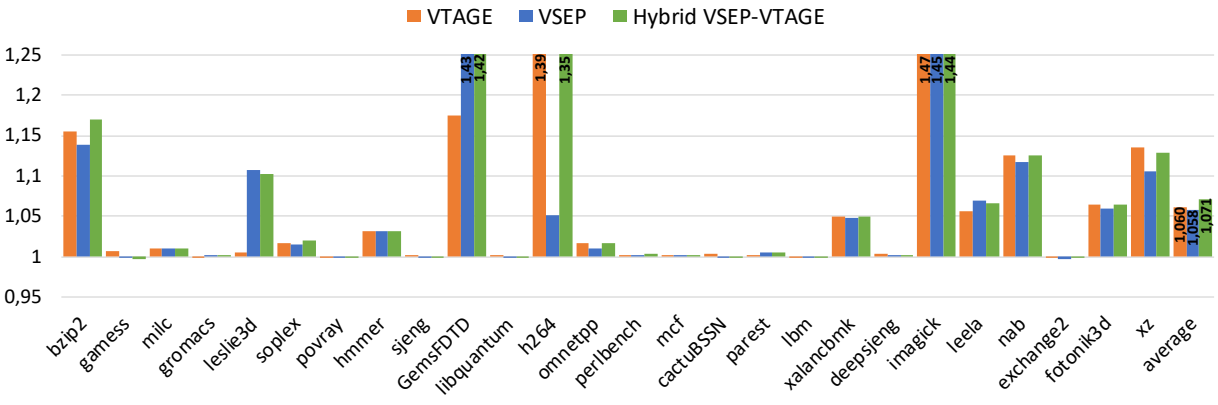


Figure 4.7 – Performance improvement of the compound model.

specifically, it confirms our thesis that the already established VP methods cannot sufficiently capture interval equality, unless intervals are very long.

Altogether, our performance evaluation suggests that both VSEP and the state-of-the-art VTAGE can individually obtain noticeable speedups, but none of the two can plainly outperform the other. This advocates for hybridization. In the following Section we conclude by presenting a compound model of VSEP and VTAGE.

4.4.2 Combining VSEP with VTAGE

As verified by our experimental analysis, VSEP can accomplish meaningful performance improvements, either comparable or higher to an established value predictor as VTAGE, by eliminating its essential weakness to make use of interval value equality. Our performance analysis also showed that VSEP and VTAGE can independently capture different cases. Since the two methods can be considered as *partly orthogonal* with respect to the cases they can capture, we study the combination of VSEP with a short-scale adaptation of VTAGE, in order to provide the best coverage and boost performance.

VSEP-VTAGE employs an ETAGE equality predictor, accompanied with a LCVT, and a moderately-sized VTAGE, which does not encompass its base component (i.e., a simple last-value table or LVT). The objective behind the balanced sizes of the two integrated predictors is to preserve their philosophy and to eliminate the overlapping between them. VSEP-VTAGE works as follows: both predictor components are indexed in parallel in the early frontend. No elaborate or random chooser is necessary to clarify which predictor makes the final prediction. Simply, if *value equality* is predicted by ETAGE, the prediction of VSEP is the one that proceeds, otherwise, the prediction of VTAGE is considered for use to cover the cases missed from VSEP. For validation/update, each entry of the FIFO queue in VSEP (as described in Section 4.2.1.3)

is augmented with the predicted value of VTAGE.

In Figure 4.7 we present the speedups brought by VSEP-VTAGE in comparison with VTAGE and VSEP alone at similar storage budgets (see Table 4.1 for a size analysis of the hybrid model and the other predictors). Evidently, our hybrid solution can efficiently combine the two schemes by retaining their independent benefits. For any benchmark individually, the perceived speedup is in the range of the highest between VSEP and VTAGE, and on average is augmented by 19% from VSEP/VTAGE alone, obtaining an overall average of 7.1%. Note that the extra storage budget required from the composite scheme (compared to a single VSEP predictor) solely concerns the additional entry in the validation queue for the prediction of VTAGE. Hence, these meaningful speedups are acquired in trade for minimum storage/complexity overhead.

In this section, we showed that VSEP-VTAGE forms a smooth combination that overcomes the different limitations imposed by its two counterparts, since both predictors work complementary to each other. In general, *equality prediction* is a technique that enhances performance gains of modern context-based VP-schemes and paves the way for the development of novel approaches to efficiently extend the value prediction range.

4.5 Conclusion

Context-based value predictors represent an important class of value predictors [SS97]. They exploit the recurrent occurrences of the same result by a single instruction. Some instructions deliver always the same result and can be predicted by simple VP methods like LVP. Other instructions produce results that follow more complex patterns, such as interval-style. Overall, these instructions represent the most significant part of the predictions covered by a state-of-the-art context-based predictor, such as VTAGE. In other words, consecutive value equality is a key feature of real-world programs [Staa; Stab] that in essence enables value prediction. Nevertheless, modern context-based value predictors were not designed in full awareness of this attribute.

VSEP was introduced to specifically target interval-style value equality, while certainly covering the uniform category as well. Instead of predicting a wide 64-bit value, VSEP predicts some binary information, i.e. whether, namely "*is the instruction's current result likely to be equal to its last committed value*". Thus, unlike regular VP-schemes that embody structures of 64-bit wide entries, VSEP features only a single *Last Committed Value Table* with full "words" and a binary-content equality predictor ETAGE. Compared to the state-of-the-art VTAGE, VSEP captures the two forms of value equality more efficiently. In practice, when it comes to interval equality, after a value switch from *X* to *Y*, VSEP is able to predict equality with high confidence as soon as the first occurrence of the new value Y has been committed. On the other hand, VTAGE has to reconstruct high confidence for each of the entries that has been predict-

ing *X*. As a result, VSEP outperforms VTAGE on several applications. Nonetheless, VTAGE is able to predict value patterns that do not fit in consecutive occurrences and therefore evenly outperforms VSEP on some other applications. Eventually, the effective hybrid combination of VSEP and VTAGE introduced in this work leads to some extra performance gains by retaining their particularities. Recall that no complicated arbitration has been employed to choose which predictor will provide the prediction. Such a simple integration shows practically that VSEP captures patterns that have not been examined yet.

As we mentioned above, VSEP accommodates full values only in its LCVT part. In our framework, VSEP features 3K value-entries, while VTAGE contains 7K value-entries, as it typically holds full values in all its entries. Therefore, when VSEP is evaluated alone, it achieves performance improvements in the same range with VTAGE by essentially employing around 58% less value-entries. In this way, regarding our observation in Section 3.5.3 about the influence of vector optimizations in value prediction, VSEP can effectively mitigate the cost of performing value prediction for wider register-values (e.g. equal to 128 bits).

Regardless of the employed model, value prediction aims to increase sequential performance by mastering ILP. Long-latency instructions are relatively more useful to predict as they can introduce significant pipeline stalls. Arguably, load instructions stand out of the rest since their execution latency can be very large depending on the cache-level that will serve their request. However, predicting the value of a load instruction does not limit its latency, rather makes it tolerable by parallelizing it with other instructions (subsequent or not). Considering techniques that aim to essentially reduce the execution latency of load instructions can potentially bring further performance improvements. To that extent, in the next Chapter we revisit the technique of load-address prediction in the context of data speculation with the purpose of radically diminishing load-execution latency.

CHAPTER 5

REDUCING LOAD EXECUTION LATENCY RADICALLY THROUGH LOAD ADDRESS PREDICTION

5.1 Introduction & Motivations

Load instructions are among the instructions that highly contribute in a program's execution time for several reasons. First, load execution latency is not a fixed parameter, as it depends on the operation of cache hierarchy, the pipeline depth and the inherent data dependencies of the instruction. Modern replacement policies in cache organization reduce considerably the memory access time by allowing loads to find their data in high cache-levels. However, while contemporary processor designs keep adopting deeper pipelines as a way to extract more ILP, load delay continues to remain an issue. Furthermore, load instructions are often found at the beginning of data dependency chains in the critical path of execution, since data are typically loaded from memory as the operands of other instructions. Consequently, long-latency loads are a menace to the overall instruction throughput of a processor's pipeline.

Value prediction (VP) mechanisms can potentially mitigate the effect of load execution delay when loaded values are predictable. Even though modern value predictors excel in capturing the predictability of instruction results, previous research [GG97b; SCD17; SH19] has revealed that memory addresses of load instructions are often more predictable than their loaded values. We illustrate this aspect in Figure 5.1, characterizing *load-value predictability* and *load-address predictability* of our benchmark set (discussed in Section 5.6.2). In our simulations, we selectively use one implementation of the state-of-the-art VTAGE [PS14b] value predictor to perform load address prediction and another one to perform load-value prediction. Predicted values/addresses are not used for data-speculative execution in the pipeline; we only report the fraction of committed loads that predicted either both their value and memory address or only one of them. The results we obtained suggest that for a subset of the SPEC'06/'17 benchmark suites, up to 75% (20% a-mean) of the committed loads could predict only their effective address (i.e. committed with a high confidence prediction of their memory address rather than

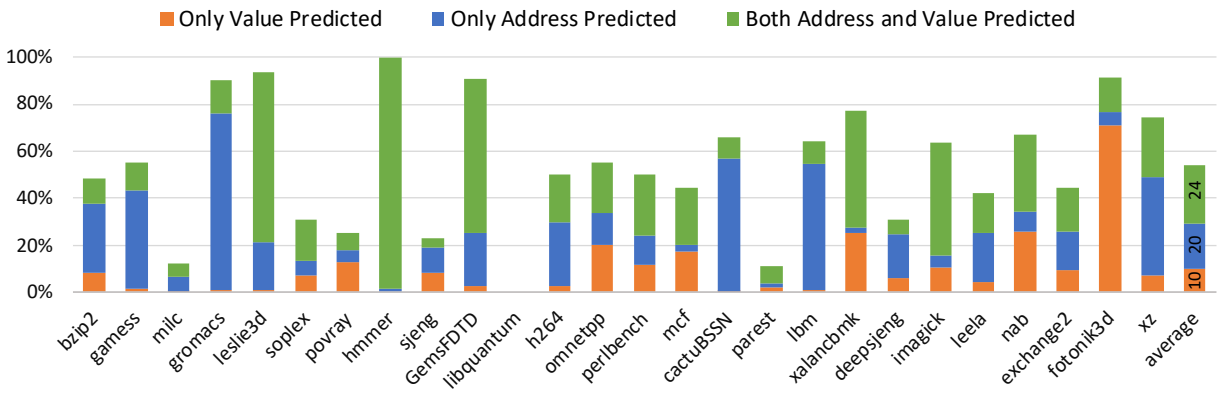


Figure 5.1 – Fraction of committed load instructions that predicted either both their value and memory address or only one of them by using VTAGE.

their value) versus 10% a-mean that could only predict their value. Inarguably, in these cases, by predicting the load address would allow to anticipate the access to the loaded data, and therefore the contribution of these load instructions to the total execution time could be potentially reduced or even nullified.

However, *load address prediction* (LAP) for data speculation has not gained much attractiveness from the research community as value prediction, mainly because predicting load addresses does not entirely remove the effect of the memory-access delay. In this Chapter, we revisit LAP by capitalizing on recent advances on value prediction techniques. We explore the enhancement of a processor's pipeline with a load-address predictor with the ultimate purpose of executing load instructions ***as early as possible***.

Contemporary processor designs leverage branch prediction in order to speculatively resolve control dependencies early in the frontend. Similarly, by employing LAP, load instructions can start their memory access ahead of time and reduce their overall execution time. When this is possible, load instructions will have practically fetched their value sooner than before. Therefore, like in value prediction, any dependents on the corresponding loaded-data can also execute earlier by using speculative operands. However, while many load-addresses are predictable, their uncontrolled exploitation for early load-execution would result in an explosion of memory-order violations. To eliminate most of them, a *memory dependence filter* has to be used to define the eligibility of a load instruction for early execution.

In typical speculative execution, e.g. branch or value prediction, predicted instructions are executed normally in the pipeline in order to validate their speculative data. In the scheme that we propose, load instructions make use of their predicted memory address in order to initiate their requests to the memory from the early frontend of the pipeline. Therefore, the underlying prediction that requires validation concerns the predicted memory address and not directly the

loaded data. Similarly to established work on VP [PS14b], address validation can be postponed before commit. Also, in the recently proposed EOLE architecture [PS14a] some extra hardware allows the late execution of very confident branches beyond the out-of-order core. Building upon this idea, we propose the use of an equivalent structure to calculate the actual address of an *early load* just before retirement. In that sense, early-executed loads need no issue to the out-of-order engine for further execution. Consequently, *early loads* do *not* occupy a place in the processor's *Instruction Queue* (IQ) as they are *not* sent to the core's scheduler, ultimately avoiding the power hungry and complex out-of-order engine.

In the rest of this Chapter, we first determine the prediction model that we incorporate in our design for load-address prediction by exploring load-address predictability with different prediction schemes. The prediction model we finally choose is derived from the hybrid of VSEP/VTAGE we introduced in Section 4.4.2. We frame the way that load-address prediction can be exploited in a pipeline by enabling very early load accesses. Then, we extensively describe the characteristics of our LAP-based model and we present a comprehensive analysis of its performance before concluding.

5.2 Using the Hybrid VSEP/VTAGE for Load-Address Prediction

Load-address prediction is considered as another form of value prediction; rather than predicting the result of a load instruction, one predicts its effective address. Value prediction exploits the fact that different dynamic executions of the same static instruction will regularly produce a predictable result/value. Similarly, LAP is based on the observation that load instructions will regularly access a predictable memory address. In Chapter 2 we presented prior work on memory-address prediction in general and on LAP in particular. While some studies investigate address predictability and introduce new prediction models [Bek+99; SCD17], majority of them incorporate mechanisms that have already been developed in the context of value prediction, adapted for memory-address speculation [MLO98; GG97a; GG97b; Bla+98]. Generally, since value predictors typically store full 64-bit register values in their value-entries, they can be respectively adapted to store and predict memory addresses.

In this study, we propose to explore the correlation of load-addresses with respect to the global branch history by accordingly using an existing prediction scheme. VTAGE [PS14b] and the prediction model of VSEP that we introduced in the previous Chapter, have established effective ways to leverage the global branch history for the prediction of instructions results. In particular, a combination of VTAGE and VSEP appeared to perform better than its single counterparts by effectively exploiting the total set of value patterns that each scheme can independently capture (see Section 4.4.2). In the motivational experiment of the previous Section, we already evaluated the load-address predictability in our framework by using the model of

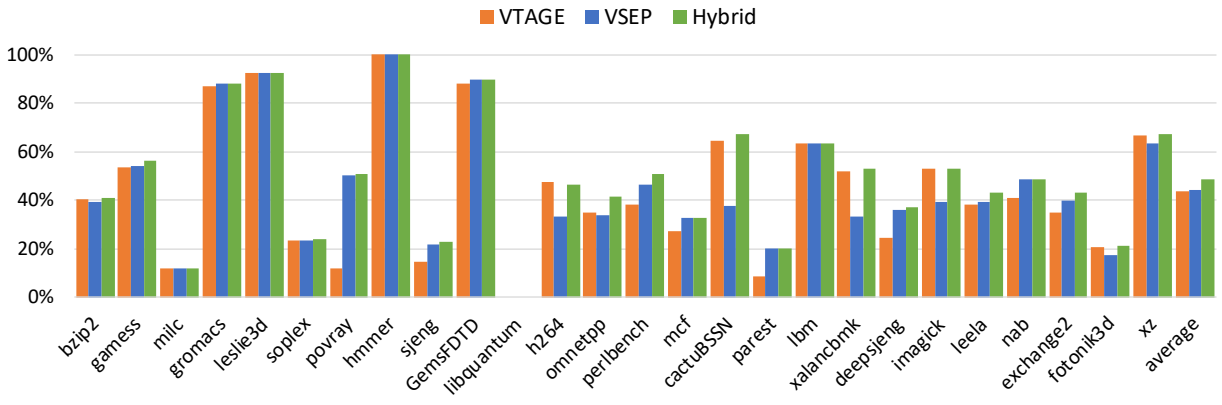


Figure 5.2 – Proportion of committed loads that were *address-predicted* by using the three prediction variants: VTAGE, VSEP and their Hybrid.

VTAGE. We similarly re-purpose the VSEP and the Hybrid VSEP/VTAGE predictor in order to perform load-address prediction and to compare the prediction coverage of all three schemes, i.e. the fraction of *address-predicted*[1] loads. We present our results in Figure 5.2. Note that as before the predictions are not used in the pipeline.

As it appears, VTAGE and VSEP achieve a fairly similar rate of address-predicted loads of around 44% on average, while their compound model reaches 48% a-mean, respectively. Essentially, the Hybrid model obtains in any application either the highest coverage or equal to the highest between VTAGE and VSEP. Therefore, in the following study we choose to perform load-address prediction by adopting the best-performing variant, i.e. the Hybrid model, so that we can fully exploit the existing potential. We basically employ the complete structure and all the particularities of the Hybrid VSEP/VTAGE predictor (internal logic of indexing & updating, equivalent scheme of 3-bit FPCs [RZ06]), as detailed in Section 4.4.2. For clarity, we will be referring to the LAP version of the hybrid model as *Address TAGE & Equality* predictor or simply *ATAGE-EQ*.

5.3 The Use Case of Load-Address Prediction

Load-address prediction has been mainly proposed for reducing the execution latency of load instructions, either by enabling data prefetching [GG97b] or by shortening the delay between fetching a particular load and performing its corresponding load memory access [SVS96; Bla+98; SCD17]. In the first case, the undergone speculation can only implicitly benefit load execution without changing its regular process. Predicted load addresses steer prefetching re-

1. A load instruction is considered *address-predicted* when it retrieves a high-confidence prediction of its memory address from the relevant predictor.

quests and promote data that will be most likely probed by future loads, in higher cache levels, diminishing the anticipated access delay. On the other hand, the second case aims to substantially modify the execution flow of load instructions by allowing them to speculatively perform their memory access earlier than before. In our study, we leverage LAP in the scope of this second approach.

5.4 LAPELE: Load-Address Prediction for Early Load Execution

5.4.1 The Anatomy of Early Load Execution

5.4.1.1 Speculative Memory Accesses

Typically, the overall execution of load instructions consists in two inherently consecutive operations, a "prelude", that of the load-address resolution, and the "main act" of memory access. In other words, loads need first to compute the memory address that they will use in order to probe memory. To do so, after having resolved their data dependencies, loads are issued to a functional unit suitable for memory address computation. Although the action of address generation can be managed by a mainstream arithmetic logic unit (ALU), modern microprocessors employ separate specialized units for this purpose, i.e., address generation units (AGU). This eliminates the contention between memory address arithmetic and regular arithmetic. The computed memory address is then used to send a load request to memory through an available cache port. The corresponding response from memory arrives to the pipeline with a latency that varies, based on the cache level that eventually served the request. Its reception finally signifies the end of the load execution.

Load memory access is the part that contributes the most to the load execution latency. Despite its criticality, it can suffer from noticeable scheduling delays, awaiting the memory address computation, especially in the deep execution pipelines that modern processors traditionally adopt. However, the memory probing itself could be seamlessly maintained in parallel with the flow of the load instruction in the pipeline. It follows that if load instructions could potentially define their effective address in advance, they could noticeably accelerate their execution by practically exploiting the pipeline depth.

Load address prediction is the straightforward solution for the preliminary definition of a load's memory address. LAPELE builds upon a pipeline that implements load-address prediction in order to promote the action of load memory access to the early frontend of the pipeline. In particular, ATAGE-EQ is indexed during the *Fetch* stage, as it only requires the instruction PC and the global branch history to search its components. As such, it allows loads with predictable memory addresses to speculatively acquire their effective address in the very early frontend of the pipeline. At that point, these loads can initiate their memory access and therefore, receive

their loaded data prematurely. From now on, we will refer to this category of loads as ***early loads***.

5.4.1.2 Late Address Generation & Execution Validation

Early loads perform a speculative memory access beyond the OoO execution engine and before computing their effective address. In order to make use of any speculatively retrieved data, early loads need to somehow validate their execution outcome. However, some early loads may receive their loaded data before reaching the OoO execution engine, while some others may require more time. An interesting design aspect lies with the further manipulation of early loads in both of these cases. In order to better illustrate our approach, we find it necessary to first describe the validation scheme that we adopt. As we described in Section 2.3.2.2, two different validation techniques can be used: *validation through the loaded value* & *validation through the effective address*.

The first alternative follows the standards of conventional value prediction and therefore, an early load needs to perform two memory accesses, as presented in Figure 5.3. First, an early load will perform its memory access using a prediction of its effective address. If it can obtain its loaded value before the phase of *Rename*, it will communicate it to any dependent instructions through the PRF (resp. an instruction will predict its value). Then, it will be typically executed in the OoO core in order to write its non-speculative result to the PRF and verify the speculative one (if any). Thus, any belated result of the early memory access, that arrive in the pipeline after the load has been issued to the OoO engine, is discarded. Consequently, unlike value prediction, early loads can make use of their speculatively retrieved data only if they can timely retrieve them from memory. A representative example of this scheme can be found in the work of Sheikh et. al [SCD17]. In this study, only short-latency loads that can be served by the highest cache-level benefit from an early memory access. Loads of longer latency, that may have accordingly started an early memory access, simply drop their belated response as they re-execute in the OoO engine.

On the other hand, based on the second approach, early loads will validate their execution correctness by verifying that the predicted memory address used for retrieving the loaded data is correct. Therefore, an early load can be scheduled only for address computation, avoiding the second memory access (see Figure 5.3). Moreover, since there is no need to compare the loaded value of an early load with that of its regular execution in the OoO core, timely memory responses to early memory accesses are not required. Any load that can predict its effective address will be eligible for early execution. However, even if a load instruction has predicted correctly its effective address, its loaded value may be eventually incorrect due to an interleaving store that modified the data of the same memory location (see next Section). Value validation would identify this ordering violation as a value misprediction, but address validation

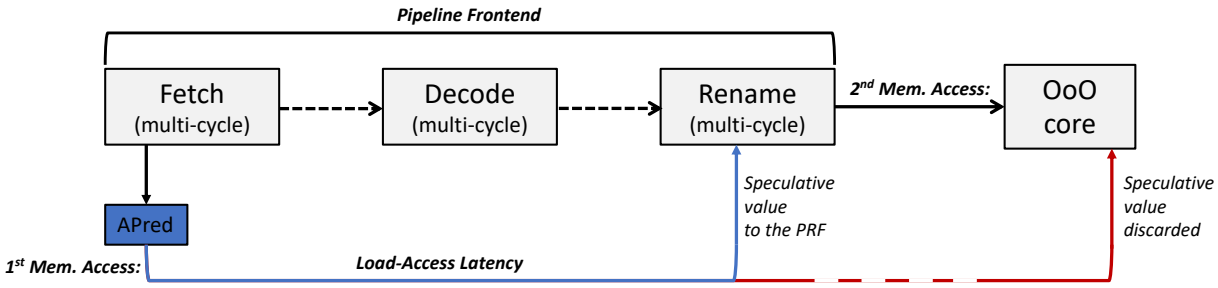


Figure 5.3 – Pipeline execution of an early load when value validation is employed. Speculatively retrieved data are only used in the pipeline when the load-access latency fits in frontend.

can not capture this case. Thus, the underlying memory disambiguation scheme should be explicitly aware of the early execution of a load in order to sustain the imposed memory order. Also, unlike value validation, since the early load is executed only once, in a possible address misprediction the early load itself will have to re-execute. A similar approach was employed by Black et al. [Bla+98], in one of the first implementations of LAP in the late 90's.

According to the above, the chosen validation scheme of early loads can completely change the model's philosophy. The outstanding difference between these two alternatives is that when value validation is employed, long-latency loads can not benefit from an early memory access. In our model, we advocate for a processor design that will employ *early load execution* in a manner agnostic to the underlying memory access delay. As a result, we propose to validate the correctness of early loads through their effective address. It should be noted here that address validation would take place in any case for predictor's training, even if the execution was validated using the loaded value.

Moreover, in order to completely decouple an early load's memory access from further scheduling delays related with its effective address, we propose to delay the computation of its memory address till the *Commit* stage. That is, similarly to the EOLE architecture [PS14a], we assume an additional pre-commit stage where early loads compute their memory address and then validate their speculative memory access. The *Late Address Generation, Validation & Training* (*LAG/VT*) stage implements *commit width* ALUs, similarly to [PS14a]. Except from the early loads, any other load will also validate its predicted address during the *AG/VT* stage in order to update the predictor. That is, like in VP-based designs, we assume commit-time validation and training in order to avoid additional hardware complexity in the OoO core [PS14b].

In general, our validation approach introduces a completely different perspective of load execution. When a load instruction predicts its memory address (i.e. can retrieve a high-confidence prediction), its two principal parts (i.e, address generation and memory access) are performed reversely and according to the program order in the early frontend and in the late backend, respectively. As such, an early load can totally bypass the OoO execution engine. In the following

paragraphs we analyze the underlying issues of this method and the necessary mechanisms that we employ to defeat them.

5.4.2 Memory-Order Issues

5.4.2.1 Ordering Limitations in Early Load Execution

The model of LAP that we propose relates a predicted memory address with an early memory request, introducing the idea of *as-soon-as-possible* load execution beyond the out-of-order core. However, a high-confident and potentially correct prediction can not always be employed for early load execution due to the underlying memory dependencies of the load instruction. Memory dependencies are another form of data dependencies between instructions, similar to register dependencies. They occur when the addresses referenced by a load instruction and a preceding, in program order, store instruction are the same or overlap. To support out-of-order load/store execution, current state-of-the-art processors employ an additional technique, called Memory Dependence Prediction [Mos98; CE98], in order to clarify memory dependencies between loads and stores before their addresses are known. This way, the processor can prevent loads that are predicted dependent on an in-flight store from executing before that store completes, avoiding memory-order violations that lead to costly pipeline flushes.

In the unique case of early loads, the earliness of their execution may uncover memory aliasing with a store that would have normally completed before the correspondent load had started its memory access or even issued to the OoO core. In Figure 5.4 we illustrate this aspect by showing the execution of a load instruction in a pipeline with and without employing LAPELE. We assume that the depicted load/store set operates on total or partially aliasing memory addresses. In normal OoO execution (upper half image), load/store execution does not overlap, but if the same load executes early (lower half image), it will violate the memory order as the previous store is not yet finished. Same phenomenon may also appear in a load that is closer to a prior aliasing store, but its execution is mechanically delayed due to register dependencies. In other words, *early load execution* is more sensitive to ordering violations.

5.4.2.2 Early-style Memory Dependence Speculation

The augmentation of memory hazards with early load execution necessitates a mechanism to timely detect memory dependencies before allowing the early execution of a load instruction. A straightforward solution would be to leverage the embedded memory dependence predictor (MDP) of the core to detect load dependencies earlier in the frontend. Conventional memory dependence predictors (MDPs), e.g. Store Sets [CE98], track the stores upon which the load has ever depended by monitoring memory violations that occurred when a load was executed without any restriction. MDPs are tightly coupled with the processor's backend and operate at

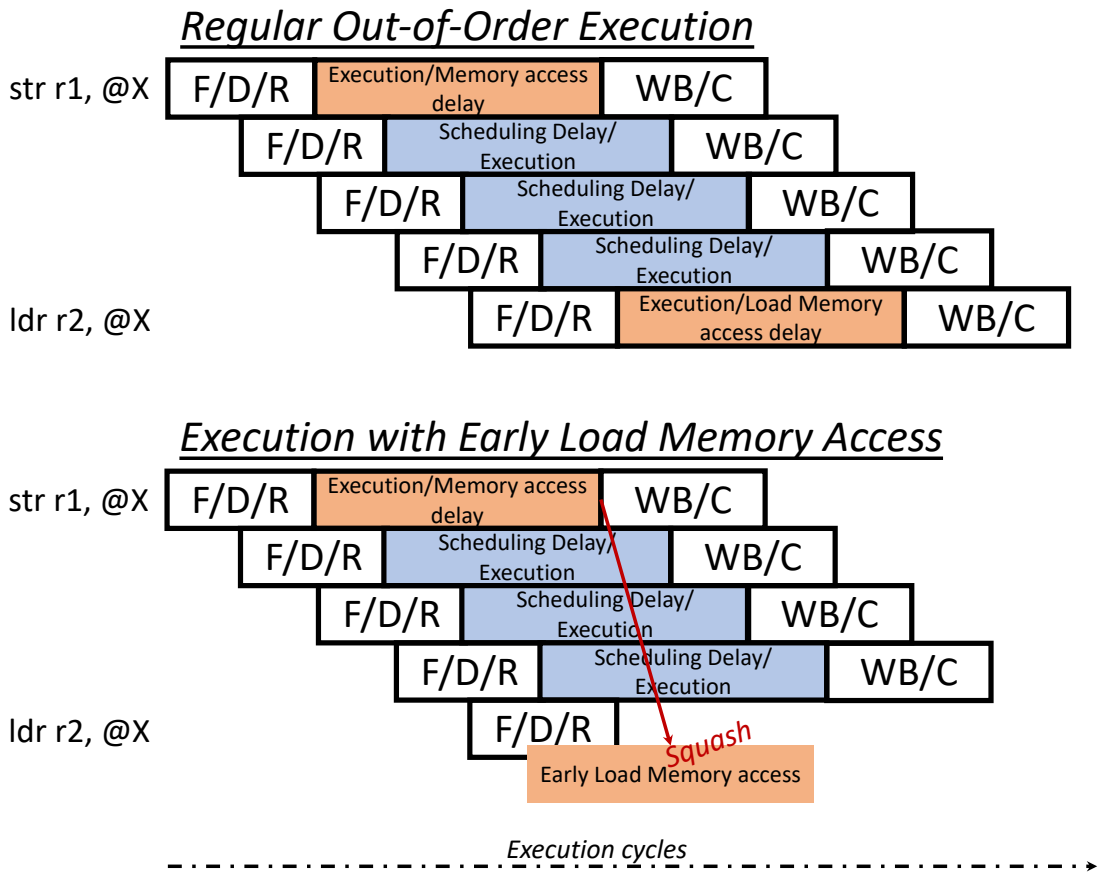


Figure 5.4 – Early memory access uncovering aliasing with older stores.

the scheduler level allowing loads to issue *as-early-as-possible*. However, since the *as-early-as-possible* execution of loads in the early frontend may uncover additional memory conflicts, using the core's MDP for early detection of memory dependencies would fail to capture them all. In addition, it could have negative effects in regular out-of-order execution, if MDP was trained with memory-order violations that occur only with early load execution.

To that extend, we propose the use of an additional mechanism, that we call Memory Dependence Filter (MDF), along with LAP, in order to detect memory dependencies of loads early in the pipeline. Loads that are found dependent on previous in-flight stores are not authorised to use their (potentially) predicted memory address for early execution and they are normally issued to the OoO core. Certainly, It would not be profitable for a dependent load to stall in the frontend till its memory dependencies are resolved, because it would insert an additional delay and could miss the chance to exploit a possible *store-load-forwarding* (STLF) event. Briefly, MDF, as its name implies, *filters out* loads that their early execution would violate the program's memory order.

Table 5.1 – Layout of the TAGE-like Distance Predictor.

Component	#Entries	Tag Width (bits)
Base	4096	5
Tagged 1	512	10
Tagged 2	512	10
Tagged 3	256	11
Tagged 4	128	11
Tagged 5	128	12

In more detail, MDF is a variation of the Instruction Distance Predictor introduced in [PS16] to enable Speculative Memory Bypassing (SMB). This machinery is a perfect match of the early memory dependency indicator that is necessary in LAPELE, as it can discover load/store dependencies at the early frontend by only using the instruction's PC. To do so, MDF consists in two main components: the Data Dependency Table (DDT), which identifies pairs of store-load instructions, and a TAGE-like [SM06] predictor that provides load instructions with a distance to the latest store referencing the same memory address. The scheme works as follows:

— Before retirement, store instructions use their effective address to index the DDT and register their Commit Sequence Number (CSN), which is a global counter that is incremented every time an instruction commits.

— Accordingly, when load instructions commit, they similarly access the DDT to discover the CSN of the latest store that allegedly wrote on the same memory address. The distance (in number of instructions) is then calculated by subtracting the CSN found in the DDT from the currently global CSN.

— The identified distance (if any) is then transferred to the associated predictor that will be accessed by any subsequent instance of the load instruction in the frontend.

The TAGE-like predictor we consider is structured on par with the one used in [PS16]. It consists in one direct mapped table (the base component) and five partially tagged components that are indexed using a hash of the instruction's PC with the path history and the global branch history. More specifically, 2, 5, 11, 27 and 64 bits of global branch history are respectively used for the index-hash generation of each tagged table. Table 5.1 summarizes the size parameters of the predictor. As the monitored distance associates a load with an in-flight store, it cannot be greater than the ROB size augmented with in-flight instructions from Fetch to Dispatch. In our framework (discussed in Section 5.6), this length is less than 256 instructions and therefore 8-bit distance fields in the predictor's components are sufficient. Included also a 4-bit confidence counter per entry, the total storage overhead of the predictor reaches the 12KB. Moreover, the DDT we employ is a 4-way set associative table of 1024 entries (256 entries per set), each of which holds the relevant CSN (64 bits) and a 5-bit tag. Altogether, DDT accounts for 4KB of

storage space. For accessing DDT, we fold (XOR) twice the relevant effective address upon itself to obtain the 7-bit index and the 5-bit tag that are needed.

By using this scheme, loads can retrieve from the internal predictor the distance to the closest prior store instruction that operates on the same memory address. The instruction window is then searched and if a store instruction is found indeed at that distance, the load instruction is marked as early memory dependent and is excluded from early execution. Since we want to gauge performance of LAPELE, MDF is only used in the pipeline frontend to regulate early loads, even though it could be potentially tuned to replace the embedded MDP of the OoO core or to enhance it. Hence, in our design we still use a discrete MDP, namely Store Sets [CE98].

5.4.2.3 Guarantee of Memory Disambiguation

In regular OoO execution, memory disambiguation is enabled though the Load-Store Queue (LSQ). After a store has executed and has computed its effective address, it searches the LQ to find any in-flight prior load that had already accessed an overlapping memory address. In this case, a *store-load* or memory-order violation is identified and recovery actions are required, as the load instruction has eventually fetched stale data.

Similarly, a memory-order violation may happen in early load execution, i.e., an early load to have erroneously performed its memory access ahead of its store producer. As we have already pointed out in Section 5.4.1.2, these violations can not be captured through the validation of the predicted memory address. Since early loads do not induce a second repairing execution, the memory disambiguation mechanism should be accordingly adapted to consider early loads as well. To accomplish that, early loads are normally registered in the LQ, even though they are not issued to the OoO core. Yet, an early load that initiated its memory access in the early frontend, may finally reach the LQ several cycles after. In this interim period, potential violations can not be captured by the typical sanity control in the LQ. To revert this event and completely guarantee memory-order correctness, we propose the use of an intermediate FIFO queue, called Early Load Queue (ELQ). Early loads will be registered in the ELQ by the time they send their early request and they will be removed when they finally reach the LQ. With this scheme, executed store instructions will first scan the LQ and then the ELQ in order to detect violations.

Interestingly, readers will notice that since early loads compute their architectural effective address before commit, they pass to the LQ with their speculative address. Accordingly, preceding stores that check for violations will verify the memory order by using this speculative address. Even though this may seem out of the norm, it is mechanically compatible with an address-based memory disambiguation scheme, i.e, first predict if loads depend on an earlier store (see memory dependence filter of the previous Section) and then compare the addresses of loads and stores to determine whether memory locations overlap (prediction validation). Because early loads perform only one memory access using their predicted effective address, the

same memory address should be used to validate store-load order, regardless if it may finally prove to be wrong when validated at commit.

5.4.3 Cache-port Contention

In order to initiate their memory access at the early frontend, early loads need to use an available cache port in order to send their memory request. Processor designs conventionally incorporate a fixed number of ports that load instructions can use. For instance, *Skylake*, one of the latest Intel microarchitectures, has two load ports, which means that two memory reads (loads) can be sustained each cycle. Load ports are managed by the scheduler, which decides whether a load instruction is ready to start its memory access.

In our model, we assume that load ports can be equally accessed from the early frontend of the pipeline so that early loads can dispatch their memory requests. However, we do not allow potential early loads to take over all the available capacity. On the contrary, we give priority to the loads that are ready to execute in the OoO engine by equating each cycle the number of feasible early loads to the left over bandwidth(i.e., *#Early_Loads = #Read_Ports - #Ready_Loads*). Interrupting the normal execution of loads in the out-of-order core can have diminishing returns, if the corresponding loads are in the execution critical path. Also, when aggressive speculative scheduling is considered, resource contention that causes delays may trigger instruction replays that can largely damage performance [Per+15]. Note here that since early loads are offloaded from the OoO core, the memory-accesses balance is (roughly) proportionally distributed between the out-of-order engine and the frontend. In our experimental analysis we saw that load-port availability is not a primary limitation of early loads.

5.4.4 The Early Memory Access Engine

LAPELE introduces a unique way to preemptively launch the memory access of load instructions beyond the out-of-order execution engine. We call the machinery that enables this process, the Early Memory Access Engine (EMAE) (shown later in Figure 5.5).

5.4.4.1 Transmission of Early Memory Requests

EMAE features a FIFO queue, namely *Memory Address Queue* (*MAQ*), of the high confidence predicted load addresses that were not found dependent to any earlier store, based on the MDF. ATAGE-EQ and MDF are *TAGE-like* predictors that are indexed using the program counter and the global branch history. Thus, they are accessed in the first stage of *Fetch* and we assume that they can deliver their predictions before the phase of *Decode*. Primarily, EMAE offers the required connection to the data cache. It is accessed in parallel with the *Decode*

phase and enables the expedition of an early memory request for the first entry of the MAQ, only if there is an available cache port. The port availability is defined based on the number of loads that were scheduled to perform their memory access in the OoO execution engine at the same cycle. For each cycle, we assume that the scheduler of the OoO core has already communicated this information to the EMAE by the end of the previous cycle. This way, EMAE allows an early request to flow towards the cache only on guaranteed availability of a read port. Therefore, the additional circuity that we add for accessing the cache ports, does not impose any complicated selection logic for choosing between the requests flowing from the frontend (EMAE) or the backend (OoO core).

Nonetheless, MAQ entries will not indefinitely wait for an available cache port in order to be processed. On the contrary, we propose to drop a MAQ entry after a certain number of cycles (N) from its allocation, if no port availability is found. The value of N guarantees that if the anticipated data of an early request can reach the pipeline with the lowest delay, the EMAE will be able to receive them before the early load has been dispatched to the LQ. Thus, it is predefined based on the pipeline depth and cache access latency. In our model, an early memory access can start in the first stage of *Decode* and the following latencies are considered: 9-cycle *Decode-To-Issue* latency, 4-cycle *load-to-use* latency on a L1 cache hit. Therefore, N equals to 5. Our experiments showed that further tolerance does not appear as a worthy tradeoff.

5.4.4.2 Propagation of Speculative Values

Unlike value prediction, the speculatively retrieved values of early loads can arrive in the pipeline with various delays. However, as of employing address validation for verifying the execution of early loads, LAPELE allows to short-circuit the execution of both short and long-latency loads. We define long-latency loads as the ones that their memory request will be served by lower cache levels (L2 and L3). In our framework (discussed in Section 5.6.1), when a long-latency load performs an early memory access, its loaded data will reach the pipeline after the load has already entered the LQ. Therefore, these early loads will use the regular way to communicate their results to any dependent instruction, through the bypass network and the PRF ports connected with the LQ. However, this scheme requires a MUX to drive the corresponding memory responses to the LQ instead of the EMAE (that would go by default), which can add some delay to the critical path. Simply though, memory responses that are labeled as *early* and that originate from the L1 will be forwarded to the EMAE, while the others will be normally directed to the LQ.

As it appears, short-latency early loads (i.e., the ones that their data reside in the L1 cache) that can receive their data before reaching the LQ, require another mechanism in the pipeline frontend in order to broadcast their values. Inspired by VP, the common way to accommodate

this is through the physical register file (PRF). To that end, EMAE provides the necessary connection to the PRF. However, in this case, the number of the additional write ports needs to be proportional to the corresponding width of the in-order frontend and can lead to a prohibitive increase of the PRF size/complexity. Hopefully, same issues have already been settled in previous studies on VP. Briefly, Perais and Seznec proposed the use of a banked PRF as an absolute solution for any VP model [PS15b]. As we explained in Section 2.2.3, register file banking allows the allocation of physical registers for the same dispatch group in different banks. According to that, since value predicted instructions are treated in-order and therefore consecutively, their physical registers will be proportionally distributed to the available banks. Such a configuration allows for less extra ports per bank and mitigates the extra area and power cost of the PRF. Capitalizing on these findings, we assume a microarchitecture that features a 4-banked PRF, as early loads are also processed in a consecutive manner in the early frontend.

5.5 Microarchitectural Overview

LAPELE is a technique that leverages load-address prediction in order to allow load instructions to start their memory access *as-soon-as-possible*. Figure 5.5 depicts an overview of the proposed microarchitecture. We assume the following process:

— At *Fetch*, index the ATAGE-EQ predictor to generate high-accuracy load-address predictions. At the same time, use MDF to predict the distance to a previous conflicting store (if any). To communicate the predicted memory address and distance to the needed structures of the pipeline, deposit them in a double-entry FIFO queue, called *Predicted Address & Distance Queue* (PADQ).

— While still in the *Fetch* phase, for loads that have a high-confidence predicted address in the PADQ, identify the in-flight instruction found at a distance same as the one predicted by MDF. If the former is a store instruction, do not promote the corresponding predicted address to the EMAE.

— At *Decode*, using the predicted memory addresses of MAQ in EMAE, opportunistically initiate an early memory access on load-execution "bubbles". For each of them, push an entry to the ELQ, that contains both the memory address and the sequence number of the load instruction.

— At *Rename*, allocate physical registers normally for the early loads. Then, at *Dispatch*, shift them from the ELQ to the LQ and reserve also an entry for them in the Reorder Buffer (ROB). However, do not dispatch them in the Scheduler (i.e. in the IQ) for OoO execution.

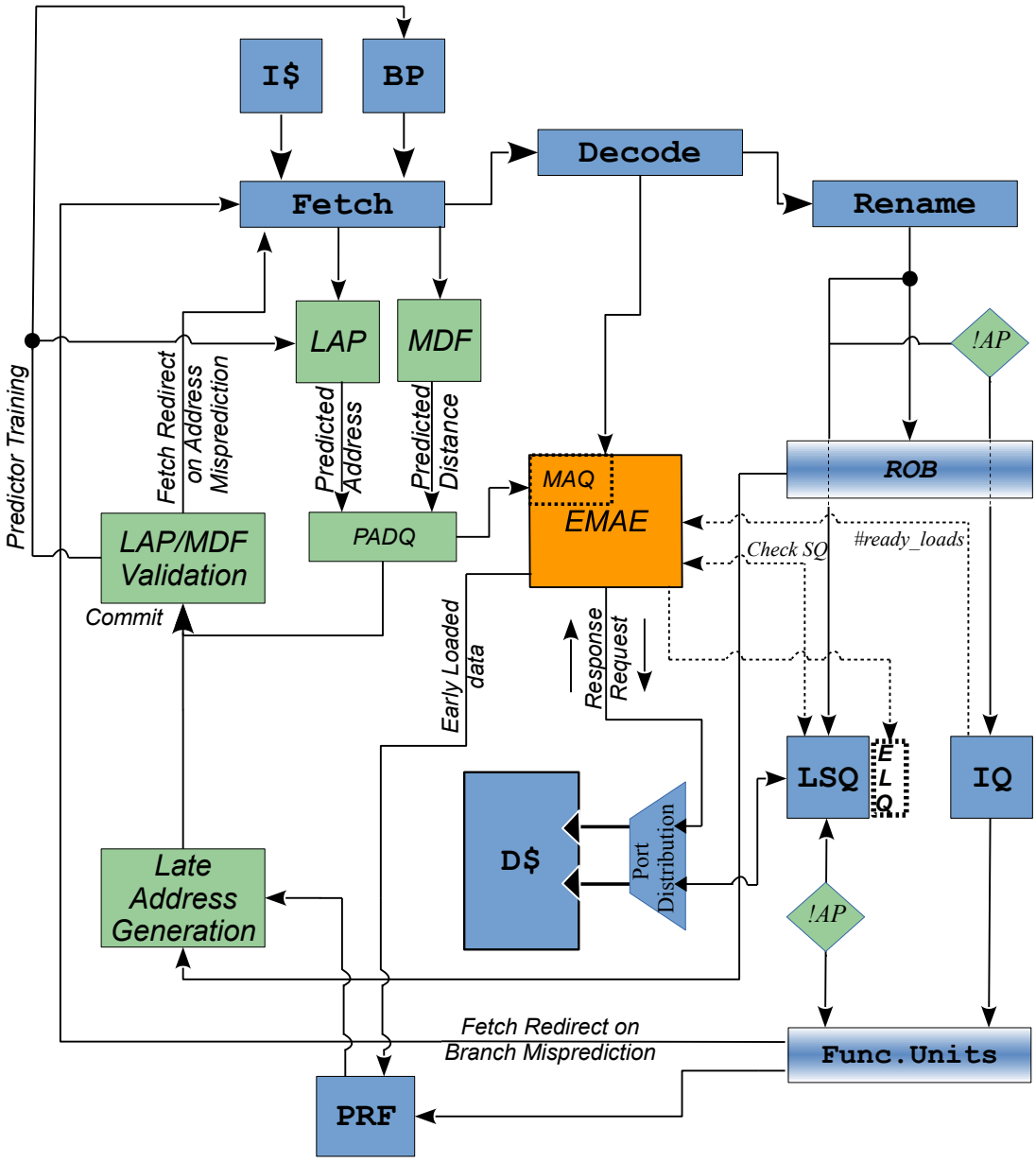


Figure 5.5 – Block diagram of a microarchitecture implementing LAPELE.

— During execution, verify the correctness of the memory-order by considering the ELQ as an extension of the LQ.

— Before Commit, compute the distance by indexing the DDT table and If the load to retire is an early load, compute also its actual address. Use the PADQ to validate the predicted address against the actual one (either freshly computed or not) and the predicted distance against the computed one. Flush the pipeline if an address misprediction occurred for an

early load.

5.6 Evaluation Methodology

5.6.1 Simulator

We reuse the simulator configuration presented in Chapter 3 (Table 3.1) implementing the x86_64 ISA. That is, we employ a relatively aggressive 4GHz, 8-issue superscalar baseline with a fetch-to-commit latency of 19 cycles. More specifically, we model a deep frontend (15 cycles) coupled with a fast backend (4 cycles), in order to obtain realistic misprediction penalties. When LAPELE is used, we add a pre-commit stage responsible for late address generation [2] and validation/training: the *LAG/VT*. The LAG/VT stage lengthens the pipeline by one cycle (20 cycles), resulting in an address misprediction penalty of at least 21 cycles, while minimum branch misprediction latency remains unchanged.

The ATAGE-EQ load-address predictor and the MDF distance predictor make a prediction at Fetch time for every load µ-op. To index the predictors, we XOR the PC of the x86 instruction left-shifted by one with the µ-op number inside the x86 instruction. With this mechanism we ensure that more than one load µ-ops of the same macro-op will generate a different predictor index and therefore, they will have their own entry in the predictor. We assume that the predictors can deliver *fetch-width* predictions by the Fetch stage.

5.6.2 Benchmarks

We reuse the set of benchmarks that were used for the evaluation of the VP-schemes in Chapter 4. More specifically, we use 13 applications from the SPEC2K6 [Staa] suite and 13 applications from the SPEC2K17 [Stab] suite, respectively. Table 4.2 summarises all the benchmarks we consider. To get relevant numbers, we identify a region of interest in the benchmark using Simpoints 3.2 [Per+03] and then we simulate the resulting slice in two step: first warm up the processor (caches, predictors) for 50M instructions, then collect statistics for 100M instructions. Presented average results are calculated as the arithmetic average, with the exception of the relative speedup, whose average is computed as the geometric mean.

5.7 Experimental Results and Analysis

In this section, we demonstrate the performance of LAPELE. We do not explicitly compare LAPELE with previous proposals that adopted a similar use case of LAP, but practically aimed

2. Late address generation takes place only for early loads that skipped the out-of-order execution engine (see Section 5.4.1.2).

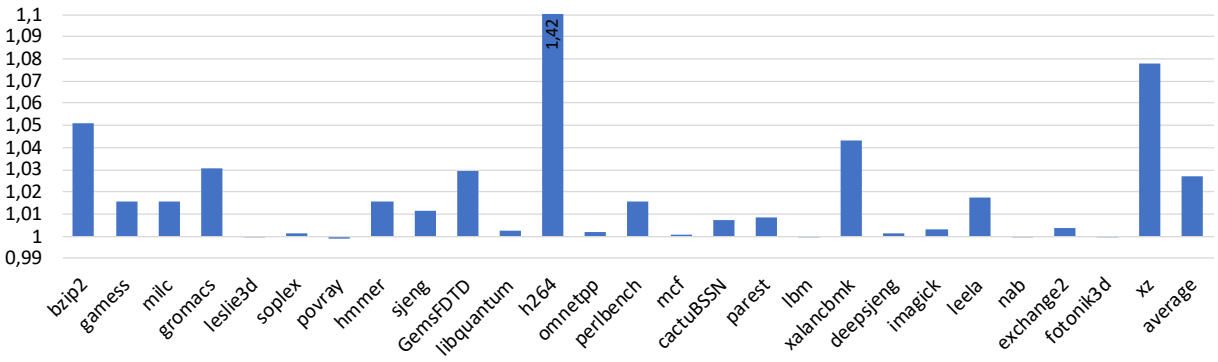


Figure 5.6 – Speedup over the baseline brought with LAPELE.

to device new prediction models (mainly back in 90's). Firstly, as in our study LAP is enabled by the ATAGE-EQ predictor that faithfully follows the operation of Hybrid VSEP/VTAGE (see Section 4.4.2), we do not introduce a new prediction scheme. Thus, it is not relevant to compare ATAGE-EQ with prior address-prediction models, since its model has been evaluated already in the scope of value prediction. Secondly, our mechanism is built with respect to the structural characteristics of a modern superscalar processor, while related prior work would mainly assume infinite resources when needed (e.g. extra read ports in the data cache dedicated for early load execution in [Bla+98]) in order to capture the potential of LAP. However, in order to evaluate the obtained speedup of LAPELE, we compare it with the performance gain that conventional value prediction can deliver when considering only load instructions. Therefore, in our experiments we first solely analyze the performance of LAPELE and stress the importance of "filtering". Then, we estimate the weight of the relevant obtained speedups by examining them together with load-value prediction. Also, we compare LAPELE with an equivalent scheme that assumes value rather than address validation, like the recently proposed DLVP [SCD17].

5.7.1 Performance Analysis

5.7.1.1 Speedup

Figure 5.6 illustrates the performance benefit of LAPELE by reporting the relative IPC of a processor implementing our model normalized to the baseline. A few benchmarks present interesting potential e.g., *bzip2, gromacs, GemsFDTD, h264, xalancbmk* and *xz*, some a more moderate potential e.g., *gamess, milc, hmmer, sjeng, perlbench* and *leela*, and some others relatively low potential. The rest applications perform roughly on-par with the baseline. As it appears, our scheme can effectively exploit the existing monotony in recurrent load accesses in order to mitigate the execution delay of load instructions.

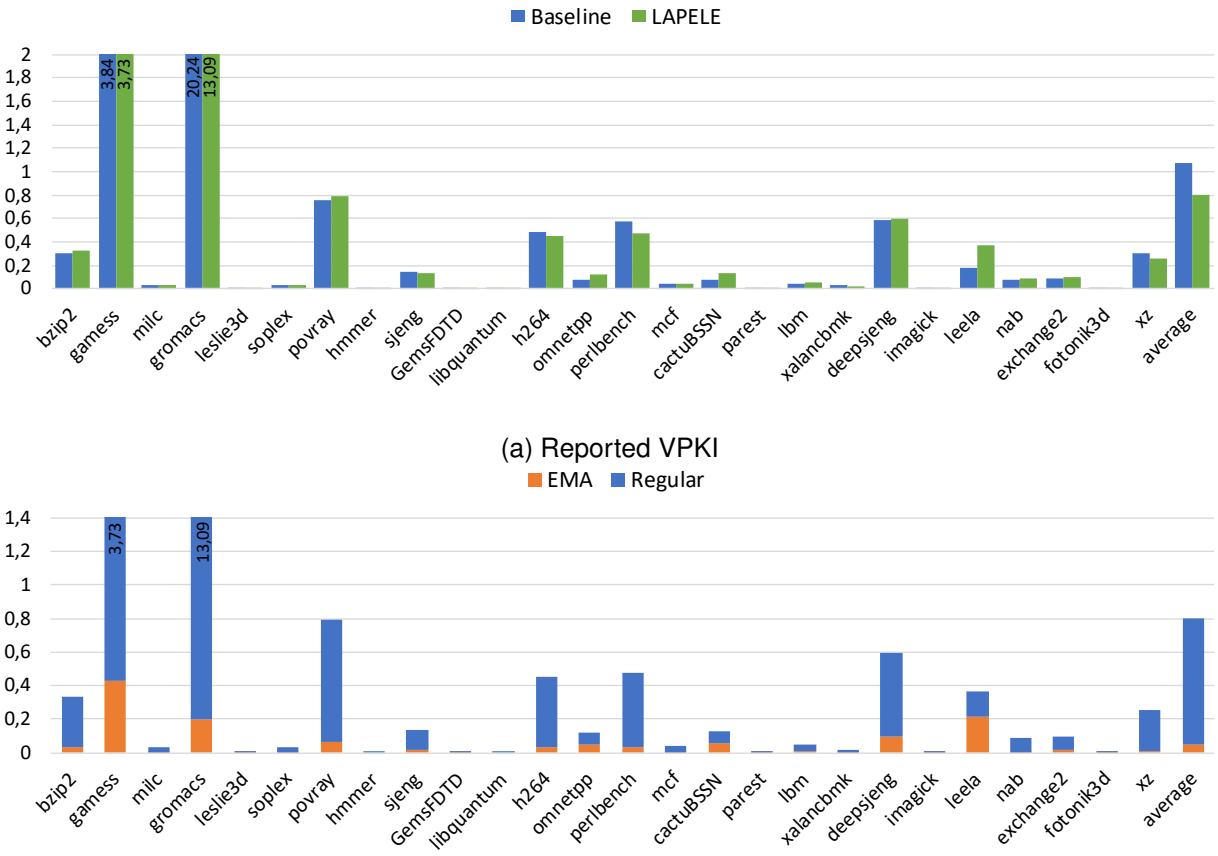
5.7.1.2 Memory Order Violations

By virtue of using FPCs in our ATAGE-EQ predictor, load-address prediction accuracy is kept in very high levels of more than 99,5%. However, as we highlighted in Section 5.4.2.2, by initiating load accesses speculatively by the early frontend, the amount of memory-order violations may increase prohibitively, if no specific care is taken. In the experiments we conducted, the uncontrollable load memory access on any high-confidence predicted address can induce up to 1500x memory-order violations. As such, not only the potential benefit of early load execution is totally vanished, but also tremendous slowdowns of more than 50% are encountered. Hopefully, with MDF, our model can overcome this burden by effectively discriminating on the loads that can proceed in early execution, avoiding the prohibitive increase of memory-order violations. In order to illustrate the efficiency of MDF's "*filtering*", we analyze our model's attitude to erroneous load execution. To do so, we carefully monitor the overall number of committed [3] memory-order violations during execution both in LAPELE and the baseline. Then, in order to better express each variant's behavior, we reduce the collected numbers for each application to *violations per kilo instructions* or *VPKI*. Figure 5.7 presents all the experimental measurements we conducted.

More specifically, Figure 5.7a contrasts the VPKI of our scheme with that of the baseline. Additionally, Figure 5.7b shows the breakdown of LAPELE's VPKI according to the origin of the observed violations, i.e., either from the pipeline frontend due to an early memory access (*EMA VPKI*) or from the regular load execution in the OoO core (*Regular VPKI*). As our exploration shows, our scheme exposes similar or lower VPKI in most of the examined applications, with some exceptions where a slight increase is observed e.g., *bzip2, povray, omnetpp, cactuBSSN* and *deepsjeng*, and the case of *leela* that the reported VPKI doubles. However, even in this event, the absolute value of VPKI remains in reasonable levels, and based on Figure 5.6, around 2% of speedup is observed in *leela*. Subsequently, Figure 5.7b verifies (as expected) that early loads are not the principal contributor of committed memory-order violations. In reality, since LAPELE's VPKI in any single benchmark is in the same range with the baseline, memory-order violations are accordingly balanced, originating either from the early load execution or from the OoO regular execution, respectively.

From the above exploration we can infer that *"filtering"* with MDF allows to uncover the potential of LAP for early load execution in our model. As it appears, *filtering* is an indispensable part of LAPELE or any other scheme that adopts similar use case of LAP (i.e. for early load memory accesses). Interestingly, readers will notice that, in practice, LAPELE reduces the average VPKI (around 25% less than the baseline), even though as we mentioned in Section 5.4.2.2, MDF predictions are not promoted to the OoO execution engine that possesses a

3. Events of memory order violations may be eventually more. Committed memory violations refer to the ones that the relevant load reached Commit phase and triggered a pipeline flush.



(a) Reported VPKI

(b) Breakdown of LAPELE's VPKI

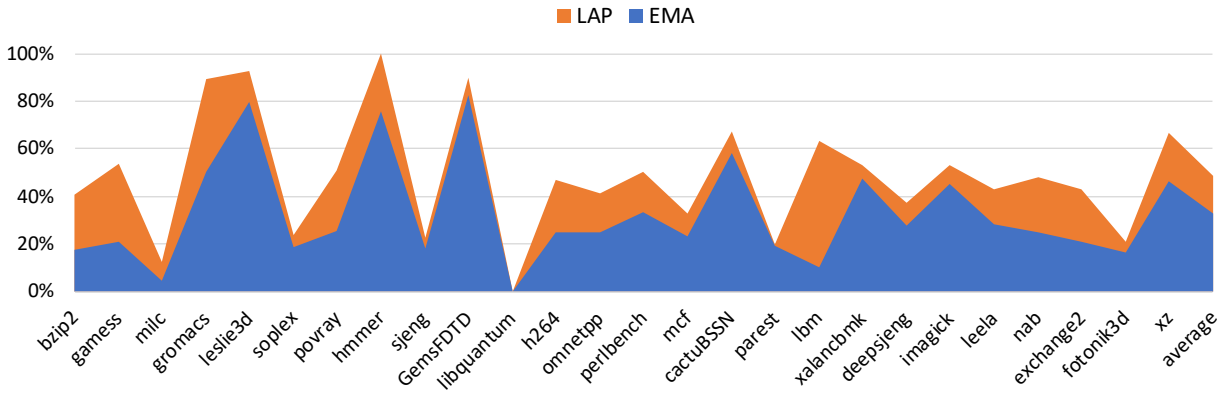
Figure 5.7 – Committed memory-order violations per kilo instructions (VPKI).

separate MDP. Apparently, this outcome is a normal aftermath of the efficient filtering of early loads. That is, as early loads are completely offloaded from the OoO core, the pressure in the integrated MDP is reduced, with the non-predictable or filtered loads to be able to better exploit MDP's allocated resources, e.g, less aliasing or less frequent resetting on SSIT/LFST tables of Store Sets[CE98] [4]. Therefore, more memory-order violations are avoided also in the OoO execution engine, reducing the overall reported amount.

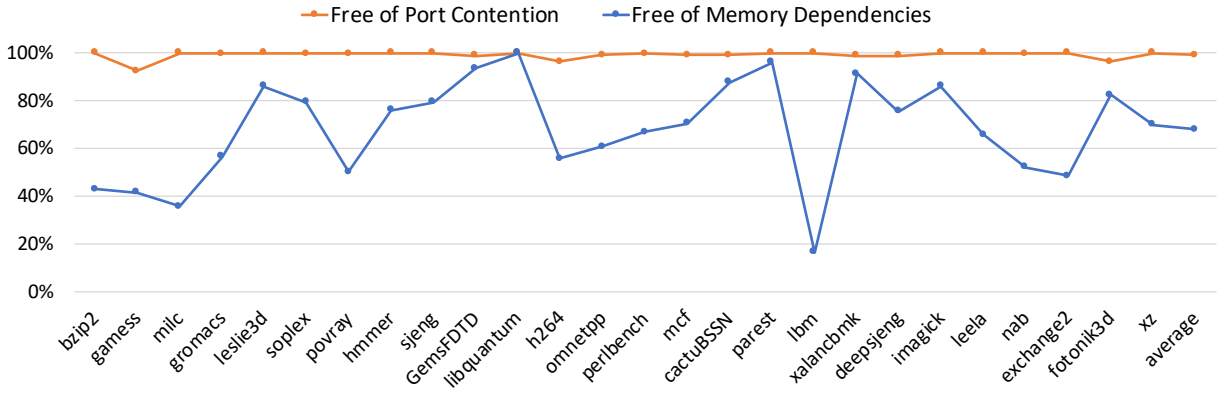
5.7.1.3 Existing Potential and Limitations

In the general case, the potential of a model that leverages a load-address predictor like ATAGE-EQ, depends on the achieved prediction coverage, i.e., the portion of loads that committed with a high-confidence prediction. However, in LAPELE, a high-confidence predicted memory address may not be used in the pipeline either due to memory dependencies or due to

4. Store Sets are cleared every 30K accesses of loads/stores in our implementation.



(a) Proportion of committed loads that predicted their address versus proportion of committed loads that performed an early memory access using their predicted address.



(b) Limitations regarding early load execution.

Figure 5.8 – Coverage and limitations of LAPELE.

structural limitations (cache port contention). Consequently, the prediction coverage does not completely represent the amount of loads that can benefit from LAP and therefore the anticipated performance benefit. In order to show the potential of our model, we have profiled both the proportion of address-predicted loads and the fraction of early memory accesses (EMA). That is, we define *LAP coverage* as the fraction of committed loads that acquired a high-confidence prediction of their memory address (i.e. *address-predicted* loads), while we refer to the fraction of committed early loads as *EMA coverage*.

As illustrated in Figure 5.8a, out of all the committed loads, 48% a-mean were address-predicted and around 32% a-mean performed an early memory access. In other words, even if LAP coverage reaches high levels, the imposed restrictions may limit (noticeably in some cases) the eligibility of early loads. For instance, in *lbm* there is a 53% gap between address-predicted and early loads. Nevertheless, note that high EMA coverage does not necessarily correlate with high performance, e.g. *cactuBSSN* exhibits around 60% of EMA coverage but

only marginal speedup, since not all the address-predicted loads are in the critical execution path in order to speedup the overall execution.

Furthermore, address-predicted loads that were not filtered-out by MDF may not eventually make use of their predicted memory address due to cache-port conflicts, in the way that we described in Section 5.4.3. For a further analysis of LAPELE's potential, Figure 5.8b presents the relevant amount of address-predicted loads that had no cache-port conflict or memory dependence when initiating their early memory access. As it appears, on average 99% of the address-predicted loads that are not found memory dependent are able to occupy an available read port for accessing the data cache. Therefore, cache-port availability is not a primary limiting factor of early loads. On the other hand, memory ordering is the major bottleneck in most of the cases where high coverage of LAP does not end up in a large number of early loads. Our experiments show that on average 67% of the address-predicted loads are not memory dependent, which justifies the gap between 48% of address-predicted loads and 32% of early loads in Figure 5.8a.

5.7.2 Evaluation Regarding to Load-Value Prediction

5.7.2.1 Load-Address vs Load-Value Validation

Our intention is to compare LAPELE with a similar scheme that uses value instead of address validation for speculative load memory accesses. To do so, we found the recently proposed DLVP mechanism [SCD17] to be a perfect match for this purpose. To get relevant numbers, in our DLVP-like variation we still use the same load-address predictor ATAGE-EQ and we *filter* predictions using MDF. The essential differences with LAPELE are:

1. Early loads are followed by a regular (corrective) execution in the OoO execution engine in order to validate any speculatively retrieved data.
2. Based on our processor parameters (see Section 5.6.1), only early loads that are served from the L1 cache can write their loaded data to the PRF.

In a nutshell, eligible loads initiate their early memory access during the phase of *Decode* and if they retrieve the requested data by the *Rename* phase, they communicate them to the PRF. Thereafter, they normally execute in the OoO core to validate the speculated data and trigger a pipeline flush in a value mismatch. Note that in this case ELQ (as discussed in Section 5.4.2.3) is not necessary for memory disambiguation, since value-validation captures also potential memory-order violations induced by early loads.

The experiments we conducted showed that our DLVP-like adaptation can only achieve marginal speedups with respect to the baseline. In reality, the embodiment of value validation inherently limits (see Section 5.4.1.2) the range of loads that can benefit from an early memory

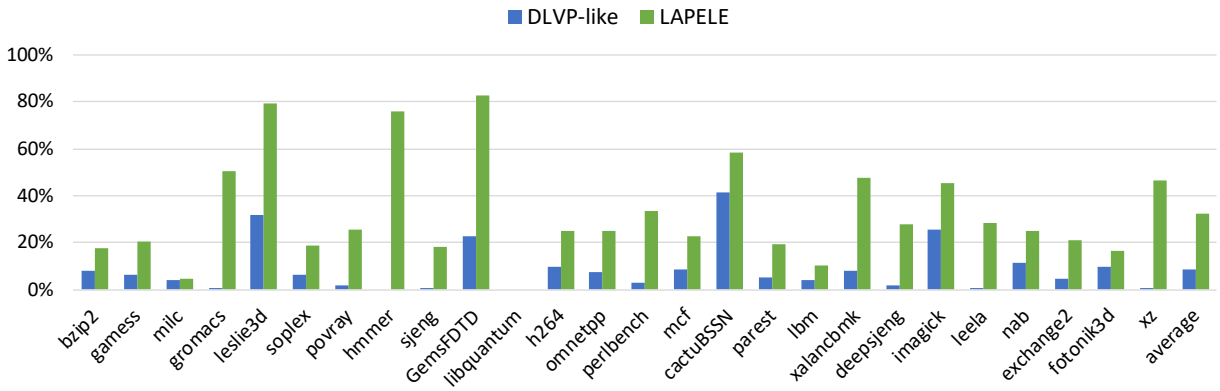


Figure 5.9 – Fraction of load instructions that committed with speculatively-loaded data in LAPELE and in a DLVP-like mechanism that employs value validation for early load memory accesses.

access and therefore, reduces the potential performance gain. For the sake of comparison, we compare in Figure 5.9 the coverage of loads that committed with speculatively-retrieved data after an early memory access in LAPELE and in the DLVP-like scheme. As reported, the latter can achieve less than one third of our model's coverage (roughly 9%). We highlight here that readers should not assume that we disregard the meaningful speedups that Sheikh et al. have reported in their study [SCD17]. Note that the DLVP-like model we recruit is an abstract adaptation and not an actual evaluation of the DLVP mechanism, as originally described by the authors. Also, in our evaluation we use the x86_64 ISA versus the ARMv7/8 ISA used in the original DLVP [SCD17].

5.7.2.2 Early Load Execution vs Prediction of Loaded Values

As we aim to evaluate our scheme with the speedups that a novel technique like value prediction can achieve, we augment our processor's pipeline also with the VSEP/VTAGE hybrid value predictor (as described in Section 4.4.2), but we tune it to explicitly consider only load instructions. Typically, predicted values of load instructions are written to the PRF at *Rename* only in high confidence and validation/training takes place before *Commit*. Figure 5.10 reports our simulation results. In particular, Figure 5.10a shows the relative IPC of the two variants normalized to the baseline and Figure 5.10b shows their prediction coverage, i.e., the fraction of *early loads* for LAPELE and the fraction of value-predicted loads for VTAGE for load-value prediction respectively.

Our evaluation indicates that LAPELE benefits two more benchmarks with respect to load-value prediction, namely, *gromacs* and *milc*, and also outperforms the already high speedup delivered by load-value prediction in *h264*. This superiority is accompanied by also higher cov-

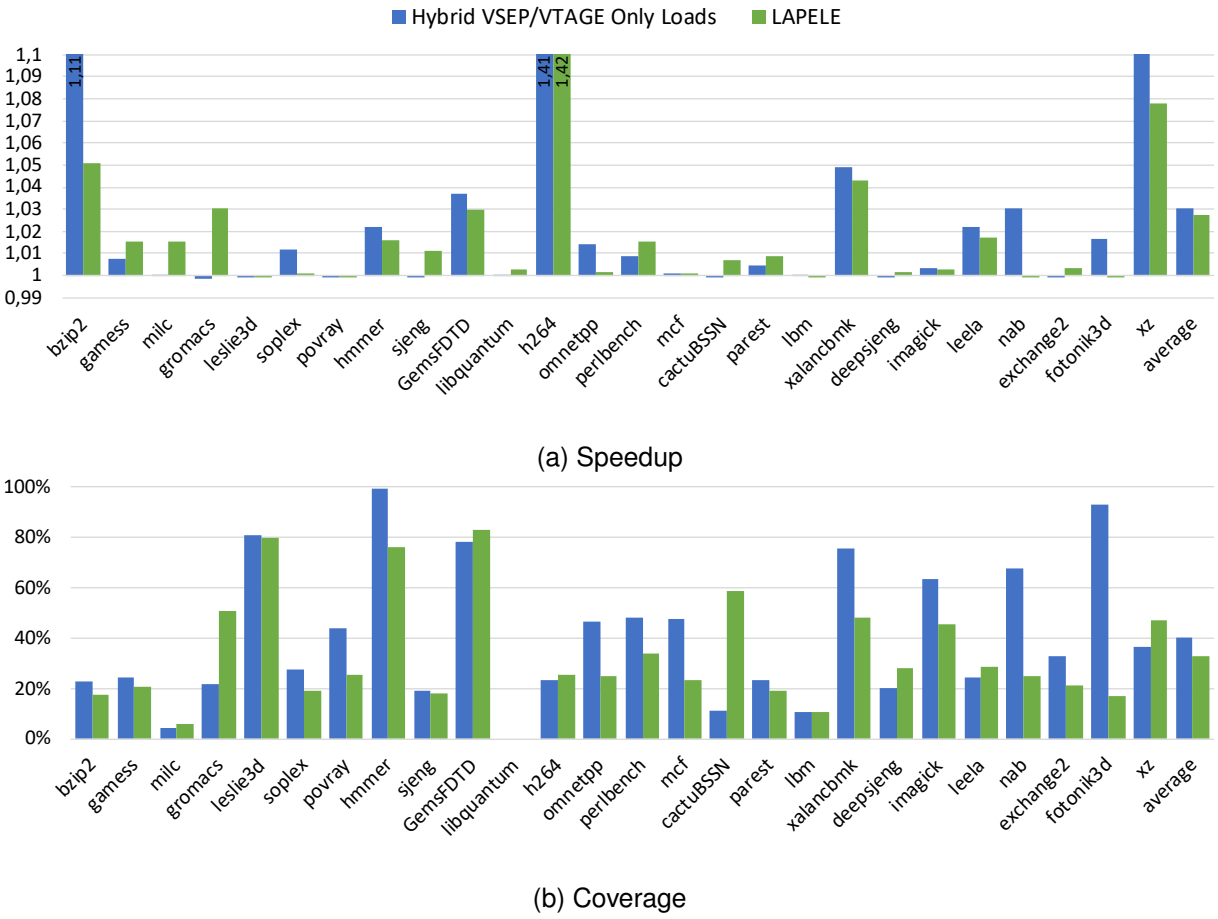


Figure 5.10 – Speedup over baseline and coverage of LAPELE and Hybrid VSEP/VTAGE for load-value prediction.

erage (even marginal) of LAPELE for the same benchmarks. In rest of the applications, the two schemes achieve speedup in close range, except from *bzip2, omnetpp, nab, fotonik3d* and *xz*, where load-value prediction clearly outweighs our model. The average coverage of load-value prediction is 40% versus 32% for our model, and their average speedup is 3% and 2,8% respectively. However, as we have already stated, high coverage does not necessarily lead to proportionally high speedup. For instance, in *perlbench*, LAPELE achieves lower coverage but obtains slightly higher speedup, while in *xz*, LAPELE's coverage is higher but its speedup is lower.

Altogether, the use case of load-address prediction that our scheme proposes can interestingly achieve comparable performance with the typically powerful load-value prediction. In our set of applications, Hybrid VSEP/VTAGE for load-value prediction does not plainly outperform LAPELE, although it is generally considered unlikely that early memory accesses can improve performance more than the direct prediction of load values. However, recall here that

in LAPELE, early loads bypass the OoO execution engine and need their source operands to compute their effective address only when these are ready by default, i.e. at commit-time when any previous instruction that could define some required data has already finished execution. Thus, early loads are totally discharged from any scheduling delays. In the worst-case scenario, the memory-access delay of an early load will be that long that its loaded data will not be delivered before the stage of *Dispatch*. In that sense, the performance gain will be lower because the stalling time of dependents will be reduced but not eliminated completely. Still, latency-wise speaking, early loads will finish execution earlier than previously possible.

On the other hand, in load-value prediction, a predicted load can radically hide its execution latency from dependents by providing them a speculative value before reaching *Dispatch*. Nonetheless, it then has to normally undergo any scheduling and pipelining delays in order to regularly execute in the OoO core and retrieve its actual value. Therefore, in the unfortunate situation that its source operands are not timely available, a predicted load instruction may apparently increase instruction throughput in the OoO execution engine (*instructions executed per cycle*) but not totally in the processor (*instructions committed per cycle*). In reality, this is the absolute scenario that justifies potential performance supremacy of LAPELE over load-value prediction, when both the result/value and the memory address of the same load instruction can be predicted.

5.8 Conclusion

In this Chapter, we re-considered *Load-Address Prediction* for data speculation in order to radically reduce the load-execution latency. We first showed that load-addresses are often more predictable by considering the same prediction scheme for predicting load-addresses and load-values. Then, by leveraging existing value-prediction techniques, we introduced LAPELE, a comprehensive model for the early execution of load instructions through load-address prediction. The uniqueness of LAPELE is that it discharges completely the out-of-order core from the execution of *early loads*, i.e. loads that predicted their memory address with high confidence and performed an early memory access. To do so, LAPELE postpones the memory address computation of early loads in a pre-commit stage and performs validation based on the load-address rather than the loaded data. As such, early loads entail only one speculative memory access, which is launched at the very early frontend, i.e. *as soon as possible*. Although many loads can predict their memory address, the absurd early execution of loads can increase prohibitively the amount of memory-order violations. However, we showed that by using a fairly small auxiliary predictor, most of the conflicting loads are *filtered out*. Eventually, LAPELE achieves to execute early around 32% of the loads and to speed up execution by up to 42% and approximately 3% in average.

Essentially, the potential speedup that can be obtained from the early execution of a load instruction is subject to instructions scheduling. In practice, an *early load* can be more beneficial when its memory access latency is lower than its routing delay from fetch to execution. On the other hand, the highest benefit in load-value prediction appears when the value predictor successfully speculates on the result of loads that their memory request will be served from low cache-levels, i.e. long-latency loads. In general, the objective of value prediction is to *hide* the execution latency of instructions. On the contrary, LAPELE aims to basically *reduce* the load-execution latency by totally decoupling it from scheduling and pipelining delays. As our experiments showed, LAPELE achieves an average speedup in the same range with an equivalent scheme of load-value prediction. Therefore, LAPELE is a mechanism that can successfully overcome the inherent limitations of a load-address prediction scheme (e.g. memory ordering, cache-port contention, execution validation) and to eventually deliver meaningful performance gains.

CONCLUSION

Current flagship processors combine multiple cores on a single die in order to provide high throughput and to enable high-performance computing. However, given the existence of intrinsically sequential programs and also parallel programs that regularly expose non-negligible sequential parts, the multicore mentality can not address alone all performance issues. As expressed by Amdahl's Law [Amd67], even in parallel programs, the sequential portions cap the speedup that can be obtained by employing more cores. Therefore, apart from the good parallel performance that multicore processors can offer, improving the sequential performance of processor cores remains highly relevant even in the multicore era.

In the past, part of the substantial increase of sequential performance came from the escalation of clock speed. However, in the current technology, the further increase of processors frequency is not feasible anymore, as it will lead to a prohibitive increase of power consumption in the chip. This limitation is traditionally dubbed the Power Wall [Bos11]. On the other hand, major performance improvements were also achieved by the improvement of cores design through pipelining, superscalar execution and out-of-order execution. Such microarchitectural techniques are vital to sequential performance, as they enable, by definition, the existence of large instruction windows and therefore, the maintenance of high ILP. Coupled with that, most, if not all, modern processors employ fundamental speculation techniques, such as branch prediction, in order to overcome main execution barriers and exploit the existing ILP.

However, as microarchitectures continue to extract more ILP by employing larger instruction windows, i.e. allowing more instructions to be scheduled for execution concurrently in the core, true data dependencies between instructions remain a major performance bottleneck. In essence, to overcome this obstacle and uncover more ILP, it is required to allow a data-wise speculative execution. *Data-speculative* execution stems from another more advanced level of speculation, mainly enabled by *Value Prediction* (VP) and *Load-Address Prediction* (LAP). The first one refers to the raw prediction of data, i.e. instruction results, while the second one focuses on the prediction of the memory address from where load instructions fetch their results. In both cases, the speculative generation of instruction values allows to harvest more ILP and to reduce the execution latency of instructions. VP and LAP are two developing techniques that are not exhaustively exploited and can therefore undergo further optimizations. In this thesis work, we have proposed certain mechanisms for advanced speculation based on VP and LAP that lead to effectively higher performance improvements.

In particular, we have firstly conducted an inter-ISA exploration of value prediction. Since

the very first introduction of value prediction, there has been a plethora of different predictors that claim particular improvements in an ISA-agnostic manner. However, since VP techniques basically depend on the nature and the behavior of programs code, it is essential to detect the existence of interactions between ISA-intrinsics and value prediction. In our study, we have carefully compared the speedup brought with VP in application binaries compiled symmetrically for the x86_64 and the Aarch64 ISA. Our exploration has shown that, indeed, certain ISA particularities, such as the amount of available registers in the ISA, may impact significantly the performance gain that can be obtained through VP. Moreover, considering that value predictors conventionally assume 64-bit value entries to track execution, vector instructions that produce values of at least 128 bits are not covered, i.e. neither tracked nor predicted. Essentially, predicting vector instructions may result in a fairly large predictor, as it requires to scale the length of value-entries according to the maximum value-width that needs to be predicted (e.g. 128 bits per entry to cover 128-bit instructions). To this respect, in our ISA-level exploration we have also considered the influence of vector optimizations in VP performance. We have discovered that vector instructions are non-negligible in ordinary applications, and when a predictor theoretically employs 128-bit value-entries, it can obtain up to 3x speedup. In other words, vector optimizations put significant barriers to speedup when one is limited to predict only up to 64-bit values.

Second, we have detected an unexplored value pattern, namely *interval equality*, which, however, characterizes a substantial portion of instructions (more than 18% on average). Such value pattern is not effectively captured by previously-proposed predictors that typically keep tracked values (i.e. that will be potentially used for prediction) and their prediction-confidence tightly together in the same predictor entry. In order to remove this limitation, we introduced VSEP predictor, that is based on a binary decision-scheme and highly decouples values from their confidence. VSEP achieves speedup that is in the same range with the previously-proposed VTAGE [PS14b], but by exploiting value patterns not previously captured. In other words, VSEP complements effectively the established way that VP is performed by increasing considerably the fraction of predicted instructions that expose interval equality (additional 15% on average). Essentially, when combined with VTAGE, VSEP improves the obtained speedup by 19% on average. Moreover, with VSEP we can obtain performance on-par with the state-of-the-art VTAGE, by employing around 58% less value-entries (3K value-entries in VSEP versus 7K value-entries in VTAGE) with the same storage budget. In this way, regarding our second observation in our inter-ISA exploration, VSEP can mitigate significantly the cost of considering value prediction for values wider than 64 bits.

Third, and last, we have designed a LAP-based mechanism, namely LAPELE, that performs early load execution beyond the out-of-order execution core. To do so, we have re-purposed our VSEP/VTAGE hybrid predictor in order to perform LAP. With such a predictor we accomplish to

predict the memory addresses of 48% of the committed loads on average. To avoid a possible explosion of memory-order violations we have proposed to use a small-scale auxiliary predictor to "filter" loads that exhibit memory conflicts. Eventually, with this scheme we can execute early 32% a-mean of the committed load instructions by completely offloading them from the out-of-order core. From a performance standpoint, LAPELE achieves an average speedup in the same range with an equally-sized value predictor that is tuned to speculate only on load values. In reality, LAP-based mechanisms can not consistently either reach or surpass the performance of general value prediction. However, with LAPELE, we have shown that LAP can be a necessary catalyst of future complexity-effective larger-window architectures to tolerate memory latency.

Perspectives

The advanced-speculation mechanisms that we have proposed in this dissertation pave the way for several other optimizations that can be explored.

First, a straightforward research direction can be the devise of a fused predictor to combine value together with load-address prediction. In LAPELE we have used the ATAGE-EQ load-address predictor that essentially copies the logic and the structure of our VSEP/VTAGE hybrid value predictor. In that sense, load addresses appear to follow similar patterns with regular instruction results. Thus, both values and load-addresses may be tracked by the same structure. LAP may not be able to outperform VP, but as our experiments illustrate, load-addresses are often more predictable. A fused predictor can be used to track values of regular instructions and values/addresses of loads. Essentially, the filtering mechanism can be used to indicate the loads for which the predictor should update their value rather than their address. Another design choice could also be to use an auxiliary predictor to distinguish static loads as value or address predictable. Then, the fused value/address predictor would accordingly consider either only the value or only the address of dynamic loads.

Second, in LAPELE, MDF predicts the distance to the most recent previous store instruction that accesses the same memory address. This store instruction is the actual producer of the corresponding load. In this case, the value sourced by the store could be forwarded to the load instruction. This scheme may practically increase the store-to-load-forwarding events. As MDF captures the distance to any in-flight store, it may couple a load to a store that has already left the out-of-order engine and would not have the chance to forward its data to the conflicting load. Moreover, one could use an additional buffer in order to exploit even overlapping stores that have committed. That is, even though the predicted distance would exceed the instruction window, it would be used to retrieve the data from the relative position in the buffer.

Third, our proposition for the equality predictor has been inspired by the TAGE branch predictor [SM06]. However, other options could be also considered and other types of equality

predictors could be designed. For instance, other conditional branch predictors such as the family of perceptron-based predictors [JL01; Jim] could also be used. In addition, one could also consider an integration of the ETAGE predictor together with other binary-content predictors of different purpose, such as the already hybrid model of Omnipredictor [PS18] that predicts branches and memory dependencies within the same predictor.

Finally, as we have already mentioned, VSEP is highly independent from the value-width by coupling the equality predictor ETAGE with a table of last-committed values, i.e. LCVT. In essence, LCVT can be tuned to track any needed value or chunks of values and then ETAGE can predict equality/inequality based on that. In this way, another perspective could be the use of equality prediction to dynamically detect phases that correspond to in-memory data movement. This would require to identify equality between the value of a load and the source value of a following store.

BIBLIOGRAPHY

[6M19] Intel 64 and IA-32 Architectures Software Developer's Manual, *in*: *Intel Developer Zone, Order Number: 253665-070US, May* (2019).

[74] « New product applications: Microcomputer product family now includes peripherals and a program development system », *in*: *IEEE Spectrum* 11.*7* (July 1974), pp. 84–88, ISSN: 1939-9340, DOI: 10.1109/MSPEC.1974.6366593.

[AA93] D. Alpert and D. Avnon, « Architecture of the Pentium microprocessor », *in*: *IEEE Micro* 13.*3* (June 1993), pp. 11–21, ISSN: 1937-4143, DOI: 10.1109/40.216745.

[ABB64] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, « Architecture of the IBM System/360 », *in*: *IBM Journal of Research and Development* 8.*2* (Apr. 1964), pp. 87–101, ISSN: 0018-8646, DOI: 10.1147/rd.82.0087.

[Amd67] Gene M Amdahl, « Validity of the single processor approach to achieving large scale computing capabilities », *in*: *Proceedings of the April 18-20, 1967, spring joint computer conference*, ACM, 1967, pp. 483–485.

[Arm19] for Armv8-A architecture profile Arm Architecture Reference Manual Armv8, *in*: *ARM Developer Docs, ARM DDI 0487E.a (ID070919)* (2019).

[AS95] T. M. Austin and G. S. Sohi, « Zero-cycle loads: microarchitecture support for reducing load latency », *in*: *Proc. 28th Annual Int. Symp. Microarchitecture*, Nov. 1995, pp. 82–92, DOI: 10.1109/MICRO.1995.476815.

[Bek+99] Michael Bekerman et al., « Correlated Load-address Predictors », *in*: *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, Atlanta, Georgia, USA: IEEE Computer Society, 1999, pp. 54–63, ISBN: 0-7695-0170-2, DOI: 10.1145/300979.300984, URL: http://dx.doi.org/10.1145/300979.300984.

[Bin+11] Nathan Binkert et al., « The gem5 simulator », *in*: *ACM SIGARCH Computer Architecture News* 39.*2* (2011), pp. 1–7.

[Bla+98] Bryan Black et al., « Load Execution Latency Reduction », *in*: *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, Melbourne, Australia: ACM, 1998, pp. 29–36, ISBN: 0-89791-998-X, DOI: 10.1145/277830.277842, URL: http://doi.acm.org/10.1145/277830.277842.

[Bos11] Pradip Bose, « Power Wall », *in*: *Encyclopedia of Parallel Computing*, ed. by David Padua, Boston, MA: Springer US, 2011, pp. 1593–1608.

[Bre95] Barry B Brey, *The Intel 32-bit microprocessors: 80386, 80486, and Pentium microprocessors*, Prentice Hall, 1995.

[Buc62] Werner Buchholz, *Planning a Computer System: Project Stretch*, New York, NY, USA: McGraw-Hill, Inc., 1962, ISBN: B0000CLCYO.

[CE98] G. Z. Chrysos and J. S. Emer, « Memory dependence prediction using store sets », *in*: *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, July 1998, pp. 142–153, DOI: 10.1109/ISCA.1998.694770.

[Col+85] and Colwell et al., « Instruction Sets and Beyond: Computers, Complexity, and Controversy », *in*: *Computer* 18.*9* (Sept. 1985), pp. 8–19, ISSN: 1558-0814, DOI: 10.1109/MC.1985.1663000.

[Dow+17] J. Doweck et al., « Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake », *in*: *IEEE Micro* 37.*2* (Mar. 2017), pp. 52–62, ISSN: 1937-4143, DOI: 10.1109/MM.2017.38.

[EPS17] Fernando Endo, Arthur Perais, and André Seznec, « On the interactions between value prediction and compiler optimizations in the context of EOLE », *in*: *ACM Transactions on Architecture and Code Optimization* (2017).

[EV93] Richard J Eickemeyer and Stamatis Vassiliadis, « A load-instruction unit for pipelined processors », *in*: *IBM Journal of Research and Development* 37.*4* (1993), pp. 547–564.

[Gab96] Freddy Gabbay, *Speculative execution based on value prediction*, Technion-IIT, Department of Electrical Engineering, 1996.

[GG97a] José González and Antonio González, « Memory address prediction for data speculation », *in*: *Euro-Par'97 Parallel Processing* (1997), pp. 1084–1091.

[GG97b] José González and Antonio González, « Speculative Execution via Address Prediction and Data Prefetching », *in*: *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, Vienna, Austria: ACM, 1997, pp. 196–203, ISBN: 0-89791-902-5, DOI: 10.1145/263580.263631, URL: http://doi.acm.org/10.1145/263580.263631.

[GM98] Freddy Gabbay and Avi Mendelson, « Using Value Prediction to Increase the Power of Speculative Execution Hardware », *in*: *ACM Trans. Comput. Syst.* 16.*3* (Aug. 1998), pp. 234–270, ISSN: 0734-2071, DOI: 10.1145/290409.290411, URL: http://doi.acm.org/10.1145/290409.290411.

[Goc+03] Simcha Gochman et al., « The Intel Pentium M processor: Microarchitecture and performance », *in*: 2003.

[GVD01] Bart Goeman, Hans Vandierendonck, and Koenraad De Bosschere, « Differential FCM: Increasing value prediction accuracy by improving table usage efficiency », *in*: *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, IEEE, 2001, pp. 207–216.

[Hin+01] Glenn Hinton et al., « The Microarchitecture of the Pentium 4 Processor », *in*: *Intel Technology Journal* 1 (2001), p. 2001.

[HM08] Mark D Hill and Michael R Marty, « Amdahl's law in the multicore era », *in*: *Computer* 41.*7* (2008).

[Huf57] David A Huffman, « The design and use of hazard-free switching networks », *in*: *Journal of the ACM (JACM)* 4.*1* (1957), pp. 47–62.

[Jim] D. A. Jimenez, « Multiperspective Perceptron Predictor », *in*: *in JWAC-5: Championship on Branch Prediction, https://www.jilp.org/cbp2016/* ().

[JL01] D. A. Jimenez and C. Lin, « Dynamic branch prediction with perceptrons », *in*: *Proc. HPCA Seventh Int. Symp. High-Performance Computer Architecture*, Jan. 2001, pp. 197–206, DOI: 10.1109/HPCA.2001.903263.

[Jou+98] Stephan Jourdan et al., « A novel renaming scheme to exploit value temporal locality through physical register reuse and unification », *in*: *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society Press, 1998, pp. 216–225.

[Kes99] R. E. Kessler, « The Alpha 21264 Microprocessor », *in*: *IEEE Micro* 19.*2* (Mar. 1999), pp. 24–36, ISSN: 0272-1732, DOI: 10.1109/40.755465, URL: https://doi.org/10.1109/40.755465.

[KS19] K. Kalaitzidis and A. Seznec, « Value Speculation through Equality Prediction », *in*: *Proc. IEEE 37th Int. Conf. Computer Design (ICCD)*, Nov. 2019, pp. 694–697, DOI: 10.1109/ICCD46524.2019.00101.

[LS96] M. H. Lipasti and J. P. Shen, « Exceeding the dataflow limit via value prediction », *in*: *Proc. 29th Annual IEEE/ACM Int. Symp. Microarchitecture. MICRO 29*, Dec. 1996, pp. 226–237, DOI: 10.1109/MICRO.1996.566464.

[Lu +07] Lu Peng et al., « Memory Performance and Scalability of Intel's and AMD's Dual-Core Processors: A Case Study », *in*: *2007 IEEE International Performance, Computing, and Communications Conference*, Apr. 2007, pp. 55–64, DOI: 10.1109/PCCC.2007.358879.

[LWS96] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen, « Value Locality and Load Value Prediction », *in*: *SIGPLAN Not.* 31.*9* (Sept. 1996), pp. 138–147, ISSN: 0362-1340, DOI: 10.1145/248209.237173, URL: http://doi.acm.org/10.1145/248209.237173.

[Man+06] M. Manusharow et al., « Dual die Pentium D package technology development », *in*: *56th Electronic Components and Technology Conference 2006*, May 2006, DOI: 10.1109/ECTC.2006.1645663.

[Man19] AMD64 Architecture Programmer's Manual, *in*: *AMD System TechDocs, Publication No. 24594, Revision 3.28, September* (2019).

[MLO98] Enric Morancho, José Maria Liaberia, and Àngel Olivé, « Split last-address predictor », *in*: *Proc. Int. Conf. Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, Oct. 1998, pp. 230–237, DOI: 10.1109/PACT.1998.727255.

[Moo98] G. E. Moore, « Cramming More Components Onto Integrated Circuits », *in*: *Proceedings of the IEEE* 86.*1* (Jan. 1998), pp. 82–85.

[Mor+80] Morse et al., « Intel Microprocessors-8008 to 8086 », *in*: *Computer* 13.*10* (Oct. 1980), pp. 42–60, DOI: 10.1109/MC.1980.1653375.

[Mor02] Enric Morancho Llena, « Address prediction and recovery mechanisms », PhD thesis, Universitat Politècnica de Catalunya, 2002.

[Mos+97] Andreas Moshovos et al., « Dynamic Speculation and Synchronization of Data Dependences », *in*: *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, Denver, Colorado, USA: ACM, 1997, pp. 181–193, ISBN: 0-89791-901-7, DOI: 10.1145/264107.264189, URL: http://doi.acm.org/10.1145/264107.264189.

[Mos98] Andreas Ioannis Moshovos, « Memory dependence prediction », PhD thesis, UNIVERSITY OF WISCONSIN—MADISON, 1998.

[MPR78] Stephen P Morse, William B Pohlman, and Bruce W Ravenel, « The Intel 8086 Microprocessor: a 16-bit Evolution of the 8080 », *in*: *Computer 6* (1978), pp. 18–27.

[NGS99] T. Nakra, R. Gupta, and M. L. Soffa, « Global context-based value prediction », *in*: *Proc. Fifth Int. Symp. High-Performance Computer Architecture*, Jan. 1999, pp. 4–12, DOI: 10.1109/HPCA.1999.744311.

[Per+03] Erez Perelman et al., « Using SimPoint for Accurate and Efficient Simulation », *in*: *SIGMETRICS Perform. Eval. Rev.* 31.*1* (June 2003), pp. 318–319, ISSN: 0163-5999, DOI: 10.1145/885651.781076, URL: http://doi.acm.org/10.1145/885651.781076.

[Per+15] Arthur Perais et al., « Cost-effective speculative scheduling in high performance processors », *in*: *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, IEEE, 2015, pp. 247–259.

[PJS97] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith, « Complexity-effective Superscalar Processors », *in*: *SIGARCH Comput. Archit. News* 25.*2* (May 1997), pp. 206–218, ISSN: 0163-5964, DOI: 10.1145/384286.264201, URL: http://doi.acm.org/10.1145/384286.264201.

[Pre16] Championship Branch Prediction, *in*: *Conjunction with the International Symposium on Computer Architecture https://www.jilp.org/cbp2016/* (2016).

[PS14a] A. Perais and A. Seznec, « EOLE: Paving the way for an effective implementation of value prediction », *in*: *Proc. ACM/IEEE 41st Int. Symp. Computer Architecture (ISCA)*, June 2014, pp. 481–492, DOI: 10.1109/ISCA.2014.6853205.

[PS14b] Arthur Perais and André Seznec, « Practical data value speculation for future high-end processors », *in*: *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, IEEE, 2014, pp. 428–439.

[PS15a] A. Perais and A. Seznec, « BeBoP: A cost effective predictor infrastructure for superscalar value prediction », *in*: *Proc. IEEE 21st Int. Symp. High Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 13–25, DOI: 10.1109/HPCA.2015.7056018.

[PS15b] A. Perais and A. Seznec, « EOLE: Toward a Practical Implementation of Value Prediction », *in*: *IEEE Micro* 35.*3* (May 2015), pp. 114–124, ISSN: 0272-1732, DOI: 10.1109/MM.2015.45.

[PS16] A. Perais and A. Seznec, « Cost effective physical register sharing », *in*: *Proc. IEEE Int. Symp. High Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 694–706, DOI: 10.1109/HPCA.2016.7446105.

[PS18] Arthur Perais and André Seznec, « Cost Effective Speculation with the Omnipredictor », *in*: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, Limassol, Cyprus: ACM, 2018, 25:1–25:13, ISBN: 978-1-4503-5986-3, DOI: 10.1145/3243176.3243208, URL: http://doi.acm.org/10.1145/3243176.3243208.

[PSR05] V. Petric, T. Sha, and A. Roth, « RENO: a rename-based instruction optimizer », *in*: *Proc. 32nd Int. Symp. Computer Architecture (ISCA'05)*, June 2005, pp. 98–109, DOI: 10.1109/ISCA.2005.43.

[Ryc+98] Bohuslav Rychlik et al., *Efficient and Accurate Value Prediction Using Dynamic Classification*, tech. rep., Carnegie Mellon University, 1998.

[RZ06] N. Riley and C. Zilles, « Probabilistic counter updates for predictor hysteresis and stratification », *in*: *Proc. Twelfth Int. Symp. High-Performance Computer Architecture*, Feb. 2006, pp. 110–120, DOI: 10.1109/HPCA.2006.1598118.

[Sat98] T. Sato, « Data dependence speculation using data address prediction and its enhancement with instruction reissue », *in*: *Proc. 24th EUROMICRO Conf. (Cat. No.98EX204)*, vol. 1, Aug. 1998, 285–292 vol.1, DOI: 10.1109/EURMIC.1998.711812.

[SC02] Eric Sprangle and Doug Carmean, « Increasing Processor Performance by Implementing Deeper Pipelines », *in*: *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, Anchorage, Alaska: IEEE Computer Society, 2002, pp. 25–34, ISBN: 0-7695-1605-X, URL: http://dl.acm.org/citation.cfm?id=545215.545219.

[SCD17] Rami Sheikh, Harold W. Cain, and Raguram Damodaran, « Load Value Prediction via Path-based Address Prediction: Avoiding Mispredictions Due to Conflicting Stores », *in*: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, Cambridge, Massachusetts: ACM, 2017, pp. 423–435, ISBN: 978-1-4503-4952-9, DOI: 10.1145/3123939.3123951, URL: http://doi.acm.org/10.1145/3123939.3123951.

[Seza] André Seznec, « Exploring value prediction with the EVES predictor », *in*: *in First Championship Value Prediction, CVP-1 2018, Los Angeles, June 3, 2018* ().

[Sezb] André Seznec, « TAGE-SC-L Branch Predictors Again », *in*: *in JWAC-5: Championship on Branch Prediction, https://www.jilp.org/cbp2016/* ().

[SH19] R. Sheikh and D. Hower, « Efficient Load Value Prediction Using Multiple Predictors and Filters », *in*: *Proc. IEEE Int. Symp. High Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 454–465, DOI: 10.1109/HPCA.2019.00057.

[Sim00] D. Sima, « The design space of register renaming techniques », *in*: *IEEE Micro* 20.*5* (Sept. 2000), pp. 70–83, ISSN: 1937-4143, DOI: 10.1109/40.877952.

[SM06] André Seznec and Pierre Michaud, « A case for (partially) TAgged GEometric history length branch prediction », *in*: *Journal of Instruction Level Parallelism* 8 (2006), pp. 1–23.

[Smi81] James E. Smith, « A Study of Branch Prediction Strategies », *in*: *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, pp. 135–148, URL: http://dl.acm.org/citation.cfm?id=800052.801871.

[Smi82] Alan Jay Smith, « Cache Memories », *in*: *ACM Comput Surv* 14.*3* (Sept. 1982), pp. 473–530, ISSN: 0360-0300, DOI: 10.1145/356887.356892, URL: http://doi.acm.org/10.1145/356887.356892.

[SS95] James E. Smith and Gurindar S. Sohi, « The Microarchitecture of Superscalar Processors », *in*: 1995.

[SS97] Y. Sazeides and J. E. Smith, « The predictability of data values », *in*: *Proc. 30th Annual Int. Symp. Microarchitecture*, Dec. 1997, pp. 248–258, DOI: 10.1109/MICRO.1997.645815.

[SS98] Yiannakis Sazeides and James E Smith, *Implementations of context based value predictors*, tech. rep., ECE-97-8, University of Wisconsin-Madison, 1998.

[Staa] Standard Performance Evaluation Corporation, « The SPEC CPU 2006 Benchmark Suite », *in*: *in http://www.spec.org* ().

[Stab] Standard Performance Evaluation Corporation, « The SPEC CPU 2017 Benchmark Suite », *in*: *in http://www.spec.org* ().

[STR02] A. Seznec, E. Toullec, and O. Rochecouste, « Register write specialization register read specialization: a path to complexity-effective wide-issue superscalar processors », *in*: *Proc. (MICRO-35) 35th Annual IEEE/ACM Int Symp. Microarchitecture*, Nov. 2002, pp. 383–394, DOI: 10.1109/MICRO.2002.1176265.

[SVS96] Y. Sazeides, S. Vassiliadis, and J. E. Smith, « The performance potential of data dependence speculation and collapsing », *in*: *Proc. 29th Annual IEEE/ACM Int. Symp. Microarchitecture. MICRO 29*, Dec. 1996, pp. 238–247, DOI: 10.1109/MICRO.1996.566465.

[TF70] G. S. Tjaden and M. J. Flynn, « Detection and Parallel Execution of Independent Instructions », *in*: *IEEE Transactions on Computers* C-19.*10* (Oct. 1970), pp. 889–895, ISSN: 2326-3814, DOI: 10.1109/T-C.1970.222795.

[Tom67] R. M. Tomasulo, « An Efficient Algorithm for Exploiting Multiple Arithmetic Units », *in*: *IBM Journal of Research and Development* 11.*1* (Jan. 1967), pp. 25–33, ISSN: 0018-8646, DOI: 10.1147/rd.111.0025.

[Tur37] A. M. Turing, « On Computable Numbers, with an Application to the Entscheidungsproblem », *in*: *Proceedings of the London Mathematical Society* s2-42.*1* (Jan. 1937), pp. 230–265, ISSN: 0024-6115, DOI: 10.1112/plms/s2-42.1.230, eprint: http://oup.prod.sis.lan/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf, URL: https://doi.org/10.1112/plms/s2-42.1.230.

[War+02] James D Warnock et al., « The circuit and physical design of the POWER4 microprocessor », *in*: *IBM Journal of Research and Development* 46.*1* (2002), pp. 27–51.

[WF97] K. Wang and M. Franklin, « Highly accurate data value prediction using hybrid predictors », *in*: *Proc. 30th Annual Int. Symp. Microarchitecture*, Dec. 1997, pp. 281–290, DOI: 10.1109/MICRO.1997.645819.

[YP92] Tse-Yu Yeh and Yale N. Patt, « Alternative Implementations of Two-level Adaptive Branch Prediction », *in*: *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, Queensland, Australia: ACM, 1992, pp. 124–134, ISBN: 0-89791-509-7, DOI: 10.1145/139669.139709, URL: http://doi.acm.org/10.1145/139669.139709.

[ZFC03] Huiyang Zhou, Jill Flanagan, and Thomas M. Conte, « Detecting Global Stride Locality in Value Streams », *in*: *SIGARCH Comput. Archit. News* 31.*2* (May 2003), pp. 324–335, ISSN: 0163-5964, DOI: 10.1145/871656.859656, URL: http://doi.acm.org/10.1145/871656.859656.

[ZK98] V. Zyuban and P. Kogge, « The Energy Complexity of Register Files », *in*: *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, ISLPED '98, Monterey, California, USA: ACM, 1998, pp. 305–310, ISBN: 1-58113-059-7, DOI: 10.1145/280756.280943, URL: http://doi.acm.org/10.1145/280756.280943.

LIST OF FIGURES

LIST OF PUBLICATIONS

CONFERENCES WITH PROCEEDINGS:

[C1] Kalaitzidis K., Seznec A. *"Value Speculation through Equality Prediction"*. 2019 IEEE 37th International Conference on Computer Design (ICCD), Abu Dhabi, United Arab Emirates, 2019, pp. 694-697.

WORKSHOPS WITHOUT PROCEEDINGS:

[W1] Kalaitzidis K., Seznec A. *"RSVP: A hybrid model of Register Sharing and Value Prediction"*. 2019 The first Young Architect Workshop (YArch'19), *Held in conjunction with the 25th IEEE International Symposium on High-Performance Computer Architecture,* (HPCA-25), Washington DC

UNIVERSITÉ DE RENNES 1

Titre : Spéculation Avancée pour Augmenter Les Performances des Processeurs Superscalaires

Mot clés : Parallélisme d'instructions, Prédiction de Valeurs, Prédiction d'adresse des lectures mémoire

Résumé : Même à l'ère des multicœurs, il est primordial d'améliorer la performance en contexte monocœur, étant donné l'existence de programmes qui exposent des parties séquentielles non négligeables. Les performances séquentielles se sont essentiellement améliorées avec le passage á l'échelle des structures de processeurs qui permettent le parallélisme d'instructions (ILP). Cependant, les chaînes de dépendances séquentielles limitent considérablement la performance. La prédiction de valeurs (VP) et la prédiction d'adresse des lectures mémoire (LAP) sont deux techniques en développement qui permettent de surmonter cet obstacle en permettant l'exécution d'instructions en spéculant sur les données.

Cette thèse propose des mécanismes basés sur VP et LAP qui conduisent à des améliorations de performances sensiblement plus élevées. D'abord, VP est examiné au niveau de l'ISA, ce qui fait apparaître l'impact de certaines particularités de l'ISA sur les performances. Ensuite, un nouveau prédicteur binaire (VSEP), qui permet d'exploiter certains motifs de valeurs, qui bien qu'ils soient fréquemment rencontrés, ne sont pas capturés par les modèles précédents, est introduit. VSEP améliore le speedup obtenu de 19% et, grâce à sa structure, il atténue le coût de la prédiction de valeurs supérieures à 64 bits. Adapter cette approche pour effectuer LAP permet de prédire les adresses de 48% des lectures mémoire. Finalement, une microarchitecture qui exploite ce mécanisme de LAP peut exécuter 32% des lectures mémoire en avance.

Title: Advanced Speculation to Increase the Performance of Superscalar Processors

Keywords: Instruction-Level Parallelism, Value Prediction, Load-address Prediction

Abstract: Even in the multicore era, making single cores faster is paramount to achieve high-performance computing, given the existence of programs that are either inherently sequential or expose non-negligible sequential parts. Sequential performance has been essentially improving with the scaling of the processor structures that enable instruction-level parallelism (ILP). However, as modern microarchitectures continue to extract more ILP by employing larger instruction windows, true data dependencies remain a major performance bottleneck. *Value Prediction* (VP) and *Load-Address Prediction* (LAP) are two developing techniques that allow to overcome this obstacle and harvest more ILP by enabling the execution of instructions in a *data-wise* speculative manner.

This thesis proposes mechanisms that are related with VP and LAP and lead to effectively higher performance improvements. First, VP is examined in an ISA-aware manner, that discloses the impact of certain ISA particularities on the anticipated speedup. Second, a novel binary-based VP model is introduced, namely VSEP, that allows to exploit certain value patterns that although they are encountered frequently, they can not be captured by previous works. VSEP improves the obtained speedup by 19% and also, by virtue of its structure, it mitigates the cost of predicting values wider than 64 bits. By adapting this approach to perform LAP allows to predict the memory addresses of 48% of the committed loads. Eventually, a microarchitecture that leverages this LAP mechanism can execute 32% of the committed loads early.