Increasing the performance of superscalar processors through value prediction

Arthur Perais

ANNÉE 2015

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Arthur PERAIS

préparée à l'unité de recherche INRIA
Institut National de Recherche en Informatique et Automatique – Université de Rennes 1

Increasing the Performance of Superscalar Processors through Value Prediction

Thèse soutenue à Rennes le 24 septembre 2015

devant le jury composé de :

Christophe WOLINSKI
Professeur à l'Université de Rennes 1 / *Président*

Frédéric PÉTROT
Professeur à l'ENSIMAG, Institut Polytechnique de Grenoble / *Rapporteur*

Pierre BOULET
Professeur à l'Université de Lille 1 / *Rapporteur*

Christine ROCHANGE
Professeur à l'Université de Toulouse / *Examinatrice*

Pierre MICHAUD
Chargé de Recherche à l'INRIA Rennes – Bretagne Atlantique / *Examinateur*

André SEZNEC
Directeur de recherche à l'INRIA Rennes – Bretagne Atlantique / *Directeur de thèse*

Je sers la science et c'est ma joie.
Basile, disciple

Remerciements

Je tiens à remercier les membres de mon jury pour avoir jugé les travaux réalisés durant cette thèse, ainsi que pour leurs remarques et questions.

En particulier, je remercie Christophe Wolinski, Professeur à l'Université de Rennes 1, qui m'a fait l'honneur de présider ledit jury. Je remercie aussi Frédéric Pétrot, Professeur à l'ENSIMAG, et Pierre Boulet, Professeur à l'Université de Lille 1, d'avoir bien voulu accepter la charge de rapporteur. Je remercie finalement Christine Rochange, Professeur à l'Université de Toulouse et Pierre Michaud, Chargé de Recherche à l'INRIA Bretagne Atlantique, d'avoir accepté de juger ce travail.

Je souhaite aussi remercier tout particulièrement Yannakis Sazeides, Professeur à l'Université de Chypre. Tout d'abord pour avoir accepté de siéger dans le jury en tant que membre invité, mais aussi pour avoir participé à l'établissement des fondements du domaine sur lequel mes travaux de thèse se penchent.

Je tiens à remercier André Seznec pour avoir assuré la direction et le bon déroulement de cette thèse. Je lui suis particulièrement reconnaissant d'avoir transmis sa vision de la recherche ainsi que son savoir avec moi. J'ose espérer que toutes ces heures passées à tolérer mon ignorance auront au moins été divertissantes pour lui.

Je veux remercier les membres de l'équipe ALF pour toutes les discussions sérieuses et moins sérieuses qui auront sûrement influencées la rédaction de ce document.

Mes remerciements s'adressent aussi aux membres de ma famille, auxquels j'espère avoir pû fournir une idée plus ou moins claire de la teneur de mes travaux de thèse au cours des années. Ce document n'aurait jamais pû voir le jour sans leur participation active à un moment ou à un autre.

Je suis reconnaissant à mes amis, que ce soit ceux avec lesquels j'ai pû quotidiennement partager les joies du doctorat, ou ceux qui ont eu le bonheur de choisir une autre activité.[1]

Finalement, un merci tout particulier à ma relectrice attitrée, sans laquelle de nombreuses fôtes subsisteraient dans ce manuscrit. Grâce à elle, la rédaction de

[1] "Un vrai travail"

ce manuscrit fut bien plus plaisante qu'elle n'aurait dû l'être.

Contents

Résumé en français

Le début des années 2000 aura marqué la fin de l'ère de *l'uni-processeur*, et le début de l'ère des *multi-cœurs*. Précédemment, chaque génération de processeur garantissait une amélioration super-linéaire de la performance grâce à l'augmentation de la fréquence de fonctionnement du processeur ainsi que la mise en œuvre de micro-architectures (algorithme matériel avec lequel le processeur exécute les instructions) intrinsèquement plus efficaces. Dans les deux cas, l'amélioration était généralement permise par les avancées technologiques en matière de gravure des transistors. En particulier, il était possible de graver deux fois plus de transistors sur la même surface tous les deux ans, permettant d'améliorer significativement la micro-architecture. L'utilisation de transistors plus fins a aussi généralement permis d'augmenter la fréquence de fonctionnement.

Cependant, il est vite apparu que continuer à augmenter la fréquence n'était pas viable dans la mesure où la puissance dissipée augmente de façon super-linéaire avec la fréquence. Ainsi, les 3,8GHz du Pentium 4 Prescott (2004) semblent être la limite haute de la fréquence atteignable par un processeur destiné au grand public. De plus, complexifier la micro-architecture d'un uni-processeur en ajoutant toujours plus de transistors souffre d'un rendement décroissant. Il est donc rapidement apparu plus profitable d'implanter plusieurs processeurs – *cœurs* – sur une même puce et d'adapter les programmes afin qu'ils puissent tirer parti de plusieurs coeurs, plutôt que de continuer à améliorer la performance d'un seul cœur. En effet, pour un programme purement parallèle, l'augmentation de la performance globale approche 2 si l'on a deux cœurs au lieu d'un seul (tous les cœurs étant identiques). Au contraire, si l'on dépense les transistors utilisés pour mettre en œuvre le second cœur à l'amélioration du cœur restant, alors l'expérience dicte que l'augmentation de la performance sera inférieure à 2.

Nous avons donc pu constater au cours des dernières années une augmentation du nombre de coeurs par puce pour arriver à environ 16 aujourd'hui. Par conséquent, la performance des programmes parallèles a pu augmenter de manière significative, et en particulier, bien plus rapidement que la performance des programmes purement séquentiels, c'est à dire dont la performance globale dépend de la performance d'un unique cœur. Ce constat est problématique dans la mesure où il existe de nombreux programmes étant intrinsèquement séquentiels, c'est à

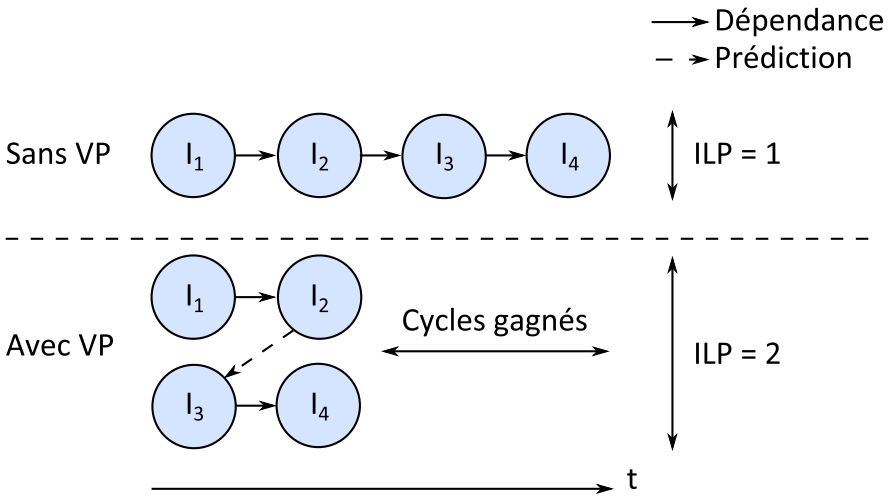


Figure 1: Impact de la prédiction de valeurs (VP) sur la chaîne de dépendances séquentielles exprimée par un programme.

dire ne pouvant pas bénéficier de la présence de plusieurs coeurs. De plus, les programmes parallèles possèdent en général une portion purement séquentielle. Cela implique que même avec une infinité de coeurs pour exécuter la partie parallèle, la performance sera limitée par la partie séquentielle, c'est la loi d'Amdahl. Par conséquent, même à l'ère des multi-coeurs, il reste nécessaire de continuer à améliorer la performance séquentielle.

Hélas, la façon la plus naturelle d'augmenter la performance séquentielle (augmenter le nombre d'instructions traitées chaque cycle ainsi que la taille de la fenêtre spéculative d'instructions) est connue pour être très couteuse en transistors et pour augmenter la consommation énergétique ainsi que la puissance dissipée. Il est donc nécessaire de trouver d'autres moyens d'améliorer la performance sans augmenter la complexité du processeur de façon déraisonnable.

Dans ces travaux, nous revisitions un tel moyen. La prédiction de valeurs (VP) permet d'augmenter le parallélisme d'instructions (le nombre d'instructions pouvant s'exécuter de façon concurrente, ou ILP) disponible en spéculant sur le résultat des instructions. De ce fait, les instructions dépendantes de résultats spéculés peuvent être exécutées plus tôt, augmentant la performance séquentielle. La prédiction de valeurs permet donc une meilleure utilisation des ressources matérielles déjà présentes et ne requiert pas d'augmenter le nombre d'instructions que le processeur peut traiter en parallèle. La Figure 1 illustre comment la prédiction du résultat d'une instruction permet d'extraire plus de parallélisme d'instructions que la sémantique séquentielle du programme n'exprime.

Cependant, la prédiction de valeurs requiert un mécanisme validant les prédictions et s'assurant que l'exécution reste correcte même si un résultat est

mal prédit. En général, ce mécanisme est implanté directement dans le cœur d'exécution dans le désordre, qui est déjà très complexe en lui-même. Mettre en œuvre la prédiction de valeurs avec ce mécanisme de validation serait donc contraire à l'objectif de ne pas complexifier le processeur.

De plus, la prédiction de valeurs requiert un accès au fichier de registres, qui est la structure contenant les valeurs des opérandes des instructions à exécuter, ainsi que leurs résultats. En effet, une fois un résultat prédit, il doit être inséré dans le fichier de registres afin de pouvoir être utilisé par les instructions qui en dépendent. De même, afin de valider une prédiction, elle doit être lue depuis le fichier de registres et comparée avec le résultat non spéculatif. Ces accès sont en concurrence avec les accès effectués par le moteur d'exécution dans le désordre, et des ports de lectures/écritures additionnels sont donc requis afin d'éviter les conflits. Cependant, la surface et la consommation du fichier de registres croissent de façon super-linéaire avec le nombre de ports d'accès. Il est donc nécessaire de trouver un moyen de mettre en œuvre la prédiction de valeurs sans augmenter la complexité du fichier de registres.

Enfin, la structure fournissant les prédictions au processeur, le prédicteur, doit être aussi simple que possible, que ce soit dans son fonctionnement ou dans le budget de stockage qu'il requiert. Cependant, il doit être capable de fournir plusieurs prédictions par cycle, puisque le processeur peut traiter plusieurs instructions par cycle. Cela sous-entend que comme pour le fichier de registres, plusieurs ports d'accès sont nécessaires afin d'accéder à plusieurs prédictions chaque cycle. L'ajout d'un prédicteur de plusieurs kilo-octets possédant plusieurs ports d'accès aurait un impact non négligeable sur la consommation du processeur, et devrait donc être évité.

Ces trois exigences sont les principales raisons pour lesquelles la prédiction de valeurs n'a jusqu'ici pas été implantée dans un processeur. Dans ces travaux de thèse, nous revisitons la prédiction de valeurs dans le contexte actuel. En particulier, nous proposons des solutions aux trois problèmes mentionnés ci-dessus, permettant à la prédiction de valeurs de ne pas augmenter significativement la complexité du processeur, déjà très élevée. Ainsi, la prédiction de valeurs devient une façon réaliste d'augmenter la performance séquentielle.

Dans un premier temps, nous proposons un nouveau prédicteur tirant parti des avancées récentes dans le domaine de la prédiction de branchement. Ce prédicteur est plus performant que les prédicteurs existants se basant sur le *contexte* pour prédire. De plus, il ne souffre pas de difficultés intrinsèques de mise en œuvre telle que le besoin de gérer des historiques locaux de façon spéculative. Il s'agit là d'un premier pas vers un prédicteur efficace mais néanmoins réalisable. Nous montrons aussi que si la précision du prédicteur est très haute, i.e., les mauvaises prédictions sont très rares, alors la validation des prédictions peut se faire lorsque les instructions sont retirées du processeurs, dans l'ordre. Il n'est donc

pas nécessaire d'ajouter un mécanisme complexe dans le cœur d'exécution pour valider les prédictions et récupérer d'une mauvaise prédiction. Afin de s'assurer que la précision est élevée, nous utilisons un mécanisme d'estimation de confiance probabiliste afin d'émuler des compteurs de 8-10 bits en utilisant seulement 3 bits par compteur.

Dans un second temps, nous remarquons que grâce à la prédiction de valeurs, de nombreuses instructions sont prêtes à être exécutées bien avant qu'elle n'entrent dans le moteur d'exécution dans le désordre. De même, les instructions prédites n'ont pas besoin d'être exécutées avant le moment ou elles sont retirées du processeur, puisque les instructions qui en dépendent peuvent utiliser la prédiction comme opérande source. De ce fait, nous proposons un nouveau modèle d'exécution, EOLE, dans lequel de nombreuses instructions sont exécutées dans l'ordre, hors du moteur d'exécution dans le désordre. Ainsi, le nombre d'instructions que le moteur d'exécution dans le désordre peut traiter à chaque cycle peut être réduit, et donc le nombre de ports d'accès qu'il requiert sur le fichier de registre. Nous obtenons donc un processeur avec des performances similaires à un processeur utilisant la prédiction de valeurs, mais dont le moteur d'exécution dans le désordre est plus simple, et dont le fichier de registres nécessite autant de ports d'accès qu'un processeur sans prédiction de valeurs.

Finalement, nous proposons une infrastructure de prédiction capable de prédire plusieurs instructions à chaque cycle, tout en ne possédant que des structures avec un seul port d'accès. Pour ce faire, nous tirons parti du fait que le processeur exécute les instructions de façon séquentielle, et qu'il est donc possible de grouper les prédictions qui correspondent à des instructions contiguës en mémoire dans une seule entrée du prédicteur. Il devient donc possible de récupérer toutes les prédictions correspondant à un bloc d'instructions en un seul accès au prédicteur. Nous discutons aussi de la gestion spéculative du contexte requis par certains prédicteurs pour prédire les résultats, et proposons une solution réaliste tirant aussi parti de l'agencement en bloc des instructions. Nous considérons aussi des façons de réduire le budget de stockage nécessaire au prédicteur et montrons qu'il est possible de n'utiliser que 16-32 kilo-octets, soit un budget similaire à celui du cache de premier niveau ou du prédicteur de branchement.

Au final, nous obtenons une amélioration moyenne des performances de l'ordre de 11.2%, allant jusqu'à 62.2%. Les performances obtenues ne sont jamais inférieures à celles du processeur de référence (i.e., sans prédiction de valeurs) sur les programmes considérés. Nous obtenons ces chiffres avec un processeur dont le moteur d'exécution dans le désordre est significativement moins complexe que dans le processeur de référence, tandis que le fichier de registres est équivalent au fichier de registres du processeur de référence. Le principal coût vient donc du prédicteur de valeur, cependant ce coût reste raisonnable dans la mesure où il est possible de mettre en œuvre des tables ne possédant qu'un seul port d'accès.

Ces trois contributions forment donc une mise en œuvre possible de la prédiction de valeurs au sein d'un processeur généraliste haute performance moderne, et proposent donc une alternative sérieuse à l'augmentation de la taille de la fenêtre spéculative d'instructions du processeur afin d'augmenter la performance séquentielle.

Introduction

Processors – in their broader sense, i.e., general purpose or not – are effectively the cornerstone of modern computer science. To convince ourselves, let us try to imagine for a minute what computer science would consist of without such chips. Most likely, it would be relegated to a field of mathematics as there would not be enough processing power for graphics processing, bioinformatics (including DNA sequencing and protein folding) and machine learning to simply exist. Languages and compilers would also be of much less interest in the absence of machines to run the compiled code on. Evidently, processor usage is not limited to computer science: Processors are everywhere, from phones to cars, to houses, and many more.

Yet, processors are horrendously complex to design, validate, and manufacture. This complexity is often not well understood by their users. To give an idea of the implications of processor design, let us consider the atomic unit of a modern processor: the transistor. Most current generation general purpose microprocessor feature *hundreds of millions* to *billions* of transistors. By connecting those transistors in a certain fashion, binary functions such as *OR* and *AND* can be implemented. Then, by combining these elementary functions, we obtain a device able to carry out higher-level operations such as additions and multiplications. Although all the connections between transistors are not explicitly designed by hand, many high-performance processors feature custom logic where circuit designers act at the transistor level. Consequently, it follows that designing and validating a processor is extremely time and resource consuming.

Moreover, even if modern general purpose processors possess the ability to perform billions of arithmetic operations per second, it appears that to keep a steady pace in both computer science research and quality of life improvement, processors should keep getting faster and more energy efficient. Those represent the two main challenges of processor architecture.

To achieve these goals, the architect acts at a level higher than the transistor but not much higher than the logic gate (e.g., OR, AND, etc.). In particular, he or she can alter the way the different components of the processor are connected, the way they operate, and simply add or remove components. This is in contrast with the circuit designer that would improve performance by rearranging tran-

sistors inside a component without modifying the function carried out by this component, in order to reduce power consumption or shorten the electric critical path. As a result, the computer architect is really like a regular architect: he or she draws the blueprints of the processor (respectively house), while the builders (resp. circuit designers) are in charge of actually building the processor with transistors and wires (resp. any house material). It should be noted, however, that the processor architect should be aware of what is actually doable with the current circuit technology, and what is not. Therefore, not unlike a regular architect, he or she should keep the laws of physics in mind when drawing his or her plans.

In general, processor architecture is about tradeoffs. For instance, tradeoffs between simplicity ("My processor can just do additions in hardware, but it was easy to build") and performance ("The issue is that divisions are very slow because I have to emulate them using additions"), or tradeoffs between raw compute power ("My processor can do 4 additions at the same time") and power efficiency ("It dissipates more power than a vacuum cleaner to do it"). Therefore, in a nutshell, architecture consists in designing a computer processor from a reasonably high level, by taking into account those tradeoffs and experimenting with new organizations. In particular, processor architecture has lived through several paradigm shifts in just a few decades, going from 1-cycle architectures to modern deeply pipelined superscalar processors, from single core to multi-core, from very high frequency to very high power and energy efficiency. Nonetheless, the goals remain similar: increase the amount of work that can be done in a given amount of time, while minimizing the energy spent doing this work.

Overall, there are two main directions to improve processor performance. The first one leverages the fact that processors are clocked objects, meaning that when the associated clock ticks, then some computation step has finished. For instance, a processor may execute one instruction per clock tick. Therefore, if the clock speed of the processor is increased, then more work can be done in a given unit of time, as there are now more clock ticks in said unit of time. However, as the clock frequency is increased, the power consumption of the processor grows super-linearly,[2] and quickly becomes too high to handle. Moreover, transistors possess an intrinsic maximum switching frequency and a signal can only travel so far in a wire in a given clock period. Exceeding these two limits would lead to unpredictable results. For these reasons, clock frequency has generally remained comprised between 2GHz and 5GHz in the last decade, and this should remain the case for the same reasons.

Second, thanks to improvements in transistor implantation processes, more and more transistors can be crammed in a given surface of silicon. This phe-

[2]Since the supply voltage must generally grow as the frequency is increased, power consumption increases super-linearly with frequency. If the supply voltage can be kept constant somehow, then the increase should be linear.

nomenon is embodied by Moore's Law, that states that the number of transistors than can be implanted on a chip doubles every 24 months [Moo98]. Although it is an empirical law, it has proven exactly right between 1971 and 2001. However, increase has slowed down during the past decade. Nonetheless, consider that the Intel 8086 released in 1978 had between 20,000 and 30,000 transistors while some modern Intel processors have more than 2 billions, that is 100,000 times more.[3] However, if a tremendous number of transistors are available to the architect, it is becoming extremely hard to translate these additional transistors into additional performance.

In the past, substantial performance improvements due to an increased number of transistors came from the improvement of uni-processors through *pipelining*, *superscalar execution* and *out-of-order execution*. All those features target sequential performance, i.e., the performance of a single processor. However, it has become very hard to keep on targeting sequential performance with the additional transistors we get, because of diminishing returns. Therefore, as no new paradigm was found to replace the current one (superscalar out-of-order execution), the trend regarding high performance processors has shifted from uni-processor chips to multi-processor chips. This allows to obtain a performance increase on par with the additional transistors provided by transistor technology, keeping Moore's Law alive in the process.

More specifically, an intuitive way to increase the performance of a program is to cut it down into several smaller programs that can be executed concurrently. Given the availability of many transistors, multiple processors can be implemented on a single chip, with each processor able to execute one piece of the initial program. As a result, the execution time of the initial program is decreased. This not-so-new paradigm targets parallel performance (i.e., throughput: how much work is done in a given unit of time), as opposed to sequential performance (i.e., latency or response time: how fast can a task finish).

However, this approach has two limits. First, some programs may not be parallel at all due to the nature of the algorithm they are implementing. That is, it will not be possible to break them down into smaller programs executed concurrently, and they will therefore not benefit from the additional processors of the chip. Second, Amdahl's Law [Amd67] limits the performance gain that can be obtained by parallelizing a program. Specifically, this law represents a program as a sequential part and a parallel part, with the sum of those parts adding up to 1. Among those two parts, only the parallel one can be sped up by using several processors. For instance, for a program whose sequential part is equivalent to the parallel part, execution time can be reduced by a factor of 2 at most, that is if there is an infinity of processors to execute the parallel part. Therefore, even

[3] Moore's Law would predict around 5 billions.

with many processors at our disposal, the performance of a single processor (the one executing the sequential part) quickly becomes the limiting factor. Given the existence of intrinsically sequential programs and the fact that even for parallel programs, the sequential part is often non-negligible, it is paramount to keep on increasing single processor performance.

This thesis provides realistic means to substantially improve the sequential performance of high performance general purpose processors. These means are completely orthogonal to the coarse grain parallelism leveraged by *Chip Multi-Processors* (CMP, several processors on a single chip a.k.a. multi-cores). They can therefore be added on-top of existing hardware. Furthermore, if this thesis focuses on the design step of processor conception, technology constraints are kept in mind. In other words, specific care is taken to allow the proposed mechanisms to be realistically implementable using current technology.

Contributions

We revisit a mechanism aiming to increase sequential performance: *Value Prediction* (VP). The key idea behind VP is to predict instruction results in order to break dependency chains and execute instructions earlier than previously possible. In particular, for out-of-order processors, this leads to a better utilization of the execution engine: Instead of being able to uncover more Instruction Level Parallelism (ILP) from the program by being able to look further ahead (e.g., by having a larger instruction window), the processor *creates* more ILP by breaking some true dependencies in the existing instruction window.

VP relies on the fact that there exist common instructions (e.g., *load*, read data from memory) that can take up to a few hundreds of processor cycles to execute. During this time, no instruction requiring the data retrieved by the *load* instruction can be executed, most likely stalling the whole processor. Fortunately, many of these instructions often produce the same result, or results that follow a pattern, which could therefore be predicted. Consequently, by predicting the result of a *load* instruction, dependents can execute while the memory hierarchy is processing the *load*. If the prediction is correct, execution time decreases, otherwise, dependent instructions must be re-executed with the correct input to enforce correctness.

Our first contribution [PS14b] makes the case that contrarily to previous implementations described in the 90's, it is possible to add Value Prediction on top of a modern general purpose processor without adding tremendous complexity in its pipeline. In particular, it is possible for VP to intervene only in the front-end of the pipeline, at Fetch (to predict), and in the last stage of the back-end, Commit (to validate predictions and recover if necessary). Validation at Commit is

rendered possible because squashing the whole pipeline can be used to recover from a misprediction, as long as value mispredictions are rare. This is in contrast with previous schemes where instructions were replayed directly inside the out-of-order window. In this fashion, all the – horrendously complex – logic responsible for out-of-order execution need not be modified to support VP. This is a substantial improvement over previously proposed implementations of VP that coupled VP with the out-of-order execution engine tightly. In addition, we introduce a new hybrid value predictor, D-VTAGE, inspired from recent advances in the field of branch prediction. This predictor has several advantages over existing value predictors, both from a performance standpoint and from a complexity standpoint.

Our second contribution [PS14a, PS15b] focuses on reducing the additional Physical Register File (PRF) ports required by Value Prediction. Interestingly, we achieve this reduction because VP actually enables a reduction in the complexity of the out-of-order execution engine. Indeed, an instruction whose result is predicted does not need to be executed as soon as possible, since its dependents can use the prediction to execute. Therefore, we introduce some logic in charge of executing predicted instructions as late as possible, just before they are removed from the pipeline. These instructions never enter the out-of-order window. We also notice that thanks to VP, many instructions are ready to be executed very early in the pipeline. Consequently, we also introduce some logic in charge of executing such instructions in the front-end of the pipeline. As a result, instructions that are executed early are not even dispatched to the out-of-order window. With this {Early | Out-of-Order | Late} (EOLE) architecture, a substantial portion of the dynamic instructions is not executed inside the out-of-order execution engine, and said engine can be simplified by reducing its issue-width, for instance. This mechanically reduces the number of ports on the register file, and is a strong argument in favor of Value Prediction since the complexity and power consumption of the out-of-order execution engine are already substantial.

Our third and last contribution [PS15a] relates to the design of a realistic predictor infrastructure allowing the prediction of several instructions per cycle with single-ported structures. Indeed, modern processors are able to retrieve several instructions per cycle from memory, and the predictor should be able to provide a prediction for each of them. Unfortunately, accessing several predictor entries in a single cycle usually requires several access ports, which should be avoided when considering such a big structure (16-32KB). To that extent, we propose a block-based prediction infrastructure that provides predictions for the whole instruction fetch block with a single read in the predictor. We achieve superscalar prediction even when a variable-length instruction set such as x86 is used. We show that combined to a reasonably sized D-VTAGE predictor, performance improvements are close to those obtained by using an ideal infrastructure combined

to a much bigger value predictor. Moreover, even with these modifications, VP still enables a reduction in the complexity of the out-of-order execution engine thanks to EOLE.

Combined, these three contributions provide a possible implementation of Value Prediction on top of a modern superscalar processor. Contrarily to previous solutions, this one does not increase the complexity of already complex processor components such as the out-of-order execution engine. In particular, prediction and prediction validation can both take place outside said engine, limiting the interactions between VP and the engine to the register file only. Moreover, we provide two structures that execute instructions either early or late in the pipeline, offloading the execution of many instructions from the out-of-order engine. As a result, it can be made less aggressive, and VP viewed as a means to reduce complexity rather than increase it. Finally, by devising a better predictor inspired from recent branch predictors and providing a mechanism to make it single-ported, we are able to outperform previous schemes and address complexity in the predictor infrastructure.

Organization

This document is divided in five chapters and architected around three published contributions: two in the International Symposium on High Performance Computer Architecture (HPCA 2014/15) [PS14b, PS15a] and one in the International Symposium on Computer Architecture (ISCA 2014) [PS14a].

Chapter 1 provides a description of the inner workings of modern general purpose processors. We begin with the simple pipelined architecture and then improve on it by following the different processor design paradigms in a chronological fashion. This will allow the reader to grasp why sequential performance bottlenecks remain in deeply pipelined superscalar processors and understand how Value Prediction can help mitigate at least one of such bottlenecks.

Chapter 2 introduces Value Prediction in more details and presents previous work. Existing hardware prediction algorithms are detailed and their advantages and shortcomings discussed. In particular, we present *computational* predictors that compute a prediction from a previous result and *context-based* predictor that provide a prediction when some current information history matches a previously seen pattern [SS97a]. Then, we focus on how VP – and especially prediction validation – has usually been integrated with the pipeline. Specifically, we illustrate how existing implementations fall short of mitigating the complexity introduced by VP in the out-of-order execution engine.

Chapter 3 focuses on our first [PS14b] and part of our third [PS15a] contributions. We describe a new predictor, VTAGE, that does not suffer from the

shortcomings of existing context-based predictors. We then improve on it by hybridizing it with a computational predictor in an efficient manner and present Differential VTAGE (D-VTAGE). In parallel, we show that Value Prediction can be plugged into an existing pipeline without increasing complexity in parts of the pipeline that are already tremendously complex. We do so by delaying prediction validation and prediction recovery until in-order retirement.

Chapter 4 goes further by showing that Value Prediction can be leveraged to actually reduce complexity in the out-of-order window of a modern processor. By enabling a reduction in the out-of-order issue-width through executing instruction early or late in the pipeline, the EOLE architecture [PS14a] outperforms a wider processor while having 33% less out-of-order issue capacity and the same number of register file ports.

Finally, Chapter 5 comes back to the predictor infrastructure and addresses the need to provide several predictions per cycle in superscalar processors. We detail our block-based scheme enabling superscalar value prediction. We also propose an implementation of the speculative window tracking inflight predictions that is needed to compute predictions in many predictors [PS15a]. Lastly, we study the behavior of the D-VTAGE predictor presented in Chapter 2 when its storage budget is reduced to a more realistic 16-32KB.

Chapter 1

The Architecture of a Modern General Purpose Processor

General purpose processors are programmable devices, hence, they present an interface to the programmer known as the *Instruction Set Architecture* or ISA. Among the few widely used ISA, the x86 instruction set of Intel stands out from the rest as it remained retro-compatible across processor generations. That is, a processor bought in 2015 will be able to execute a program written for the 8086 (1978). Fortunately, this modern processor will be much faster than a 8086. As a result, it is important to note that the interface exposed to the software is not necessarily representative of the underlying organization of the processor and therefore its computation capabilities. It is required to distinguish between the architecture or ISA (e.g., x86, ARMv7, ARMv8, Alpha) from the *micro-architecture* implementing the architecture on actual silicon (e.g. Intel Haswell or AMD Bulldozer for x86, ARM Cortex A9 or Qualcomm Krait for ARMv7). In this thesis, we focus on the micro-architecture of a processor and aim to remain agnostic of the actual ISA.

1.1 Instruction Set – Software Interface

Each ISA defines a set of instructions and a set of registers. Registers can be seen as scratchpads from which instructions read their operands and to which instructions write their results. They also define an address space used to access global memory, which can be seen as a backing store for registers. These three components are visible to the software, and they form the interface with which the processor can be programmed to execute a specific task.

The distinction is often made between *Reduced Instruction Set Computer* ISAs (RISC) and *Complex Instruction Set Computer* ISAs (CISC). The former indicates that instructions visible to the programmer are usually limited to simple

operations (e.g., arithmetic, logic and branches). The latter provides the programmer with more complex instructions (e.g., string processing, loop instruction). Therefore, RISC usually has less instructions available to the programmer but its functionalities are easier to implement. As a result, more instructions are generally required than in CISC to do the same amount of work. However, CISC is usually considered as harder to implement because of the complex processing carried out by some of its instructions.

The main representatives of RISC ISAs are ARMv7 (32-bit) and ARMv8 (64-bit) while the most widely used CISC ISA is x86 in its 64-bit version (x86_64). 32- and 64-bit refer to the width of the – virtual – addresses used to access global memory. That is, ARMv7 can only address 2^{32} memory locations (4 GB), while ARMv8 and x86_64 can address 2^{64} memory locations (amounting to 16 EB).

1.1.1 Instruction Format

The instructions of a program are stored in global memory using a binary representation. Each particular ISA defines how instructions are encoded. Nonetheless, there exists two main paradigms for instruction encoding.

First, a fixed-length can be used, for instance 4 bytes (32 bits) in ARMv7. These 4 bytes contain – among other information – an *opcode* (the nature of the operation to carry out), two *source registers* and one *destination register*. In that case, decoding the binary representation of the instruction to generate the control signals for the rest of the processor is generally straightforward. Indeed, each field is always stored at the same offset in the 4-byte word. Unfortunately, memory capacity is wasted for instructions that do not need two sources (e.g., incrementing a single register) or that do not produce any output (e.g., branches).

Conversely, a variable-length can be used to encode different instructions. For instance, in x86, instructions can occupy 1 to 15 bytes (120 bits!) in memory. As a result, instruction decoding is much more complex as the location of a field in the instruction word (e.g., the first source register) depends on the value of previous fields. Moreover, since an instruction has variable-length, it is not straightforward to identify where the next instruction is. Nonetheless, variable-length representation is generally more storage-efficient.

1.1.2 Architectural State

Each ISA defines an architectural state. It is the state of the processor that is visible by the programmer. It consists of all registers[1] defined by the ISA as well as the state of global memory, and it is agnostic of which program is currently

[1]Including the *Program Counter* (PC) pointing to the memory location containing the instruction to execute next.

being executed on the processor. The instructions defined by the ISA operate on this architectural state. For instance, an addition will take the architectural state as its input (or rather, a subset of the architectural state) and output a value that will be put in a register, hence modify the architectural state. From the software point of view, this modification is atomic: there is no architectural state between the state *before* the addition and the state *after* the addition.

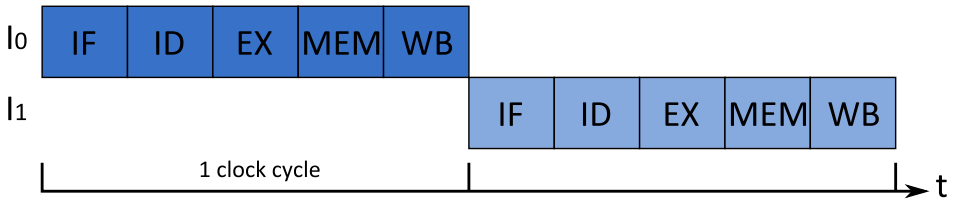
Consequently, a program is a sequence of instructions (as defined by the ISA) that keeps on modifying the architectural state to compute a result. It follows that programs running on the processor have sequential semantics. That is, from the point of view of the software, the architectural state is modified atomically by instructions following the order in which they appear in the program. In practice, the micro-architecture may work on its own internal version of the architectural state as it pleases, as long as updates to the software-visible architectural state respect atomicity and the sequential semantics of the program. In the remainder of this document, we differentiate between the software-visible state and any *speculative* (i.e., not software-visible yet) state living inside the processor.

1.2 Simple Pipelining

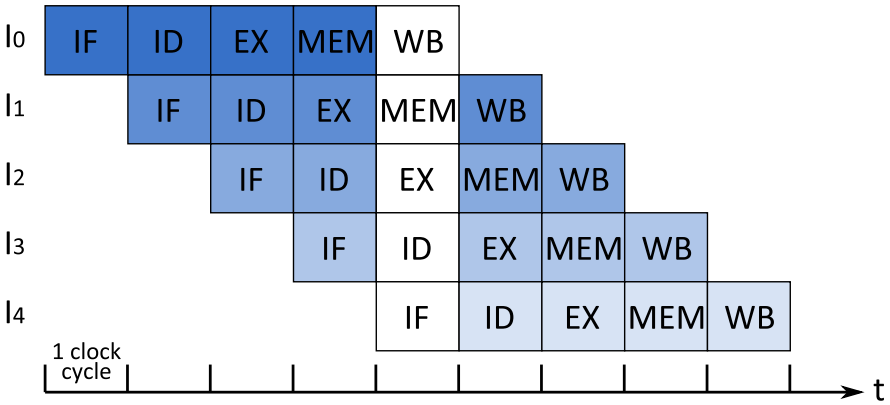
Processors are clocked devices, meaning that they are driven by a clock that ticks a certain number of times per second. Each tick triggers the execution of some work in the processor. Arguably, the most intuitive micro-architecture is therefore the *1-cycle micro-architecture* where the operation triggered by the clock tick is the entire execution of an instruction. To guarantee that this is possible, the clock has to tick slow enough to allow the slowest (the one requiring the most processing) instruction to execute in a single clock cycle.

Regardless, the execution of an instruction involves several very different steps requiring very different hardware structures. In a 1-cycle micro-architecture, a single instruction flows through all these hardware structures during the clock cycle. Assuming there are five steps to carry out to actually execute the instruction and make its effects visible to the software, this – roughly – means that each of the five distinct hardware structures is idle during four-fifth of the clock cycle. As a result, this micro-architecture is highly inefficient.

Pipelined micro-architectures were introduced to remedy this issue. The high-level idea is that since an instruction flows sequentially through several *stages* to be processed, there is potential to process instruction I_0 in stage S_1 while instruction I_1 is concurrently processed in stage S_0. Therefore, resources are much better utilized thanks to this pipeline parallelism. Moreover, instead of considering the slowest instruction when choosing cycle time, only the delay of the slowest pipeline stage has to be considered. Therefore, cycle time can be



(a) 1-cycle architecture.



(b) Simple 5-stages pipeline. As soon as the pipeline is filled (shown in white), all stages are busy processing different instructions every cycle.

Figure 1.1: Pipelined execution: x-axis shows time in cycles while y-axis shows sequential instructions.

decreased.

Both 1-cycle architecture and simple 5-stages pipeline are depicted in Figure 1.1. In this example, a single stage takes a single cycle, meaning that around five cycles are required to process a single instruction in the pipeline shown in (b). Nonetheless, as long as the pipeline is full, one instruction finishes execution each cycle, hence the throughput of one instruction per cycle is preserved. Moreover, since much less work has to be done in a single cycle, cycle time can be decreased as pipeline depth increases. As a result, performance increases, as illustrated in Figure 1.1 where the pipeline in (b) can execute five instructions in less time than the 1-cycle architecture in (a) can execute two, assuming the cycle time of the former is one fifth of the cycle time of the latter.

One should note, however, that as the pipeline depth increases, we start to observe diminishing returns. The reasons are twofold. First, very high frequency (above 4GHz) should be avoided to limit power consumption, hence deepening the pipeline to increase frequency quickly becomes impractical. Second, structural hazards inherent to the pipeline structure and the sequential processing of instructions makes it inefficient to go past 15/20 pipeline cycles. Those hazards will be described in the next Section.

1.2.1 Different Stages for Different Tasks

In the following paragraphs, we describe how the execution of an instruction can be broken down into several steps. Note that it is possible to break down these steps even further, or to add more steps depending on how the micro-architecture actually processes instructions. In particular, we will see in 1.3.3 that additional steps are required for out-of-sequence execution.

1.2.1.1 Instruction Fetch – IF

As previously mentioned, the instructions of the program to execute reside in global memory. Therefore, the current instruction to execute has to be brought into the processor to be processed. This requires accessing the memory hierarchy with the address contained in the *Program Counter* (PC) and updating the PC for the next cycle.

1.2.1.2 Instruction Decode – ID

The instruction retrieved by the Fetch stage is encoded in a binary format to occupy less space in memory. This binary word has to be expanded – *decoded* – into the control word that will drive the rest of the pipeline (e.g., register indexes to access the register file, control word of the functional unit, control word of the datapath multiplexers, etc.). In the pipeline we consider, Decode includes reading sources from the *Physical Register File* (PRF), although a later stage could be dedicated to it, or it could be done at *Execute*.

For now, we only consider the case of a RISC pipeline, but in the case of CISC, the semantics of many instructions are too complex to be handled by the stages following Decode as is. Therefore, the Decode stage is in charge of *cracking* such instructions into several micro-operations (μ-ops) that are more suited to the hardware. For instance, a load instruction may be cracked into a μ-op that computes the address from which to load the data from, and a μ-op that actually accesses the memory hierarchy.

1.2.1.3 Execute – EX

This stage actually computes the result of the instruction. In the case of load instructions, address calculation is done in this stage, while the access to the memory hierarchy will be done in the next stage, Memory.

To allow the execution of two dependent instructions back-to-back, the result of the first instruction is made available to the second instruction on the *bypass network*. In this particular processor, the bypass network spans from Execute to Writeback, at which point results are written to the PRF. For instance, let us

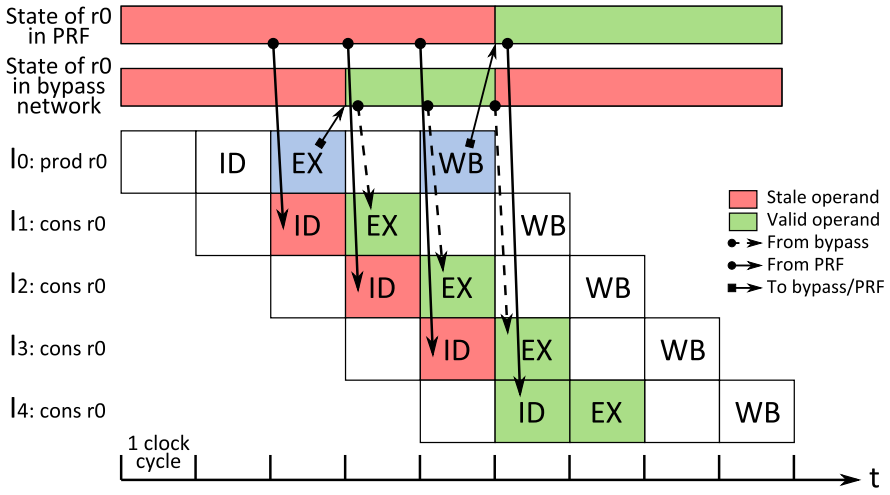


Figure 1.2: Origin of the operands of four sequential instructions dependent on a single instruction. Some stage labels are omitted for clarity.

consider Figure 1.2 where I_1, I_2, I_3 and I_4 all require the result of I_0. I_1 will catch its source r_0 on the bypass network since I_0 is in the Memory stage when I_1 is in Execute. I_2 will also catch r_0 on the network because I_0 is writing back its result to the PRF when I_2 is in Execute. Interestingly, I_3 will also catch its source on the network since when I_3 read its source registers in the Decode stage, I_0 was writing its result to the PRF, therefore, I_3 read the old value of the register. As a result, only I_4 will read r_0 from the PRF.

Without the bypass network, a dependent instruction would have to wait until the producer of its source left Writeback[2] before entering the Decode stage and reading it in the PRF. In our pipeline model, this would amount to a minimum delay of 3 cycles between any two dependent instructions. Therefore, the bypass network is a key component of pipelined processor performance.

1.2.1.4 Memory – MEM

In this stage, all instructions requiring access to the memory hierarchy proceed with said access. In modern processors, the memory hierarchy consists of several levels of cache memory intercalated between the pipeline and main memory. Cache memories contain a subset of main memory, that is, a datum that is in the cache also resides in main memory[3].

We would like to stress that contrarily to *scratchpads*, caches are not managed by the software and are therefore transparent to the programmer. Any memory

[2]Results are written to the PRF in Writeback.

[3]Note that a more recent version of the datum may reside in the cache and be waiting to be written back to main memory.

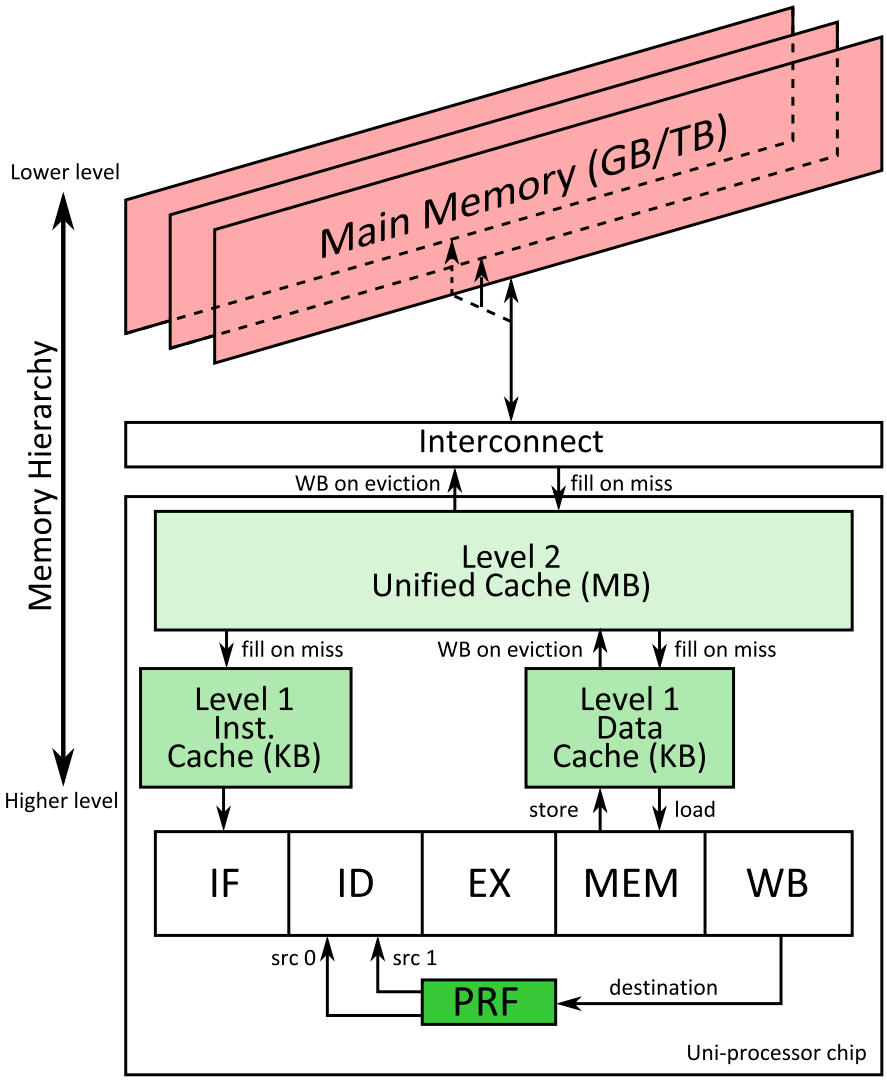


Figure 1.3: 3-level memory hierarchy. As the color goes from strong green to red, access latency increases. WB stands for writeback.

instruction accessing data that is not in the Level 1 (L1) cache (*miss*) will bring the required data from Level 2 (L2) cache. If said data is not in the L2, it will be brought from lower levels of the hierarchy. Since the L1 is generally small, cache space may have to be reclaimed to insert the new data. In this case, another piece of data is selected using a dedicated algorithm e.g., *Least Recently Used* (LRU) and written back to the L2 cache. Then it is overwritten by the new data in the L1 cache. Note that since we are putting the data evicted from the L1 in the L2, L2 space may also need to be reclaimed. In that case, the data evicted in the L2 is written back to the next lower level of the hierarchy (potentially main memory), and so on.

An organization with two levels[4] of cache is shown in Figure 1.3. The first level of cache is the smallest but also the fastest, and as we go down the hierarchy, size increases but so does latency. In particular, an access to the first cache level only requires a few processor cycles while going to main memory can takes up to several hundreds of processor cycles and stall the processor in the process. As a result, thanks to *temporal*[5] and *spatial*[6] *locality*, caches allow memory accesses to be very fast on average, while still providing the full storage capacity of main memory to the processor.

1.2.1.5 Writeback – WB

This stage is responsible for committing the computed result (or loaded value) to the PRF. As previously mentioned, results in flight between Execute and Writeback are made available to younger instructions on the bypass network. In this simple pipeline, once an instruction leaves the Writeback stage, its execution has completed and from the point of view of the software, the effect of the instruction on the architectural state is now visible.

1.2.2 Limitations

Control Hazards In general, branches are resolved at Execute. Therefore, an issue related to control flow dependencies begins to emerge when pipelining is used. Indeed, when a *conditional* branch leaves the Fetch stage, the processor does not know which instruction to fetch next because it does not know what direction the branch will take. As a result, the processor should wait for the branch to be resolved until it can continue fetching instructions. Therefore, *bubbles* (no operation) must be inserted in the pipeline until the branch outcome is known.

[4]Modern Chip Multi-Processors (CMP, a.k.a. multi-cores) usually possess a L3 cache shared between all cores. Its size ranges from a few MB to a few tens of MB.

[5]Recently used data is likely to be re-used soon.

[6]Data located close in memory to recently used data is likely to be used soon.

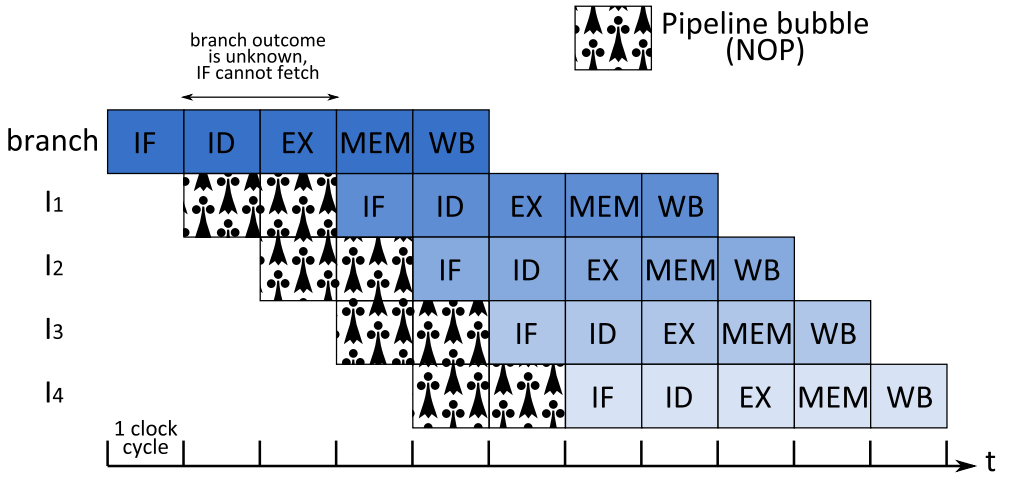


Figure 1.4: Control hazard in the pipeline: new instructions cannot enter the pipeline before the branch outcome is known.

In this example, this would mean stalling the pipeline for two cycles on each branch instruction, as illustrated in Figure 1.4. In general, a given pipeline would be stalled for *Fetch-to-Execute* cycles, which may amount to a few **tens** of cycles in some pipelines (e.g., Intel Pentium 4 has a 20/30-stages pipeline depending on the generation). Therefore, if deepening the pipeline allows to increase processor speed by decreasing the cycle time, the impact of control dependencies quickly becomes prevalent and should be addressed [SC02].

Data Hazards Although the bypass network increases efficiency by forwarding values to instructions as soon as they are produced, the baseline pipeline also suffers from data hazards. Data hazards are embodied by three types of register dependencies.

On the one hand, *true* data dependencies, also called *Read-After-Write* (*RAW*) dependencies cannot generally be circumvented as they are intrinsic to the algorithm that the program implements. For instance, in the expression $(1 + 2 * 3)$, the result of $2 * 3$ is always required before the addition can be done. Unfortunately, in the case of long latency operations such as a load missing in all cache levels, a *RAW* dependency between the load and its dependents entails that said dependents will be stalled for potentially hundreds of cycles.

On the other hand, there are two kind of *false* data dependencies. They both exist because of the limited number of architectural registers defined by the ISA (only **16** in x86_64). That is, if there were an infinite number of registers, they would not exist. First, *Write-after-Write* (*WAW*) hazards take place when two instructions plan to write to the same architectural register. In that case, writes must happen in program order. This is already enforced by the in-order

nature of execution in the simple pipeline model we consider. Second, *Write-after-Read* (*WAR*) hazards take place when a younger instruction plans to write to an architectural register than must be read by an older instruction. Once again, the read must take place before the write, and this is once again enforced in our case by executing instructions in-order. As a result, *WAW* and *WAR* hazards are a non-issue in processors executing instructions in program order.

Nonetheless, we will see in the next section that to mitigate the impact of *RAW* hazards, it is possible to execute instructions out-of-sequence (*out-of-order* execution). However, in this case, *WAW* and *WAR* hazards become a substantial hindrance to the efficient processing of instructions. Indeed, they forbid two independent instructions to be executed out-of-order simply because both instructions write to the same architectural register (case of *WAW* dependencies). The reason behind this *name collision* is that ISAs only define a few number of architectural registers. As a result, the compiler often does not have enough architectural registers (i.e., *names*) to attribute a different one to each instruction destination.

1.3 Advanced Pipelining

1.3.1 Branch Prediction

As mentioned in the previous section, the pipeline must introduce bubbles instead of fetching instructions when an unresolved branch is in flight. This is because neither the direction of the branch nor the address from which to fetch next should the branch be taken are known. To address this issue, *Branch Prediction* was introduced [Smi81]. The key idea is that when a branch is fetched, a predictor is accessed, and a predicted direction and branch target are retrieved.

Naturally, because of the speculative nature of branch prediction, it will be necessary to throw some instructions away when the prediction is found out to be wrong. In our 5-cycle pipeline where branches are resolved at *Execute*, this would translate in two instructions being thrown away on a branch misprediction. Fortunately, since execution is in-order, no rollback has to take place, i.e., the instructions on the wrong path did not incorrectly modify the architectural state. Moreover, in this particular pipeline, the cost of never predicting is similar to the cost of always mispredicting. Therefore, it follows that adding branch prediction would have a great impact on performance even if one branch out of two was mispredicted. This is actually true in general although mispredicting has a moderately higher cost than not predicting. Nonetheless, branch predictors are generally able to attain an accuracy in the range of 0.9 to 0.99 [SM06, Sez11b, cbp14], which makes branch prediction a key component of processor performance.

Although part of this thesis work makes use of recent advances in branch prediction, we do not detail the inner workings of existing branch predictors in

this Section. We will only describe the TAGE [SM06] branch predictor in Chapter 3 as it is required for our VTAGE value predictor. Regardless, the key idea is that given the address of a given *branch* instruction and some additional processor state (e.g., outcome of recent branches), the predictor is able to predict the branch direction. The target of the branch is predicted by a different – though equivalent in the input it takes – structure and can in some cases be computed at Decode instead of Execute.

1.3.2 Superscalar Execution

In a sequence of instructions, it is often the case that two or more neighboring instructions are independent. As a result, one can increase performance by processing those instructions in parallel. For instance, pipeline resources can be duplicated, allowing several instructions to flow in the pipeline in parallel, although still following program order. That is, by being able to fetch, decode, execute and writeback several instructions per cycle, throughput can be greatly increased, as long as instructions are independent. If not, only a single instruction can proceed while subsequent ones are stalled.

For instance, in Figure 1.5, six instructions can be executed by pairs in a pipeline having a superscalar degree of 2 (i.e., able to process two instructions concurrently). One should note that only 7 cycles are required to execute those instructions while the scalar pipeline of Figure 1.1 (b) would have required 10 cycles.

Unfortunately, superscalar execution has great hardware cost. Indeed, if increasing the width of the Fetch stage is fairly straightforward up to a few instructions (get more bytes from the instruction cache), the Decode stage is now in charge of decoding several instructions *and* resolving any dependencies between them to determine if they can proceed further together or if one must stall. Moreover, if functional units can simply be duplicated, the complexity of the bypass network in charge of providing operands on-the-fly grows quadratically with the superscalar degree. Similarly, the PRF must now handle 4 reads and 2 writes per cycle instead of 2 reads and 1 write in a scalar pipeline, assuming two sources and one destination per instruction. Therefore, ports must be added on the PRF, leading to a much higher area and power consumption since both grow super-linearly with the port count [ZK98]. The same stands for the L1 cache if one wants to enable several load instructions to proceed each cycle.

In addition, this implementation of a superscalar pipeline is very limited in how much it can exploit the independence between instructions. Indeed, it is not able to process concurrently two independent instructions that are a few instructions apart in the program. It can only process concurrently two neighboring instructions, because it executes in-order. This means that a long latency instruc-

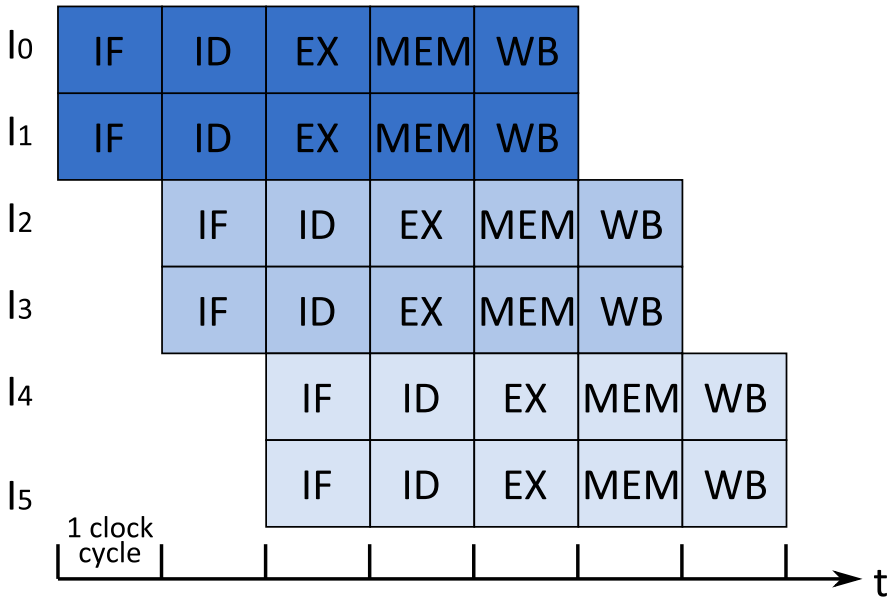


Figure 1.5: Degree 2 superscalar pipeline: two instructions can be processed by each stage each cycle as long as they are independent.

tion will still stall the pipeline even if an instruction further down the program is ready to execute. Even worse, due to the unpredictable behavior of program execution (e.g., cache misses, branch mispredictions) and the limited number of architectural registers, the compiler is not always able to reorder instructions in a way that can leverage this independence. As a result, increasing the superscalar degree while maintaining in-order execution is not cost-efficient. To circumvent this limitation, out-of-order execution [Tom67] was proposed and is widely used in modern processors.

1.3.3 Out-of-Order Execution

1.3.3.1 Key Idea

Although modern ISAs have sequential semantics, many instructions of a given program are independent on one another, meaning that they could be executed in any order. This phenomenon is referred to as *Instruction Level Parallelism* or ILP.

As a result, most current generation processors implement a feature known as *out-of-order* execution (OoO) that allows them to reorder instructions in hardware. To preserve the sequential semantics of the programs, instructions are still fetched in program order, and the modifications they apply on the architectural state are also made visible to the software in program order. It is only the exe-

cution that happens out-of-order, driven by *true* data dependencies (RAW) and resource availability.

To ensure that out-of-order provides both high performance and sequential semantics, numerous modifications must be made to our existing 5-stage pipeline.

1.3.3.2 Register Renaming

We previously presented the three types of data dependencies: RAW, WAW and WAR. We mentioned that WAW and WAR are false dependencies, and that in an in-order processor, they are a non-issue since in-order execution prohibits them by construction. However, in the context of out-of-order execution, these two types of data dependencies become a major hurdle to performance.

Indeed, consider two independent instructions I_0 and I_7 that belong to different dependency chains, but both write to register $r0$ simply because the compiler was not able to provide them with distinct architectural registers. As a result, the hardware should ensure that any instruction dependent on I_0 can get the correct value for that register before I_7 overwrites it. In general, this means that a substantial portion of the existing ILP will be inaccessible because the destination *names* of independent instructions *collide*. However, if the compiler had had more architectural registers to allocate, the previous dependency would have disappeared. Unfortunately, this would entail modifying the ISA.

A solution to solve this issue is to provide more registers in hardware but keep the number of registers visible to the compiler the same. This way, the ISA does not change. Then, the hardware can dynamically rename *architectural* registers to *physical* registers residing in the PRF. In the previous example, I_0 could see its architectural destination register, $ar0$ (arch. reg 0), renamed to $pr0$ (physical register 0). Conversely, I_7 could see its architectural destination register, also $ar0$, renamed to $pr1$. As a result, both instructions could be executed in any order without overwriting each other's destination register. The reasoning is exactly the same for WAR dependencies.

To track register mappings, a possibility is to use a *Rename Map* and a *Free List*. The Rename Map gives the current mapping of a given architectural register to a physical register. The Free List keeps track of physical registers that are not mapped to any architectural register, and is therefore in charge of providing physical registers to new instructions entering the pipeline. These two steps, *source renaming* and *destination renaming* are depicted in Figure 1.6.

Note however that the mappings in the Rename Map are speculative by nature since renamed instructions may not actually be retained if, for instance, an older branch has been mispredicted. Therefore, a logical view of the renaming process includes a *Commit Rename Map* that contains the non-speculative mappings, i.e., the current software-visible architectural state of the processor. When an

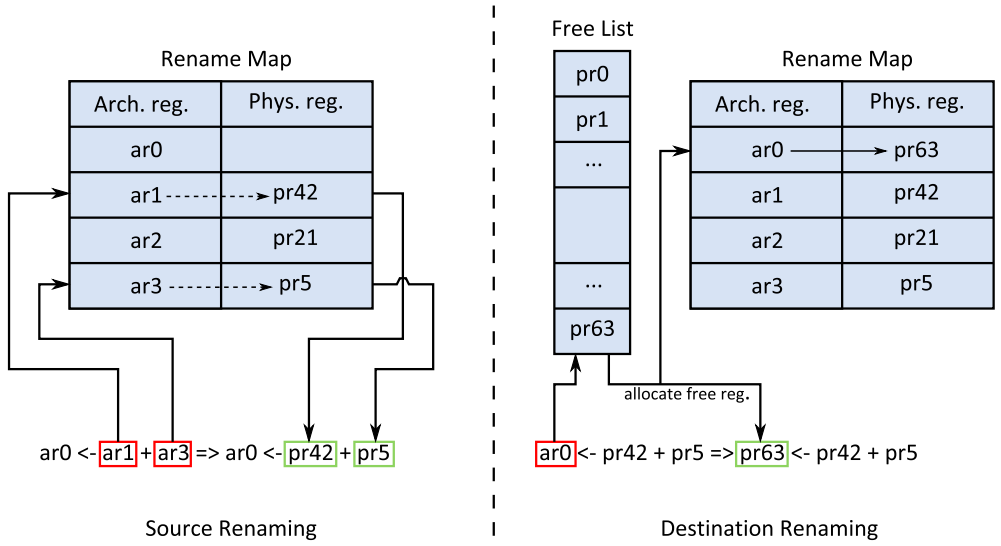


Figure 1.6: Register renaming. Left: Both instruction sources are renamed by looking up the Rename Table. Right: The instruction destination is renamed by mapping a free physical register to it.

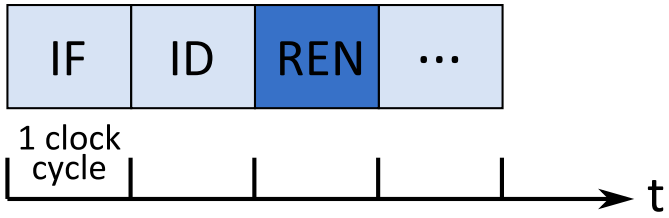


Figure 1.7: Introduction of register renaming in the pipeline. The new stage, Rename, is shown in dark blue.

instruction finishes its execution and is removed from the pipeline, the mapping of its architectural destination register is now logically part of the Commit Rename Map.

A physical register is put back in the Free List once a younger instruction mapped to the same architectural register (i.e., that has a more recent mapping) leaves the pipeline.

Rename requires its own pipeline stage(s), and is usually intercalated between Decode and the rest of the pipeline, as shown in Figure 1.7. Thus, it is not possible to read the sources of an instruction in the Decode stage anymore. Moreover, to enforce the sequential semantics of the program, Rename must happen in program order, as Fetch and Decode. Lastly, note that as the superscalar degree increases, the hardware required to rename registers becomes slower as well as more complex [PJS97], since among a group of instruction being renamed in the

same cycle, some may depend on others. In that case, the younger instructions require the destination mappings of the older instructions to rename their sources, but those mappings are being generated. An increased superscalar degree also puts pressure on the Rename Table since it must support that many more reads and writes each cycle.

1.3.3.3 In-order Dispatch and Commit

To enable out-of-order execution, instruction scheduling must be decoupled from in-order instruction fetching and in-order instruction retirement. This decoupling is achieved with two structures: The *Reorder Buffer* (ROB) and the *Instruction Queue* (IQ) or *Scheduler*.

The first structure contains all the instructions that have been fetched so far but have not completed their execution. The ROB is a *First In First Out* (FIFO) queue in which instructions are added at the tail and removed at the head. Therefore, the visible architectural state of the processor (memory state and Commit Rename Map) corresponds to the last instruction that was removed from the ROB.[7] Moreover, when a branch misprediction is detected when a branch flows through Execute, the ROB allows to remove all instructions on the wrong path (i.e., younger than the branch) while keeping older instructions. As a result, the ROB is really a speculative frame representing the near future state of the program. In other words, it contains instructions whose side-effects will soon be visible by the software, as long as everything goes *well* (i.e., control flow is correct, no interruption arrives, no exception is raised).

Conversely, the IQ is a fully associative buffer from which instructions are selected for execution based on operand readiness each cycle. As a result, the IQ is the only structure where instructions can appear and be processed out-of-order.

As instructions are executed, they advance towards the head of the ROB where their modifications to the architectural state will be committed. In particular, once an instruction reaches the head of the ROB, the last stage of the pipeline, the *Commit* stage is in charge of checking if the instruction can be removed, as illustrated in the right part of Figure 1.8. An instruction cannot commit if, for instance, it has been not executed.

Both structures are populated in-order after instruction registers have been renamed, as shown in the left part of Figure 1.8. We refer to this step as instruction dispatch, and to the corresponding pipeline stage as *Dispatch*. Similarly, instructions are removed from the ROB in program order by the Commit stage, to enforce sequential semantics. Commit is also responsible for handling any exception. These two additional stages are depicted in Figure 1.9. The four first stages in the Figure form what is commonly referred to as the pipeline *frontend*.

[7] With a few exceptions in the multi-core case, depending on the memory consistency model.

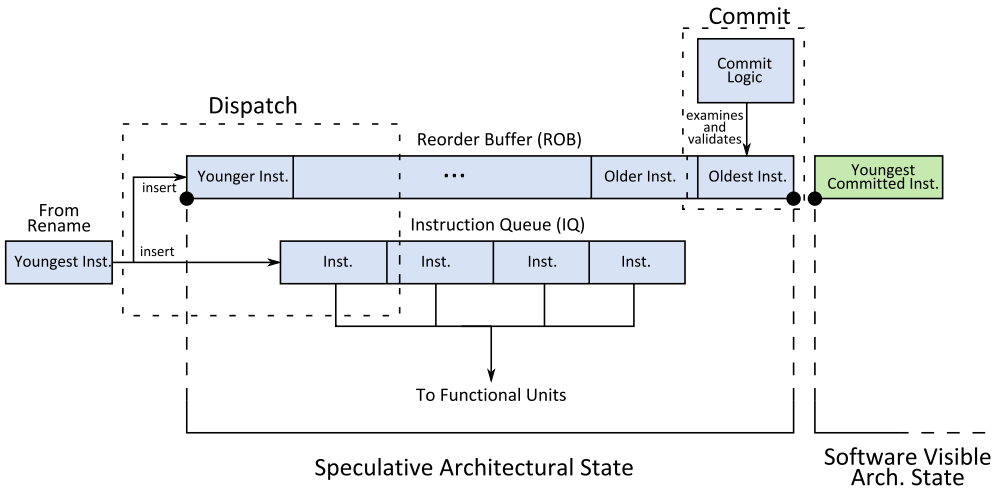


Figure 1.8: Operation of the Dispatch and Commit stages. Left: Instructions from Rename are inserted in the ROB and IQ by Dispatch. Right: In the meantime, the Commit logic examines the oldest instruction (head of ROB) and removes it if it has finished executing.

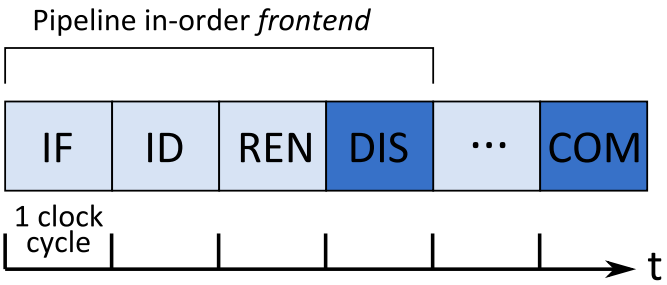


Figure 1.9: Introduction of instruction dispatch and retirement in the pipeline. The new stages, Dispatch and Commit, are shown in dark blue.

1.3.3.4 Out-of-Order Issue, Execute and Writeback

Operation The IQ is responsible for instruction scheduling or *issuing*. Each cycle, logic scans the IQ for n instructions[8] whose sources are ready i.e., in the PRF or the bypass network. These *selected* instructions are then *issued* to the functional units for execution, with no concern for their ordering. To carry out scheduling, a pipeline stage, *Issue*, is required.

Before being executed on the functional units (FUs), issued instructions will read their sources from either the PRF or the bypass network in a dedicated stage we refer to as *Register/Bypass Read* (RBR). In the two subsequent stages, Execute and Writeback, selected instructions will respectively execute and write their result to the PRF. Instruction processing by this four stages is depicted in Figure 1.10.

Note that if two dependent instructions I_0 and I_1 execute back-to-back, then the destination of I_0 (respectively source of I_1) is not available on the multiplexer of the *RBR* stage when I_1 is in *RBR*. Consequently, in this case, the bypass network is able to forward the result of I_0 from the output of the FU used by I_0 during the previous cycle to the input of the FU used by I_1 when I_1 enters Execute. The bypass network is therefore *multi-level*. Also consider that a hardware implementation may not implement several levels of multiplexers as in Figure 1.10, but a single one where results are buffered for a few cycles and tagged with the physical register they correspond to. In that case, an instruction may check the network with its source operands identifiers to get their data from the network.

These four stages form the out-of-order engine of the processor, and are decoupled from the in-order frontend and in-order commit by the IQ itself. The resulting 9-stage pipeline is shown in Figure 1.11.

Complexity It should be mentioned that the logic used in the out-of-order engine is complex and power hungry. For instance, assuming an issue width of n, then up to $2n$ registers may have to be read from the PRF and up to n may have to be written, in the same cycle. Moreover, an increased issue-width usually implies a larger instruction window, hence more registers. As a result, PRF area and power consumption grow quadratically with the issue width [ZK98].

Similarly, to determine operand readiness (*Wakeup* phase), each entry in the IQ has to monitor the name of the registers whose value recently became available. That is, each source of each IQ entry must be compared against n physical register identifiers each cycle. Assuming $n = 8$, 8-bit identifiers (i.e., 256 physical registers), 2 sources per IQ entry, and a 60-entry IQ, this amounts to 960 8-bit comparisons per cycle. Only when this comparisons have taken place can the

[8] n is the *issue-width*, which does not necessarily correspond to the superscalar degree.

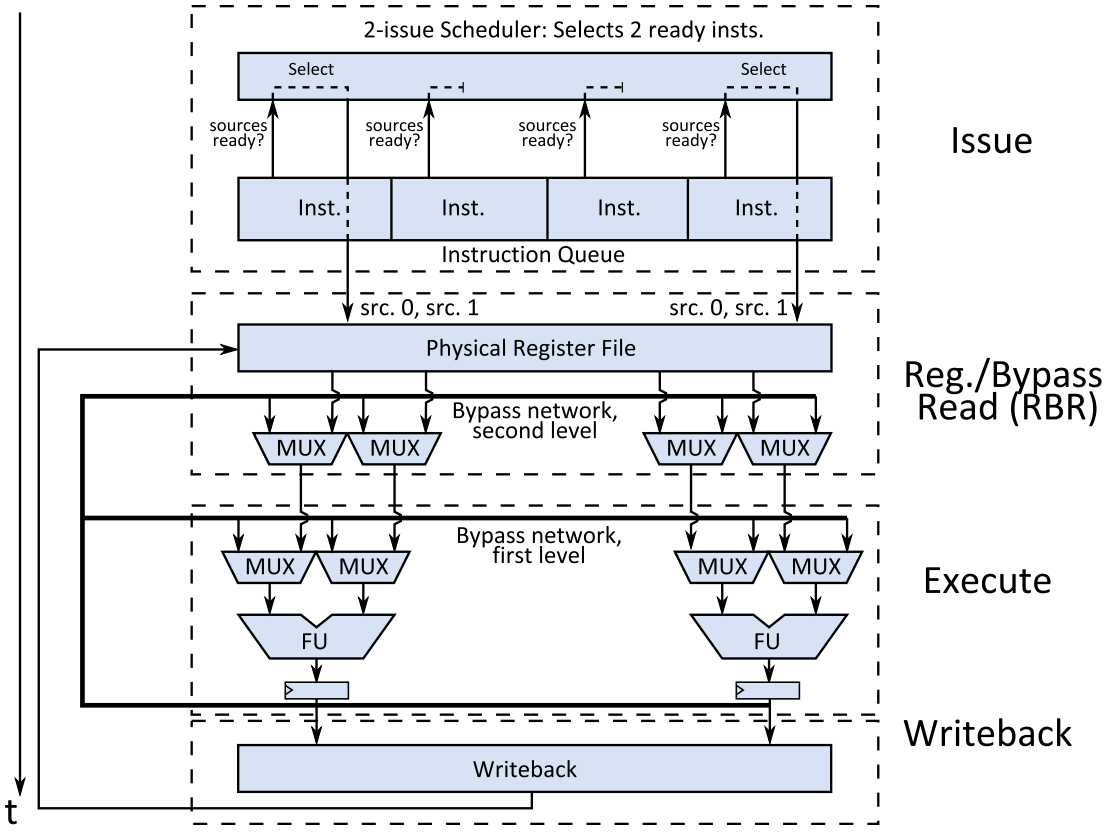


Figure 1.10: Operation of the out-of-order execution engine. The Scheduler is in charge of *selecting* 2 ready instructions to *issue* each cycle. Selected instructions collect their sources in the next stage (*RBR*) or in Execute in some cases, then execute on the functional units (FUs). Finally, instructions write their results to the PRF.

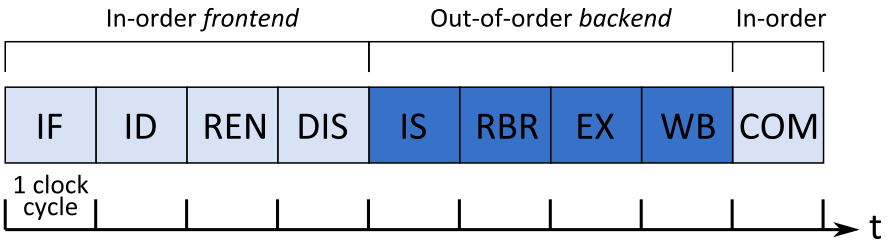


Figure 1.11: Introduction of the out-of-order execution engine in the pipeline. New stages are shown in dark blue. IS: Issue, RBR: Register/Bypass Read, EX: Execute, WB: Writeback.

scheduler select (*Select* phase) ready instructions. Selection includes reading all the *ready* bits from the IQ and resolving any structural hazards (e.g., not enough available functional units). Once instructions are selected, their destination identifiers are broadcast to the IQ to prepare the next Wakeup phase. Due to its hardware complexity and 2-phase behavior, the Issue stage tends to be slow and its power consumption quite high (up to 18% of the processor [GBJ98]).

Moreover, to enable back-to-back execution of two dependent instructions, Wakeup & Select must take place in a single cycle. As a result, the delay of the Issue stage cannot easily be addressed by pipelining Wakeup & Select. Doing so would result in a bubble always being introduced between two dependent instructions, as shown in Figure 1.12. Although Stark et al. [SBP00] proposed a design where instructions are speculatively selected without the guarantee that their operands are ready to remove this bubble, common wisdom suggests that pipelining this stage is very hard in general. Consequently, the issue-width and IQ size have only marginally increased over processor generations to avoid increasing the whole pipeline cycle-time by increasing the delay of the Issue stage.

A last difficulty lies with the existence of a delay between the *decision* to execute the instructions (at Issue) and its actual execution (at Execute), modern processors employ speculative scheduling for instructions depending on variable latency instructions (e.g., loads) [SGC01b, KMW98]. That is, a dependent on a load is generally issued assuming that the load will hit, so that both can execute back-to-back, without paying the issue-to-execute delay for each instruction (RBR stage in our case). However, when speculation is wrong (e.g., the load misses or is delayed by a bank conflict at the L1 cache), then the dependent goes through the functional unit with a random source operand value. The pipeline detects this and replays the instruction and its own dependents if there are some [SGC01b, KMW98]. This is another example of the complexity that is involved in out-of-order execution. Although this thesis focuses on enhancing performance through Value Prediction, the author has also published work on how to reduce the number of replays due to incorrect scheduling in the International Symposium on Computer Architecture (ISCA 2015) [PSM+15].

In a nutshell, the out-of-order engine is a key contributor to power consumption and overall complexity in modern processors. Any means to reduce one or both would be of great help and has generally been a topic of research in the computer architecture community [KL03, EA02, TP08, FCJV97]. Similarly, improving sequential performance without increasing complexity in the out-of-order engine is also a key requirement in modern processor design.

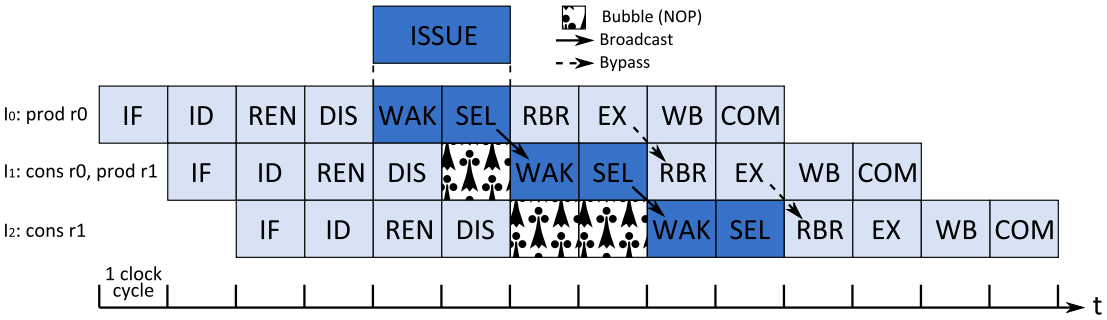


Figure 1.12: Pipelining Issue (Wakeup & Select) over two cycles introduces a bubble between any two dependent instructions.

1.3.3.5 The Particular Case of Memory Instructions

Readers may have noticed that the Memory stage has been removed from Figure 1.11. The reason is that because instruction sequencing and instruction scheduling/execution are decoupled, the Execute stage can take up as many cycles as necessary without stalling subsequent instructions, thanks to out-of-order execution. As a result, we embed the Memory stage inside the Execute stage.

Regardless, in its current state, our out-of-order model is incorrect, for two reasons. First, if loads can generally be executed speculatively (i.e., on the wrong control path) without threatening correctness, the same is not true for store instructions. Indeed, if a value is committed to memory only to find out that the store instruction must be thrown away because an older branch was mispredicted, then execution is incorrect.

Second, if RAW dependencies between registers can easily be enforced, RAW memory dependencies cannot. The reason is that the addresses accessed by memory instructions are generally not known until Execute. That is, an older store may write to a location that is accessed by a younger load, but there is no way to know it before both instructions have computed their respective addresses. If the load is selected for execution before the store, then it will obtain a stale value and break the sequential semantics of the program.

Therefore, it follows that the hardware should provide two guarantees:

1. Store instructions may only write their data to memory once they are guaranteed to be on the correct control path, i.e., after they have been committed.

2. Sequential ordering between dependent memory operations must be enforced[9].

[9]With regard to the memory consistency model provided by the ISA.

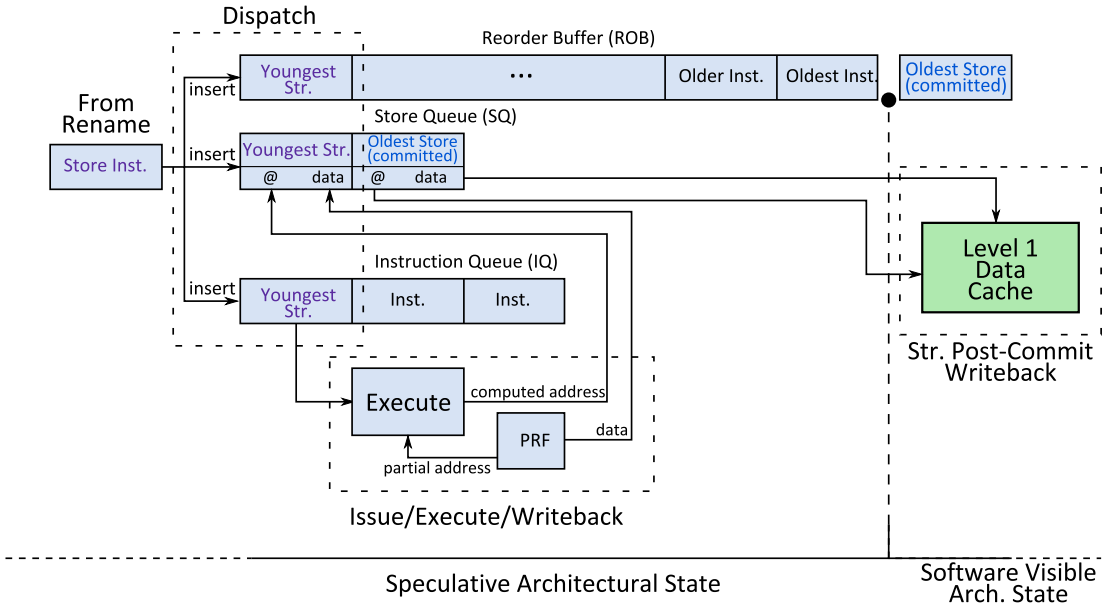


Figure 1.13: *Store Queue* operation. Left: New store instructions are put at the tail in Dispatch. Bottom: During Execute, the address is computed and put in the SQ with the corresponding data. Right: After Commit, the store is written back to the data cache when it arrives at the head of the SQ.

Post-Commit Writeback Since stores may only access the data cache after having committed, another structure is required to keep track of pending stores. Thus, current processors provide a third structure, the *Store Queue* (SQ), that tracks both inflight stores (i.e., speculative) and pending stores (i.e., committed but waiting to write their data to the data cache). The operation of the SQ is illustrated in Figure 1.13.

The left part of the Figure shows that the SQ is populated at Dispatch, as the ROB and IQ. As the ROB, it keeps track of stores in a FIFO fashion, to enforce sequential ordering of memory writes. An entry in the SQ has two main fields: *address* and *data*. Keeping data in the SQ is required as once a store commits, there is no guarantee that the data to be written will remain in the PRF until the cache is actually accessed. Keeping the computed address is required to enable a feature known as *Store-to-Load Forwarding* (STLF), which plays a key role in enforcing sequential semantics for memory instructions. It will be described in the next paragraph.

Regardless, with this mechanism, a store instruction is issued in the out-of-order engine, as soon as its sources (partial address and data) are ready. During Execute, the data cache is not accessed. Rather, the address is computed and both computed address and data are copied to the SQ entry, as depicted in the

bottom of Figure 1.13.

Once a store has committed, it becomes eligible to be written back to the data cache. In particular, each cycle, the processor writes back stores at the head of the SQ to the data cache, if a cache port is available. Then, it clears their SQ entry. This is shown in the right corner of Figure 1.13.

Astute readers may have noticed that some stores may not have written back their data to the cache, yet have committed. This committed memory write should be part of the software-visible architectural state of the processor. However, in this particular pipeline instance, software visibility of the write is not guaranteed since the data may still be in the SQ when a subsequent load to the same address executes. This is addressed by Store-To-Load Forwarding (STLF).

Store-to-Load Forwarding STLF consists in probing all SQ entries containing an older store with the load address every time a load executes. If a match is found, then the data in the SQ can be forwarded to the load destination register. Such a mechanism is illustrated in Figure 1.14.

With STLF, any committed store waiting for its turn to write its data in the cache can still supply its data to younger load instructions. In effect, any pending memory write becomes visible to younger instructions, guaranteeing that committed stores are part of the software-visible architectural state. Note that STLF must not be limited to committed stores: any store is allowed match a given load address as long as it is older. In particular, in Figure 1.14, an uncommitted store provides the data.

Yet, correctness is still not guaranteed. Indeed, consider the case where the load in Figure 1.14 should actually match the younger older store (2^{nd} entry), but the address collision remains undetected since that store has not executed yet. In that case, execution is incorrect because the hardware was not able to enforce and even *detect* the Read-after-Write dependency existing between the two memory instructions.

Out-of-Order Execution of Memory Operations To remedy the memory ordering violation example described in the last paragraph and provide the second guarantee discussed in 1.3.3.5, the hardware can simply forbid memory instructions to issue out-of-order. However, this would defeat the purpose of out-of-order execution and greatly hinder performance in the process.

Thus, a more efficient solution would be to allow stores being executed to check that all younger loads either access a different address or have not executed yet. To do so, another structure is required to track the addresses of inflight loads. Specifically, current processors implement a *Load Queue* (LQ), which is very similar to the Store Queue, except for the instructions it tracks and the fact

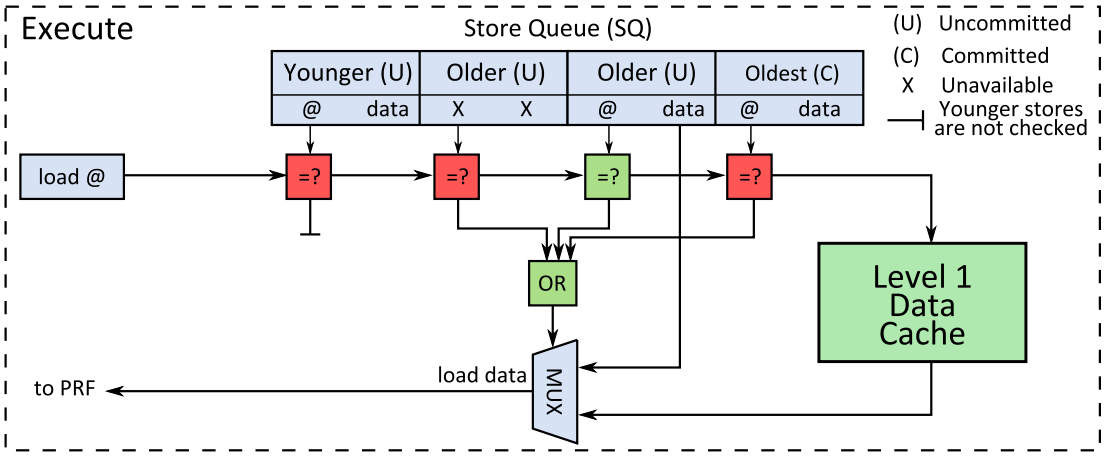


Figure 1.14: *Store-to-Load Forwarding*: When a load executes, all SQ entries are checked for a store containing a more recent version of the data. The arbitration logic in case several SQ entries match is omitted for clarity (the youngest older matching store provides the data).

that entries are released at Commit, as ROB entries. Each LQ entry tracks the address of its associated load instruction.

Then, when a store executes, it scans the LQ for younger loads to the same address, as illustrated in Figure 1.15. This allows the detection of any incorrect behavior. In that event, the most simple way to recover is to *squash* (i.e., remove) all instructions younger than the load instruction that matched (including the load). Then, the Fetch stage restarts from the load instruction. With such mechanism, the penalty is similar to a branch misprediction. As a result, the number of loads that incorrectly execute before conflicting stores should be minimized, but not at the cost of stalling a load until **all** older stores have executed.

Memory Dependency Prediction Because the SQ acts as a memory renamer, stores can issue out-of-order even if a Write-after-Write dependency exists between them. The reason is that each SQ entry provides a *physical name* where to put the data. The SQ addresses Write-after-Read dependencies through the same renaming mechanism. In both cases, correctness is guaranteed by writing back to the cache in-order, after Commit.

However, precise tracking of Read-after-Write dependencies between memory instructions prior to their execution is impossible in general. Therefore, inflight memory locations are renamed in the SQ, but dependency linking is done at Execute by checking the LQ or SQ for conflicting instructions. As a result, depending on the aggressiveness of the scheduler, RAW dependencies can either lead to many memory order violations (by assuming that loads never depend on

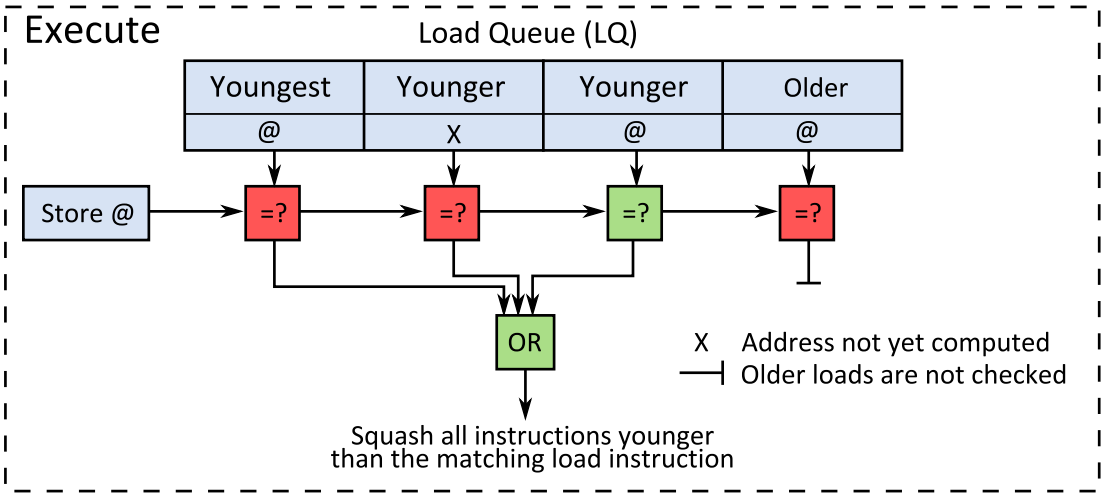


Figure 1.15: Memory ordering check: When a store executes, all younger inflight loads addresses are compared against the store address. Here, a younger load matches, meaning that the instruction has loaded stale data. All instructions younger than the load (including the load) will be removed from the pipeline and re-fetched. The arbitration logic in case several LQ entries match is omitted for clarity (squashing starts from the oldest matching load).

stores) or to a loss in potential performance (by assuming that a load depends on **all** older un-executed stores). Therefore, to enable efficient out-of-order execution of memory operations, current processors usually implement a *Memory Dependency Predictor*. Indeed, instances of a given static load instruction often depend on instances of a single static store instruction [CE98]. As a result, a structure can be used to remember the past behavior of memory instructions and predict that a given dynamic load will repeat the behavior of past instances of the same static instruction.

We do not detail any specific algorithm of memory dependency prediction, but we note that with the *Store Sets* predictor [CE98], most of the potential of executing memory instructions out-of-order is retained while minimizing the number of ordering violations.

1.3.3.6 Summary and Limitations

Figure 1.16 gives a higher-level view of a modern out-of-order pipeline, starting from the in-order frontend on the left, then going to the out-of-order engine in the middle, to finish with in-order instruction removal on the right. Although some links between structures are omitted for clarity (e.g., Commit updates the branch predictor and the memory dependency predictor, Execute can squash the pipeline on branch mispredictions and memory order violations), the Figure gives

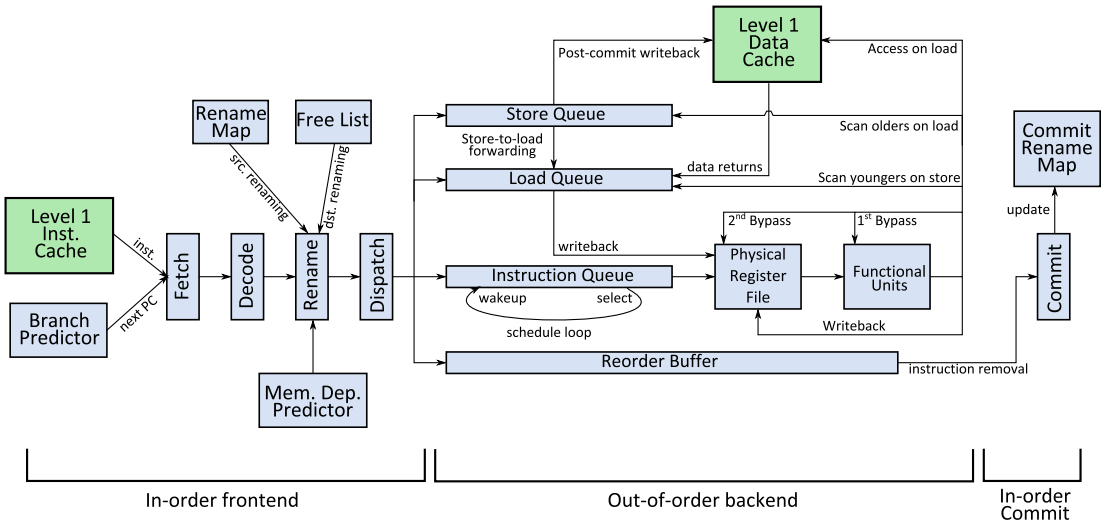


Figure 1.16: Overview of a modern out-of-order superscalar processor. Instructions flow from left to right.

a good overview of the interactions between the different hardware components.

The main conclusion we can draw is that even at this level of abstraction, complexity is very high and great care has to be taken to handle corner cases of out-of-order execution. Nonetheless, this organization has proven highly effective due to its high sequential performance.

However, if raw performance comes from the out-of-order engine, actual performance come from several mechanisms mitigating dependencies between instructions and hardware limitations. Those mechanisms allow the out-of-order engine to leverage the underlying ILP of the program. For instance, branch prediction can mitigate control dependencies and keep the instruction window occupancy high. Register renaming removes WAW and WAR dependencies for registers, while the limited memory renaming capacity provided by the SQ removes those dependencies for memory instructions. Finally, memory dependency prediction enables out-of-order execution of load instructions at the cost of occasional ordering violations.

Yet, although out-of-order execution allows to partially hide them, RAW dependencies are always strictly enforced. This is very natural, as they express the semantics of the program. Unfortunately, they still limit ILP by serializing the execution of dependent instructions. Devising a way to break RAW dependencies could lead to a direct increase in sequential performance as it would potentially shorten the *critical path* (longest chain of dependent instructions) of the program.

In addition, the usual way to improve sequential performance – increasing the aggressiveness of the out-of-order engine – costs great amounts of transistors and

increases power consumption substantially. As a result, sequential performance has only increased marginally with processor generations. Breaking RAW dependencies would increase sequential performance by uncovering more ILP without paying the cost of a more aggressive out-of-order engine, rendering it even more attractive a solution.

This thesis work revisits *Value Prediction* (VP), which goal is to predict the result of dynamic instructions. That is, it allows a dependent on a predicted instruction to execute with its producer concurrently by using the prediction as a source operand. Hence, the RAW dependency is effectively broken. To enforce correctness, wrong predictions must be detected and the pipeline state must be rolled-back, much like in the case of a branch misprediction. Therefore, in addition to a realistic but efficient prediction algorithm, a practical implementation of VP should provide a complexity-effective way to detect mispredictions and recover from them. It should also increase sequential performance enough to be worth its cost, and said cost should be made as small as possible.

Chapter 2

Value Prediction as a Way to Improve Sequential Performance

As mentioned in the previous Chapter, Value Prediction (VP) speculates on the result of instructions to break RAW dependencies and artificially increase ILP. This is illustrated in Figure 2.1, where the sequential execution of a chain of dependency comprising four instructions is depicted, with and without VP. VP was introduced independently by Lipasti et al. [LWS96, LS96] and Gabbay and Mendelson [GM96]. It leverages the low dynamic range exhibited by the result stream of some instructions, also known as *value locality*. In other words, instances of the same static instruction often produce the same result, or results that fall within an easily recognizable pattern (e.g., *strided*).

Value Prediction has been an active topic of research in the 90's. As a result, many prediction algorithms have been devised, as well as other mechanisms leveraging VP. For instance, Aragon et al. [AGGG01] and Gonzalez and Gonzalez [GG99] use it to improve branch prediction, while other works use it to predict memory addresses and trigger prefetches [EV93, GG97]. Burtscher et al. even consider the use of value prediction to compress execution traces more efficiently [BJ03, Bur04]. However, due to an ambient skepticism regarding the feasibility of an actual implementation, the focus of the community shifted away from VP in the late 90's. The main reason being the assumption that VP introduces complexity in critical stages of the pipeline: prediction in the frontend, and validation and recovery in the backend. In particular, we already mentioned the high complexity of the out-of-order engine, therefore, complexifying it further with VP is not an attractive proposition.

In the common approach, a predictor is added to the frontend and accessed on a per-instruction basis. This predictor provides a prediction that acts as a temporary result for the instruction and can therefore be sourced by dependents before the *actual* result is computed. When the predicted instructions leaves the

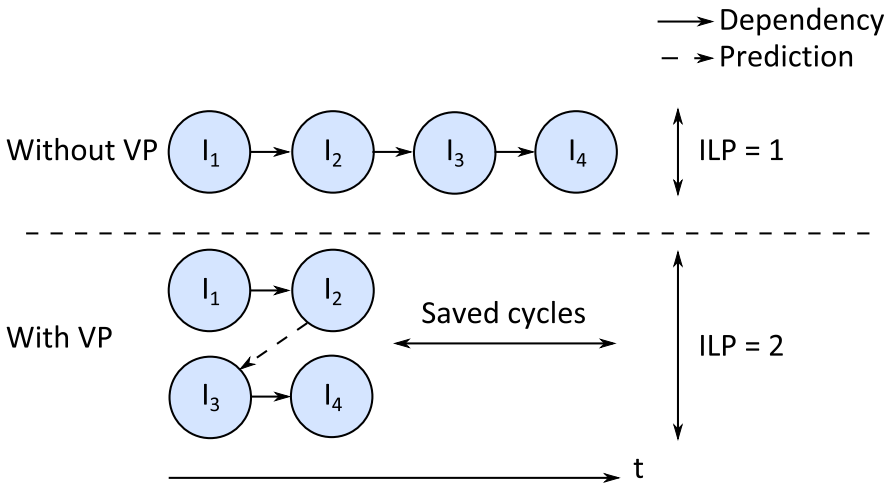


Figure 2.1: Impact of Value Prediction on the ILP that the processor can extract from a given program.

functional units, its prediction is validated against the actual result, and recovery takes place if necessary. Finally, when the instruction leaves the pipeline, its result is sent to the predictor to update it according to the prediction algorithm.

Arguably, the key component of a value prediction micro-architecture is the value predictor. Sazeides and Smith define two main types of predictors: *computational* and *context-based* [SS97a]. We detail the main representatives of both categories in the following paragraphs as well as the way they are updated.

2.1 Hardware Prediction Algorithms

2.1.1 Computational Prediction

Computational predictors *compute* a prediction by applying a function to the result of the previous instance, or previous instances. Therefore, they are able to generate values that have not yet been seen in the value stream, depending on the applied function.

Last Value Prediction The *Last Value Predictor* (LVP), as its name suggests, predicts that the result of an instruction is the result produced by the previous instance of the same static instruction. In other words, it applies the identity function to the last result. It was introduced by Lipasti and Shen [LS96] and may be the most simple prediction generation mechanism. Nonetheless, it is able to improve performance by 4.5% on average in the framework used in [LS96].

This predictor is shown in Figure 2.2. It is a direct-mapped table addressed by

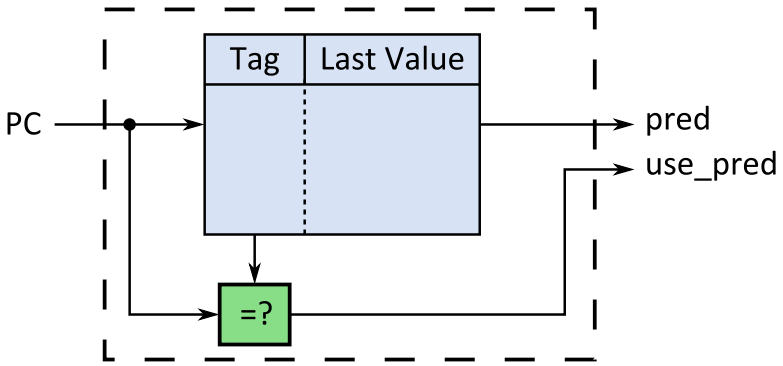


Figure 2.2: Last Value Predictor. The prediction for a given instruction is the value produced by the previous instance.

the lower bits of the instruction PC. Then, the higher bits of the PC are compared against the tag contained in the entry. On a tag match, the value contained in the entry is used as a prediction for the instruction. On a miss, the instruction is not predicted. To reduce conflicts, it is possible to use a set-associative structure.

Update is straightforward: when the instruction retires, it writes its actual result into the predictor table and updates the tag if necessary.

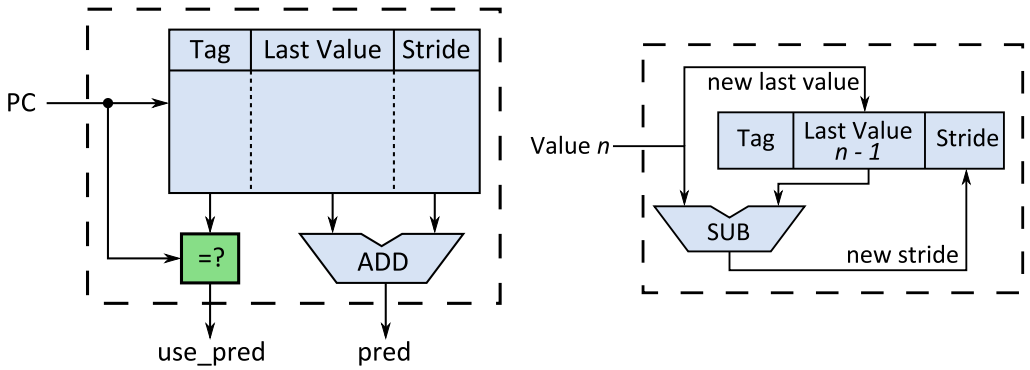
2.1.1.1 Stride-based Predictors

Stride and 2-delta Stride Prediction Stride prediction was introduced in [GM96] and leverages the fact that many computations exhibit strided patterns. For instance, the addresses of an array that is read sequentially in a loop typically have a strided relation. More generally, computations based on the loop induction variable often exhibit a strided pattern. As a result, the function used by this predictor to *compute* the prediction is the addition of a constant, the stride, to the last value.

Contrarily to LVP, the Stride predictor is able to generate values that have not yet been seen in the value stream by quickly latching to a pattern and generating the subsequent values of the pattern before they are observed.

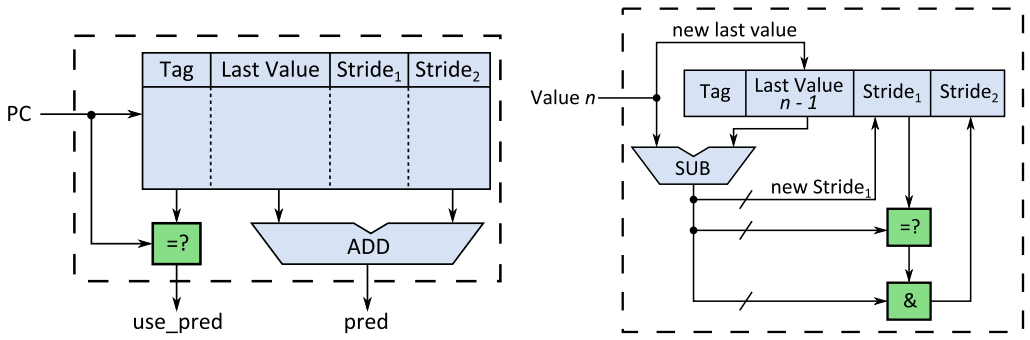
Figure 2.3 (a) details a simple Stride predictor. It is very similar to LVP, except each entry also contains a stride. To compute the prediction for instance$_n$, the result of instance$_{n-1}$ is added to the difference between the result of instance$_{n-1}$ and that of instance$_{n-2}$. The update process is depicted in Figure 2.3 (b). The stride is updated when the instance retires, by computing the difference between the last value currently in the entry $(n-1)$ and the value produced by the current instance, n. Then, the value of instance$_n$ replaces the value of instance$_{n-1}$ in the entry.

A variation of the baseline Stride predictor, *2-delta*, is described in [EV93].



(a) Prediction. (b) Predictor entry update.

Figure 2.3: (a) Simple Stride Predictor. The prediction for a given instance$_n$ is the value produced by instance$_{n-1}$ plus the given stride. (b) Entry update: the new stride is computed by subtracting result$_n$ and result$_{n-1}$. Then, result$_n$ replaces result$_{n-1}$ in the *Last Value* field of the entry.



(a) Prediction. (b) Predictor entry update.

Figure 2.4: (a) 2-delta Stride Predictor. The prediction for a given instance$_n$ is the value produced by instance$_{n-1}$ plus stride$_2$. (b) Entry update: the new stride is computed by subtracting result$_n$ and result$_{n-1}$. If the new stride equals Stride$_1$, then it replaces Stride$_1$ and Stride$_2$, otherwise, it replaces only Stride$_1$. Then, result$_n$ replaces result$_{n-1}$ in the *Last Value* field of the entry.

Instead of storing a single stride in each predictor entry, this predictor keeps two. The first stride is always updated, but the second stride is updated only if the first stride has not changed after the update. Only the second stride is used to generate the prediction.

The key insight behind 2-delta is that if the stride is always updated, then there will be at least two mispredictions per instance of a loop: one when the loop starts again, and one for the second iteration of the loop. By storing two strides to provide hysteresis, only one misprediction takes place in that case: when the loop starts again.

Per-Path Stride Prediction Another predictor using a stride-based algorithm is the *Per-Path Stride* (PS) predictor of Nakra et al. [NGS99]. The idea is to use the global branch history to select the stride to use for this particular instance of a static instruction, by hashing it with the PC to index the table containing the strides. The PS predictor is depicted in Figure 2.5.

Selecting the last value depending on the global branch history is also a possibility, but is less space efficient as instead of having a single last value for each static instruction, we might have several values relating to different dynamic streams of a single static instruction. However, depending on the number of global branch history bits that are used, selecting the last value with the global branch history may be the only way to capture certain patterns [NGS99]. As a result, the authors also propose a predictor where both stride and last value are selected using a hash of the branch history and the PC of the instruction to predict.

Regardless, in both cases, only a few bits of the global history are used in [NGS99] (2 bits), hence correlation is limited.

gDiff Prediction The gDiff predictor of Zhou et al. takes the approach of global correlation [ZFC03]. That is, instead of trying to detect a strided pattern within the stream of values produced by a given static instruction, it tries to correlate the result of different neighbouring dynamic instructions. This predictor is depicted in Figure 2.6. On the left, a PC-indexed prediction table tracks, for each static instruction, the recently observed differences between the result of the last instance of this static instruction, and the results of dynamic instructions immediately preceding this instance. Another field, *distance*, indicates how many dynamic instructions were observed between the instruction to predict and the one it correlates with. That is, the *distance* field indicates what difference to use to generate the prediction.

On the right, the *Global Value Queue* (GVQ) is a window of in-flight results. When a result becomes available, the hardware computes the differences between this result and all the other values present in the GVQ. If one of these differences

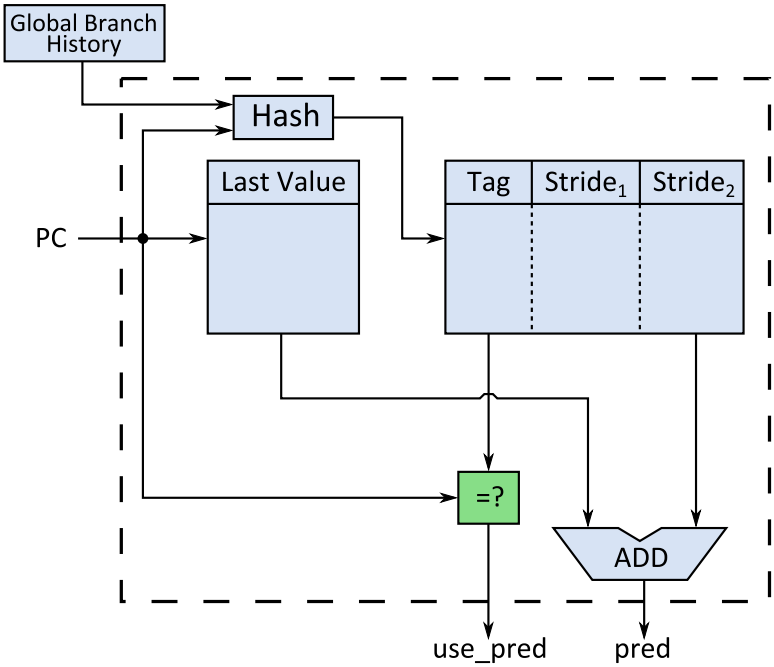


Figure 2.5: Per-path Stride (PS) predictor. The prediction for a given instruction is the value produced by the previous instance plus a stride chosen depending on the current control flow. Updating PS is similar to updating the 2-delta Stride predictor.

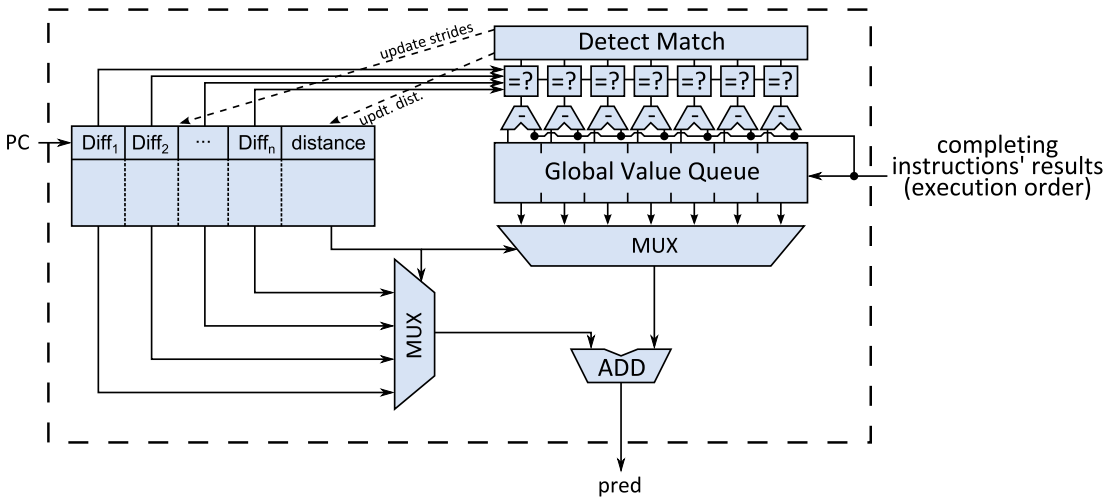


Figure 2.6: Order n gDiff Predictor. The prediction is generated by adding the $distance^{th}$ stride to the $distance^{th}$ youngest value in the Global Value Queue.

matches a difference that is already stored in the prediction table at the same position, then correlation has been identified.

However, in the baseline gDiff predictor, the GVQ is filled at retirement, meaning that a result required to generate a prediction may not be in the GVQ at prediction time. Therefore, the authors propose to use results produced at Execute to fill the GVQ earlier. Unfortunately, in most modern high-performance processors, results become available out-of-order. Therefore, the order in which results are put in the GVQ may vary depending on processor events impacting dynamic scheduling (e.g., cache misses). Consequently, the authors propose to back gDiff with another predictor (e.g., stride-based) to fill the gaps and enforce program order within the GVQ. This implies that the performance of gDiff will be contingent on the performance of this backing predictor, especially as the instruction window grows in size.

It follows that gDiff is a complex predictor. For an order n predictor (8 in [ZFC03]), a prediction involves reading and multiplexing n differences from the prediction table. The base value is read from the GVQ according to the *distance* field, and then added to the difference. If gDiff does not have a prediction for the instruction, the backing predictor produces a prediction and inserts it in the GVQ.

Updating involves computing n full-width differences in parallel, then comparing them to the n differences stored in the prediction table, for each dynamic instruction eligible for prediction. In addition, at Writeback, the result produced by the instruction overwrites the prediction in the GVQ. Consequently, the GVQ appears as a complex structure that must support several random accesses per cycle as several instructions may complete each cycle.

Lastly, space efficiency is questionable as results in the GVQ also live in the PRF. Similarly, among the n differences stored for each static instructions, only one will be used to predict.

2.1.2 Context-based Prediction

Context-based predictors are really *pattern-matching* predictors. If a given context of the processor matches the context recorded in the predictor, then the predictor simply has to look at the value that was produced following that context and provide it to the instruction.

As a result, and contrarily to computational predictors, context-based predictors cannot generally predict a value that has never been seen during execution.

2.1.2.1 Finite Context-Method (FCM) Prediction

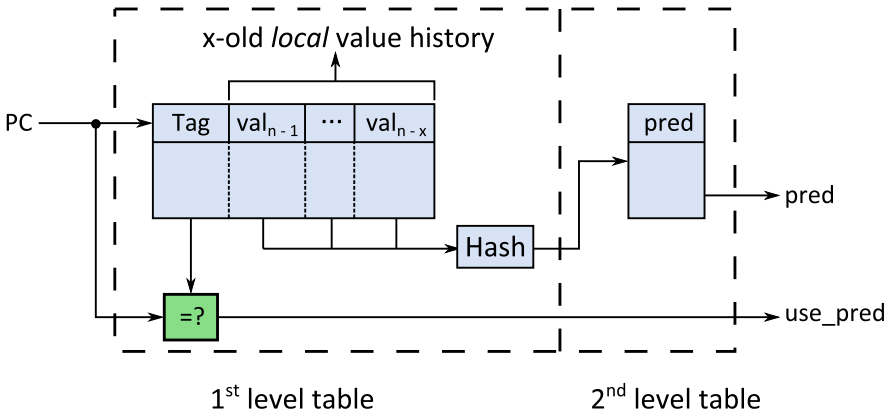


Figure 2.7: Order x Finite Context Method Predictor. The prediction is retrieved from the second level table using a hash of the x last values as an index.

FCM Framework Sazeides and Smith define a framework for building *Finite Context Method* (FCM) predictors in [SS98a]. FCM predictors are two-level predictors. The first level, the *Value History Table* (VHT), contains the local value history of an instruction and is accessed with the instruction PC. The second level, the *Value Prediction Table* (VPT), is accessed with a hash of the local value history of the VHT and contains the actual prediction.

In effect, a given hash corresponds to a given pattern of recent values. Therefore, FCM provides the value that comes after said pattern in the local value stream. Sazeides and Smith define an n^{th} order FCM predictor as a predictor tracking the last n values produced by a given static instruction in the first level table. An order x FCM predictor is depicted in Figure 2.7.

To capture most of the patterns, an order of 4 is sufficient [SS98a]. This hash can be computed at update time, but depending on the hash function, the n last values may have to kept in the predictor entry, although they can be pre-hashed themselves (e.g., folded [SS98a]). Update also includes writing the retired value to the VPT entry that was accessed at prediction time.

Differential FCM Goeman et al. [GVDB01] build on FCM by tracking differences between values in the VHT and the VPT instead of values themselves. As such, the *Differential* FCM (D-FCM) predictor is much more space efficient because partial strides can be stored in the tables. An order x D-FCM predictor is depicted in Figure 2.8, where an additional table can be seen on the bottom. This table contains the last values to which the strides will be added.

Updating the D-FCM predictor is similar to updating FCM, albeit slightly more complex: First, the new stride has to be computed by subtracting the last value in the predictor table with the produced value. Second, this new stride is

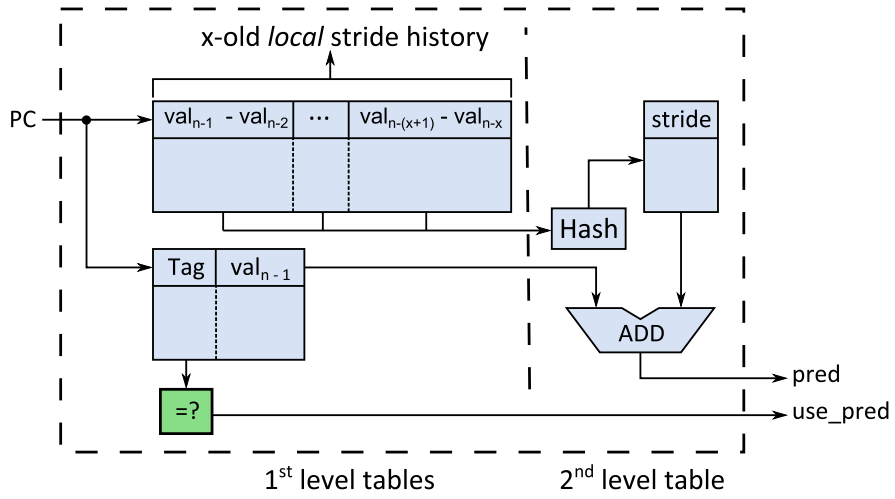


Figure 2.8: Order x Differential Finite Context Method Predictor. The stride is retrieved from the second level table using a hash of the x last strides as an index. Then, it is added to the value produced by the last dynamic instance of instruction.

pushed in the local history and the new hash is computed. In parallel, the stride is written in the VPT entry that was accessed at prediction time.

2.1.2.2 Last-n values Prediction

Perfect Selector In their pioneering paper, Lipasti and Shen [LS96] also introduced the *Last-n* predictor. Instead of storing the last value produced by the most recent instance of an instruction, they propose to store the n most recent values. Then, a selector is responsible for choosing a prediction among those n values. Therefore, an intuitive representation of a last-n predictor is to consider n Last Value predictors that are accessed in parallel and drive their respective predictions to a multiplexer that is controlled by the selector. In their study, Lipasti and Shen only consider a perfect selector, as a result, the first version of the Last-n predictor was unrealistic.

2-level Local Outcome History-based Burtscher and Zorn [BZ99] devise a realistic selector that leverages the *outcome* history of each recent value. That is, a bit-vector is added to each of the n values contained in a given entry. In this vector, a 1 at position k means that the $(k+1)^{th}$-old instance was correctly predicted, and a 0 means that it was incorrectly predicted.

This vector is used to index an array of saturating counters. That is, the prediction table is accessed using the lower bits of the PC to retrieve the n outcome histories that will be used to index n counter arrays. Finally, the counter values

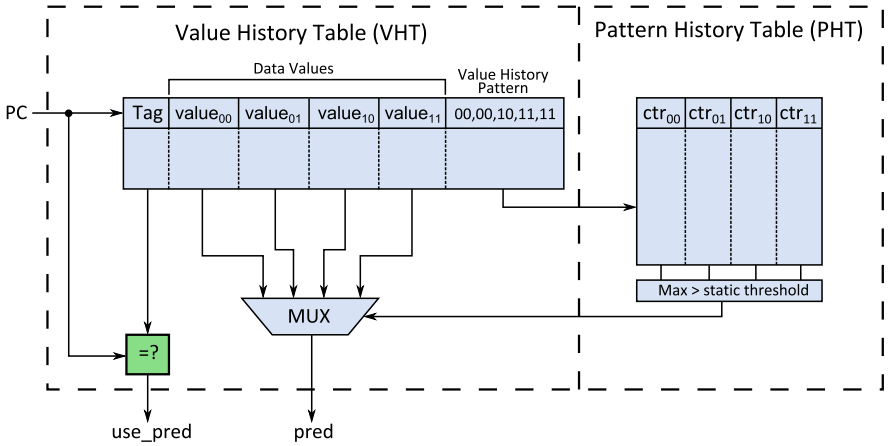


Figure 2.9: 2-level value predictor. The *Value History Pattern* of the first table (VHT) records the order in which the four most recent values appeared in the local value history. This pattern is used to index the *Pattern History Table* (PHT), which contains a counter for each value. The counter with the highest value corresponds to the value that will be used as prediction, if it is above a statically defined threshold.

are compared in order to find the most confident prediction for the current outcome histories. Due to this two-step operation, the last-n predictor of Burtscher and Zorn can be considered as a 2-level predictor.

Updating this predictor includes checking if the produced value is already present in one the the value fields of the entry, and inserting it if not. This requires n full-width comparisons. Then, the counters accessed at prediction time are updated by increasing the one corresponding to the produced value and decreasing the other ones.

In [BZ00], Burtscher and Zorn improve on the space-efficiency of their predictor by noting that the last n recent values share most of their high order bits. Therefore, they keep only one full-width value but only a few low order bits of the remaining $n-1$ values. Moreover, they note that using the LRU information, a stride can be computed on the fly between value_{n-1} and value_{n-2}, meaning that stride prediction is also achievable with this predictor.

2-level Local Value History-based The 2-level predictor was proposed by Wang and Franklin [WF97]. It can be seen a variation of both the Last-n values predictor of Burtscher and Zorn [BZ99] and the FCM predictor of Sazeides and Smith [SS97a, SS98a].

A 2-level predictor tracking the last four values observed in each static instruction value stream is depicted in Figure 2.9. It features two tables. First,

the *Value History Table* (VHT) contains the last four values observed in the local value history of each static instruction, as well as the order in which they appeared in the *Value History Pattern* (VHP). Each item of the VHP encodes one of the four values in the VHT. For instance, if the last four values are respectively $[0\ 1\ 2\ 3]_{hex}$, and the VHP is $[00\ 01\ 00\ 01\ 11]_b$ this means that the local value history is $[0\ 1\ 0\ 1\ 3]_{hex}$ (from oldest to youngest).

The VHP is used to index the *Pattern History Table* (PHT). An entry of the PHT contains one counter for each value in the VHT. The counter having the highest value corresponds to the value in the VHT that will be used as the prediction, as long as the counter value is above a statically defined threshold.

Similarly to FCM, the 2-level predictor gives a prediction corresponding to a given local value history. However, it can only choose among the four more recent values, as the Last-4 values predictor imagined by Lipasti and Shen [LS96]. Moreover, the history is not perfectly represented as a new value may overwrite an older one in the VHT. As a result, all elements of the VHP pointing to the VHT entry that contained the old value now point to the value that was just inserted. In FCM, the values stored in the first level table perfectly match the local value history.

Regardless, the 2-level predictor presented in [WF97] is not evaluated against an equivalent FCM predictor. Thus, it is not clear which performs best. However, both leverage the local value history to predict, which is impractical as we will illustrate in Chapter 3.

Updating the 2-level predictor involves checking if the new value is already present in the VHT. This involves four 64-bit comparisons in [WF97]. If not, then a value is evicted using a regular replacement algorithm such as LRU. Then, the VHP is left-shifted (by two bits in [WF97]) and the new outcome is inserted. In the meantime, the counter corresponding to the new value in the PHT entry selected at prediction time is incremented by 3, and all remaining counters are decremented by 1. Finally, note that contrarily to FCM, the values in the local history cannot be kept pre-hashed as they will be used as predictions. Therefore, each entry is quite big as n full-width values must be kept uncompressed.

2.1.2.3 Dynamic Dataflow-inherited Speculative Context Predictor

Finally, Thomas et al. introduce a predictor that uses predicted dataflow information to predict values: the Dynamic Dataflow-inherited Speculative Context predictor or DDISC [TF01]. As *gDiff*, DDISC relies on a backing predictor to generate predictions for easy-to-predict instructions. This is required since for hard-to-predict instructions, DDISC makes use of an improved context consisting not only of the instruction PC, but of the value of its operands. Since these operands may become available very late in the pipeline, using their predicted

values is necessary.

To build this improved context, a second register file is introduced. It contains the *signature* of each dynamic instruction, which is generated by hashing the signature of its producers. In particular, the signature of an instruction predicted by the backing predictor is the prediction itself, while the signature of an instruction not predicted by the backing predictor is a XOR of its producers signatures.[1] As a result, the DDISC prediction table is a simple last value table, except it is accessed with the signature rather than the instruction PC.

As mentioned in [TF01], the performance of DDISC is contingent on the performance of the backing predictor, and its ability to classify instructions as being predictable or not. Moreover, due to DDISC requiring the predicted values of producers to access its predictions table for consumers, superscalar value prediction may be too slow because the backing predictor must be done predicting the producer before the DDISC table can be accessed for the consumer.

2.1.3 Hybrid Predictors

Sazeides and Smith suggest that computational and context-based predictors are complementary to some-extent [SS97a]. As a result, it is possible to get the best of both world by *hybridizing* those two types of predictors. Due to this observation, many hybrid predictors have actually been developed.

The simplest way to design a hybrid is to put two or more predictors alongside and add a metapredictor to arbitrate. Since value predictors generally use confidence estimation,[2] the metapredictor can be very simple e.g., never predict if both predictors are confident but disagree, otherwise use the prediction from the confident predictor.

An example of this "naive" hybridization technique is the 2-level + Stride predictor of Wang and Franklin [WF97]. In their case, Stride can predict only when 2-level does not predict (because it is not confident enough). Therefore, there is no interaction between the two predictors, although they are able to share some hardware (e.g., the last value required by Stride is also present in 2-level, therefore, it is tracked in only one location).

Moreover, if components are complementary, there often is overlap: storage is wasted if an instruction predictable by all components has an entry in every component. Similarly, if an instruction is not predictable by one component, it is not efficient to let it occupy an entry in said component. Rychlik et al. propose to use a classifier to attribute an instruction to one component at most [RFK+98]. This partially addresses space-efficiency, but not complexity.

[1] Loads use the signature of the associated store producer.

[2] Whether it be regular saturating counters associated to each prediction or more refined schemes such as the outcome history-based counters of Burtscher and Zorn [BZ99].

As a result, more tightly-coupled predictors were proposed. In fact, many of the predictors that we previously mentioned can be seen as tightly-coupled hybrid predictors. For instance, D-FCM [GVDB01] is a hybrid of Stride and FCM. However, contrarily to a "naive" hybrid, it is able to predict the stride based on the local value history, when the former can only predict a single stride for a given static instruction (Stride), or a pattern of *values* for a given static instruction (FCM). Similarly, the *Per-Path Stride* (PS) predictor of Nakra et al. [NGS99] can also be seen a tightly-coupled hybrid, since it uses a computational predictor (2-delta Stride), but indexes it based on context (global branch history).

2.1.4 Register-based Value Prediction

Instead of relying on a predictor, Tullsen and Seng propose to reuse values that are already present in registers [TS99]. They present both static and dynamic mechanisms. In the former case, the ISA has to be modified so that the programmer or compiler can explicitly state that an instruction should use the value already present in a specific register as a prediction. In the latter case, no ISA modifications is required as all instructions (or a subset, e.g., loads) are predicted using this technique, depending on per-PC confidence counters. In both cases, the compiler has opportunities to alter register allocation so that register reuse is more frequent.

This form of speculation is able to capture local correlation (e.g., last-value reuse) as well as global correlation (e.g., reuse of a register produced by a different instruction). Nonetheless, only reuse can be captured, while more advanced relations between values (e.g., strided patterns) cannot.

Register-based value prediction is particularly interesting as no additional storage is required, which alleviates concerns regarding storage requirement and power consumption in the value predictor. However, additional logic is still required to handle mispredictions as well as identify instructions eligible for prediction in the dynamic case (e.g., comparing the produced result with the previous value of the architectural destination register).

2.2 Speculative Window

In most of the predictors we described, a strong requirement is that the result of the previous instance(s) is required to generate the prediction for the current instance. For example, in stride-based predictors, the last value is required to predict for the next instance, and in FCM, the last n values are required.

Unfortunately, the previous instance may not have retired at prediction time, meaning that a stale value is present in the predictor entry. In fact, Lee and Yew

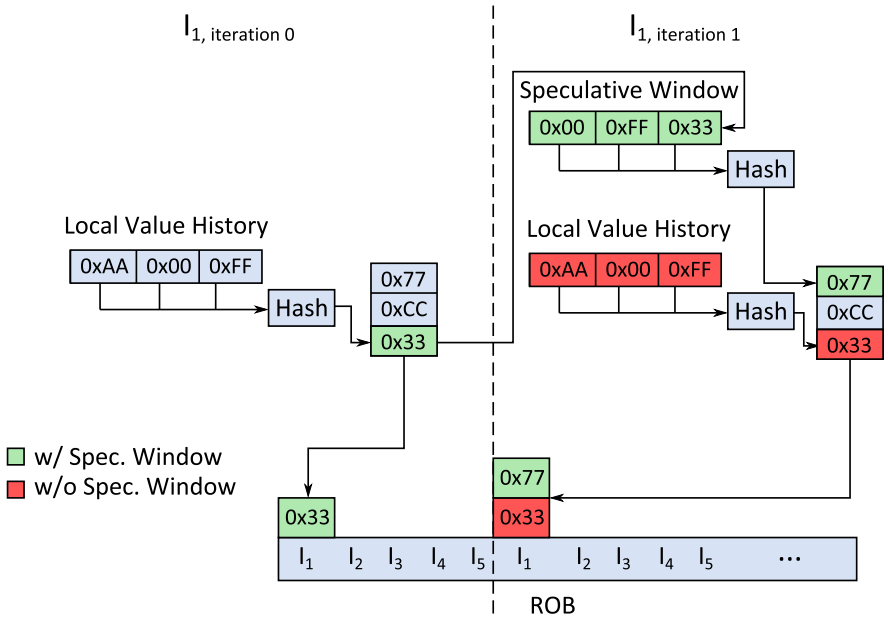


Figure 2.10: Prediction of two close instances of a given static instruction with FCM. In the absence of a speculative window, the local value history is not up to date for the second instance, and the prediction is most likely wrong.

found that for SPECint95 benchmarks, as much as 36% of instructions access the same prediction table within 5 cycles, and 22% within 2 cycles [LY00]. Therefore, the predictor should be updated speculatively at prediction time, under penalty of losing potential in workloads having many short loops. Any wrong speculative update should be undone at Commit.

To better illustrate how potential can be lost, consider Figure 2.10 that shows how FCM would handle close instances of the same static instruction. In particular, in the absence of a speculative window, the result of $I_{1,iteration0}$ would be put in the local value history at retirement, well after $I_{1,iteration1}$ would have been fetched. To allow speculative updates, a speculative window tracking predictions of instructions that have not retired yet must be implemented.

Unfortunately, in most studies,[3] this requirement is not addressed while the absence of a speculative window has substantial impact on performance in specific benchmarks, as we will demonstrate in Chapter 5. In this Chapter, we will also provide a realistic implementation of the speculative window. Indeed, an intuitive way of implementing it would be to build a fully associate ROB-sized[4] structure that would be queried and updated on each prediction. However, such a structure

[3]The exception is *gDiff* where the GVQ is essentially a speculative window filled by the backing predictor.

[4]192 entries in recent Intel Haswell processors.

would be slow and power hungry [EA02, KL03, PJS97], especially since several predictions (i.e., associative search of the window, then inserting the prediction in it) can be made each cycle in superscalar processors. Therefore, a different way to implement the speculative window is required.

Lastly, it should be noted that for stride-based predictors, only one result has to be tracked per instruction in the speculative window. However, for predictors using local value histories (e.g., FCM [SS98a] and 2-level [WF97]) or local outcome histories (e.g., Last-n with SAg confidence estimation [BZ99]), up to n values (respectively outcome histories) have to be tracked for each instruction, and merging the information contained in the predictor entry and the speculative window may be required to generate the index of the second level. As a result, managing the speculative window should be much more complex for those predictors.

2.3 Confidence Estimation

As a value misprediction entails a certain cost (depending on the recovery mechanism), predictors are usually accompanied by a confidence estimation mechanism, to filter out uncertain predictions. As for branch predictors, 2- or 3-bit saturating counters [Smi81, JRS96] can be used, with a potentially asymmetric update policy. In particular, a counter is associated with each prediction, and the latter is used by the pipeline only if the counter value is saturated or above a certain threshold. Said counter is updated at retire time, by incrementing it if the prediction was correct, and decrementing – or resetting – it if not.

A more refined scheme proposed independently by Burtscher and Zorn [BZ99, Bur00] and Calder et al. [CRT99] leverages the per-instruction recent outcomes (i.e., prediction success or failure) to index a table containing saturating counters. This SAg confidence estimation scheme (inspired by Yeh and Patt [YP93]) is more accurate than a regular array of PC-indexed counters, leading to additional speedup in [Bur00] (from 0.1% to 2% on average). Nonetheless, it is slower due to its 2-level nature and requires much more storage, as each predictor entry now stores n outcome history bits to index the counter arrays, with n being 10 in [Bur00].

In Chapter 3, we will argue that regular saturating counters can be made very efficient by having them saturate probabilistically, after many correct predictions. Therefore, very high accuracy can be achieved with very simple hardware and low storage overhead.

2.4 Validation

As Value Prediction is speculative in nature, a prediction must be validated against the actual result once it is known. To our knowledge, validation takes place as soon as possible in virtually all previous studies [GM98a, GVDB01, ZFC03, RFK+98, ZYFRC00, SS98a, GM96, LS96, WF97, RFKS98, BZ99, Bur00, GM98b, GG98b, CRT99, SS97a, TS99, NGS99, LWS96, TF01, LY02, LWY00, LY01]. That is, predictions are validated as results are produced, at the output of the functional units. In that case, the prediction has to be read from the Physical Register File (PRF) along the source operands and propagated to the functional unit (FU) when the predicted instruction is selected for execution. Lastly, a comparator must be implemented after each FU to do the validation. This comparison step may require an additional pipeline stage or be done during Writeback.

As a result, validation at Execute also has some implications on the PRF. Indeed, it is commonly assumed that predictions and actual results co-exist in the PRF to ease the management of predictions. In particular, this removes the need to multiplex from the PRF and a buffer dedicated to predictions. However, this implies that predictions are written to the PRF, hence that additional write ports are available at Dispatch. Moreover, in addition to reading two operands from the PRF when it is issued, an instruction will also need to read the prediction in order to propagate it to the FU and validate it. As a result, in the worst case, up to 66% more ports are required on the PRF, assuming 1 write port and 2 read ports for each potentially issued instruction. This is already impractical for moderate-issue processors given the increase in area and power consumption of the PRF as the number of ports increases [ZK98]. Fortunately, we will show in Chapter 3 that validation can actually be delayed until Commit, greatly decreasing the additional port pressure put by VP on the PRF.

In the case of a correct prediction, the instruction continues towards Writeback and Commit. In the case of an incorrect prediction, however, dependent instructions may have executed with an incorrect source operand. As a result, they should be re-executed.

2.5 Recovery

Contrarily to a branch misprediction, a value misprediction does not necessarily imply that execution is on the wrong control path. As a result, recovery is more flexible and does not *require* a pipeline squash. In theory, only the instructions that depend directly or indirectly on the value misprediction must be re-executed. The computations made by independent instructions are correct regardless of the misprediction.

2.5.1 Refetch

Although not necessary, squashing the pipeline starting from the mispredicted instruction or the oldest dependent remains the simplest way to recover. This mechanism, known as *Refetch* [TS99] is similar to the mechanism used to handle branch mispredictions.

Refetch has several drawbacks. First, it is slow: the cost of a value misprediction becomes similar to the cost of a branch misprediction. As a result, it can only be used if the value predictor is very accurate. Indeed, if all instructions producing a register are predicted (i.e., not only loads), then the average gain of a correct prediction may be very low (e.g., one cycle if a simple addition is predicted), while the cost of a misprediction would be very high.

Second, it may require more hardware than expected. Indeed, consider that on a branch prediction, a *checkpoint* of the architectural state (e.g., register mappings, pipeline state, etc.) must be taken to correctly restart execution should the branch be mispredicted. Indeed, since branch resolution is done at Execute, the state to restore on a misprediction is not the software-visible state provided by the Commit stage. Rather, the architectural state corresponding to the branch instruction itself should be restored, hence a checkpoint is required because this state might still be speculative at branch resolution time since older instructions might still be in flight. However, since branches may represent one out of eight or ten instructions, the number of hardware checkpoints to implement is manageable. Moreover, checkpoints may only be taken on low confidence branches, which are even less frequent.

Consequently, and since common wisdom suggests that resolving branches at Commit doubles the average penalty of a branch misprediction, checkpointing is worth the benefit of resolving branches at Execute instead than Commit.

We already mentioned that value predictions are validated at Execute. Therefore, recovery with pipeline squashing implies that the speculative state to restore has been checkpointed. However, contrarily to branches, instructions eligible for VP represent the majority of the dynamic instructions. Moreover, the dynamic range of values is much higher than that of branch outcomes, meaning that the accuracy of the value predictor may be much lower. Consequently, an ideal implementation of Refetch should provision checkpoint space for every in-flight instruction, or around two hundred checkpoints if we consider a recent microarchitecture such as Intel Haswell.[5] As a result, despite its apparent simplicity, an optimal implementation of Refetch may not be possible for Value Prediction. In particular, the number of checkpoints may be limited, leading to an increase in the misprediction penalty as instructions older than the misprediction but younger than the youngest correct checkpoint will be also squashed.

[5]There may be 192 in-flight μ-ops in Haswell.

2.5.2 Replay

An alternative to Refetch is to simply re-execute instructions, since they are already in the pipeline because the control path is correct. This mechanism, known as *Replay*, can be either *selective* or not. As both mechanisms do not imply a pipeline squash, they are much less expensive than Refetch (in cycles). Nonetheless, their operation is not trivial and still likely to cost several cycles.

Selective Replay This first version of Replay re-executes only the instructions that have executed with the wrong source(s). Instructions independent on the misprediction are **not** replayed.

Selective replay requires the identification of all instructions directly or indirectly dependent on the misprediction through both register and memory dependencies. Since the concerned instructions necessarily have executed before the instruction that was wrongly predicted, selective replay for VP cannot be done in the same fashion as selective replay for schedule mispredictions [KL04].

In the latter case, instructions dependent on a variable-latency instruction may execute with wrong sources when the producer latency is higher than expected. Nonetheless, since the RAW dependency is enforced in the scheduler [PSM+15, MBS07, MS01, MSBU00, MLO01], the dependents are still issued **after** the long-latency instruction. As a result, by dedicating a pipeline stage to the validation of all instructions after *Execute*, the wrong execution of a dependent can be identified because the unexpected longer latency of the producer is known *before* the dependent actually executes.

In the case of Value Prediction, a dependent may have executed well **before** the producer executes. As a result, it has left the execution pipeline and only sits in the ROB and potentially a dedicated buffer holding instructions that may have to be replayed. Therefore, the only way to identify that it needs to be replayed is to scan the ROB or buffer and re-dispatch instructions to the scheduler accordingly[6]. Consequently, selective replay for VP is likely to take several cycles as an arbitrarily long dependency chain has to be identified and replayed. Moreover, it does not integrate seamlessly with the mechanism used for schedule mispredictions because the producer triggers the re-execution of dependents, while in the case of schedule mispeculations, each dependent determines if it correctly executed or not (at least in some implementations such as Intel Pentium 4 [SGC01b]).

Nonetheless, an implementation adapted to Value Prediction is provided by Kim and Lipasti in [KL04]. In a nutshell, each instruction possesses a moderately small bit-vector (8-16 bits) indicating if the instruction depends on a given *token*.

[6] Alternatively, instructions may retain their entry in the scheduler as long as they remain dependent on an un-resolved value prediction

A token belongs to an instruction that may execute incorrectly, be it a value predicted instruction, or a load that woke up its dependents speculatively assuming a hit. The last key component is that instructions retain their scheduler entry as long as they may have to be replayed. As a result, even if a dependent has issued before a producer because the producer is value predicted, the producer-consumer relation is preserved in the scheduler thanks to the bit-vector. Hence, replay is not dependent on the issue order. If an instruction that does not have a token (all have been allocated to other instructions) is mispredicted, the scheduler is flushed and instructions are re-inserted from the ROB, guaranteeing correctness.

Non-selective Replay Conversely to the previous scheme, non-selective replay simply has to re-dispatch all in-flight instructions that are younger than the mispredicted instruction from the ROB. In particular, it is not necessary to scan the ROB and LQ/SQ in a sequential fashion to precisely identify dependencies, only the faulting instruction has to be found. As a result, recovery latency may be on par with selective replay, although more instructions will have to be replayed, increasing contention.

2.5.3 Summary

Of the three mechanisms described in the literature, none can be implemented in a straightforward fashion. Refetch requires substantial checkpointing hardware since most instructions are eligible for Value Prediction. Selective replay has high complexity yet will still require several cycles to process the arbitrarily long dependency chain to replay. Non-selective replay has lower complexity but a potentially higher latency. Note that both flavors of replay will still be much faster than Refetch.

2.6 Revisiting Value Prediction

Given the salient issues existing with previous propositions (e.g., complex predictors, execute-time validation, complex recovery, PRF port requirement, etc.), the reader may question the choice to reconsider Value Prediction as a means to increase sequential performance. Nonetheless, let us consider the following:

1. Value Prediction intrinsically targets sequential performance. It does not try to leverage multi-cores to improve sequential performance, contrarily to certain schemes such as *Helper Threading* [CSK+99], *Speculative Multithreading* (such as *Multiscalar* [Fra93]) and *Slipstream* [SPR00], to cite a few. Therefore, it is the author's intuition that Value Prediction is a more natural candidate to improve sequential performance.

2. Value Prediction is orthogonal to mechanisms usually considered to increase sequential performance, such as improved branch prediction and prefetching. Moreover, we argue that it can generally be leveraged to improve said mechanisms as knowing the output of instructions can be of great help e.g., address prediction for better disambiguation and cache bank prediction [YERJ], branch prediction [AGGG01, GG99], or prefetching [EV93, GG97].

3. Value Prediction has high potential: a perfect predictor can yield tremendous improvements up to 5X in [PS14b], especially if the cache hit ratio is low.

As a result, if one could address the implementation challenges posed by Value Prediction, additional sequential performance could be directly harvested from the increased ILP. In a context where parallel performance is limited by Amdahl's Law and many intrinsically sequential workloads remain, this additional performance would be more than welcome. Moreover, still assuming that a low cost implementation can be provided, VP would improve performance without increasing the size of the out-of-order window or the issue width, which contribute to sequential performance but also to overall complexity and power consumption. Better yet, it would actually allow for a better utilization of the out-of-order engine by increasing average ILP. As a result, it may even be possible to *reduce* the out-of-order aggressiveness through Value Prediction. Therefore, despite the remaining challenges, there are strong arguments in favor of VP.

This thesis revisits Value Prediction in a contemporary context and tackles the main issues preventing VP from being added to a modern superscalar pipeline. More specifically, it provides an end-to-end (predictor in the frontend to validation and recovery in the backend) implementation of VP. This implementation actually possesses a *less aggressive* out-of-order engine, greatly diminishing the overall complexity of the pipeline.

Chapter 3

Revisiting Value Prediction in a Contemporary Context

This Chapter covers a publication in the International Symposium on High Performance Computer Architecture (HPCA 2014) [PS14b] as well as part of a second publication in the same conference (HPCA 2015) [PS15a].

3.1 Introduction

Initial studies have led to moderately to highly accurate value predictors [NGS99, SS97a, WF97] while predictor accuracy has been shown to be critical due to the misprediction penalty [CRT99, ZYFRC00]. Said penalty can be as high as the cost of a branch misprediction, yet the benefit of an individual correct prediction is often very limited. As a consequence, high coverage[1] is mostly irrelevant in the presence of low accuracy.[2]

To that extent, we first present a simple yet efficient confidence estimation mechanism for value predictors. The *Forward Probabilistic Counters* (FPC) scheme yields value misprediction rates well under 1%, at the cost of reasonably decreasing predictor coverage. All classical predictors are amenable to this level of accuracy. FPC is very simple to implement and does not require substantial change in the confidence counters update automaton. Our experiments show that when FPC is used, no complex repair mechanism such as selective replay [KL04, TS99] is needed. Prediction validation can even be delayed until commit time and be done in-order: Complex and power hungry logic needed for execution time validation is not required anymore. As a result, prediction is performed in the in-order pipeline front-end, validation is performed in the in-order

[1] Correct predictions over total number of dynamic instructions eligible for VP.

[2] Correct predictions over total number of predictions that were used by the pipeline.

pipeline back-end while the out-of-order execution engine is only marginally modified. This is in contrast with all previous propositions relying on execution time validation.

Second, we introduce the *Value TAGE* predictor (VTAGE) and a direct improvement, the *Differential VTAGE* predictor (D-VTAGE). VTAGE is derived from research propositions on branch predictors [SM06] and more precisely from the indirect branch predictor ITTAGE. VTAGE is the first hardware value predictor to leverage a long global branch history and the path history. Like all other value predictors, VTAGE is amenable to very high accuracy thanks to *Forward Probabilistic Counters*. D-VTAGE is a tightly-coupled hybrid that resembles the Differential Finite Context Method (D-FCM) predictor. In particular, it tracks the difference between successive results of instances on a given path, rather than each distinct values. It is shown to outperform previously proposed context-based predictors such as Finite Context Method (FCM) [SS97a] and a naive hybrid between VTAGE and a 2-delta Stride predictor [EV93].

Moreover, we point out that unlike two-level predictors (in particular, predictors based on local value histories such as FCM), VTAGE and D-VTAGE can seamlessly predict back-to-back occurrences of instructions, that is, instructions inside tight loops. Practical implementations are then feasible.

3.2 Motivations

We identify two factors that will complicate the adaptation and implementation of value predictors in future processor cores. First, the misprediction recovery penalty and/or hardware complexity. Second the need to handle back-to-back predictions for two occurrences of the same instruction which can be very complex to implement while being required to predict instructions within tight loops.

3.2.1 Misprediction Recovery

Most of the initial studies on Value Prediction were assuming that the recovery on a value misprediction is immediate and induces – almost – no penalty [GM98b, LWS96, LS96, ZFC03] or simply focused on accuracy and coverage rather than actual speedup [GVDB01, NGS99, RFK+98, SS97a, TF01, WF97]. The latter studies were essentially ignoring the performance loss associated with misprediction recovery. Moreover, despite quite high coverage and reasonable accuracy, one observation that can be made from these early studies is that *the average performance gain per correct prediction is rather small.*

Furthermore, Zhou et al. observed that to maximize the interest of VP, the total cost of recoveries should be as low as possible [ZYFRC00]. To limit this

total cost, one can leverage two factors: The average misprediction penalty P_{value} and the absolute number of mispredictions N_{misp}. A very simple modelization of the total misprediction penalty is $T_{recov} = P_{value} * N_{misp}$.

3.2.1.1 Value Misprediction Scenarios

Two mechanisms already implemented in processors can be adapted to manage value misprediction recovery: Pipeline squashing and selective replay. They induce very different average misprediction penalties, but are also very different from a hardware complexity standpoint, as we illustrated in 2.5.

Pipeline squashing is already implemented to recover from branch mispredictions. On a branch misprediction, all the subsequent instructions in the pipeline are flushed and instruction fetch is resumed at the branch target. This mechanism is also generally used on load/store dependency mispredictions [BBB+11, CE98]. Using pipeline squashing on a value misprediction is straightforward, but costly as the minimum misprediction penalty is the same as the minimum branch misprediction penalty. However, to limit the number of squashes due to VP, squashing can be avoided if the predicted result has not been used yet, that is, if no dependent instruction has been issued.

Selective replay is implemented in processors to recover in case where instructions have been executed with incorrect operands, in particular this is used to recover from L1 cache hit/miss mispredictions [PSM+15, MBS07, MS01, MSBU00, SGC01b, MLO01, KL04] (i.e., load-dependent instructions are issued after predicting a L1 hit, but finally the load results in a L1 miss). When the execution of an instruction with an incorrect operand is detected, the instruction as well as all its dependent chain of instructions are canceled then replayed.

Validation at Execution Time vs. Validation at Commit Time On the one hand, selective replay must be implemented at execution time in order to limit the misprediction penalty. On the other hand, pipeline squashing can be implemented either at execution time or at commit time. Pipeline squashing at execution time results in a minimum misprediction penalty similar to the branch misprediction penalty. However, validating predictions at execution time necessitates to redesign the complete out-of-order engine: The predicted values must be propagated through all the out-of-order execution engine stages and the predicted results must be validated as soon as they are produced in this out-of-order execution engine. Moreover, the repair mechanism must be able to restore processor state for any predicted instruction. Lastly, prediction checking must also be implemented in the commit stage(s) since predictors have to be trained even when predictions were not used due to low confidence.

On the contrary, pipeline squashing at Commit results in a quite high average

misprediction penalty since it can delay prediction validation by a substantial number of cycles. Yet, it is much easier to implement for Value Prediction since it does not induce complex mechanisms in the out-of-order execution engine. It essentially restrains the Value Prediction related hardware to the in-order pipeline front-end (prediction) and the in-order pipeline back-end (validation and training). Moreover, it allows not to checkpoint the rename table since the committed rename map contains all the necessary mappings to restart execution in a correct fashion.

A Simple Synthetic Example Realistic estimations of the average misprediction penalty P_{value} could be 5-7 cycles for selective replay,[3] 20-30 cycles for pipeline squashing at execution time and 40-50 cycles for pipeline squashing at Commit.

For the sake of simplicity, we will respectively use 5, 20 and 40 cycles in the small example that follows. We assume an average benefit of 0.3 cycles per correctly predicted value (taking into account the number of unused predictions). With predictors achieving around 40% coverage and around 95% accuracy as often reported in the literature, 50% of predictions used before execution, the performance benefit when using selective replay would be around 64 cycles per Kinstructions, a loss of around 86 cycles when using pipeline squashing at execution time and a loss of around 286 cycles when using *pipeline squashing* at commit time.

Our experiments in Section 3.7 confirm that when a value predictor exhibits a few percent misprediction rate on an application, it can induce significant performance loss when using pipeline squashing at Commit. Therefore, such a predictor should rather be used in conjunction with selective replay.

3.2.1.2 Balancing Accuracy and Coverage

The total misprediction penalty T_{recov} is roughly proportional to the number of mispredictions. Thus, if one drastically improves the accuracy at the cost of some coverage then, as long as the coverage of the predictor remains quite high, there might be a performance benefit brought by Value Prediction, even though the average value misprediction penalty is very high.

Using the same example as above, but sacrificing 25% of the coverage (now only 30%), and assuming 99.75% accuracy, the performance benefit would be around 88 cycles per Kinstructions cycles when using *selective replay*, 83 cycles

[3]Including tracking and canceling the complete chain of dependent instructions as well as the indirect sources of performance loss encountered such as resource contention due to reexecution, higher misprediction rate (e.g. a value predicted using wrong speculative value history) and lower prediction coverage

when using pipeline squashing at execution time and 76 cycles when using pipeline squashing at commit time.

In Section 3.7, we will show that the ranges of accuracy and coverage allowed by our FPC proposition are in the range of those used in this small example. As a consequence, it is not surprising that our experiments confirm that, using FPC, pipeline squashing at commit time achieves performance in the same range as an idealistic 0-cycle selective replay implementation.

3.2.2 Back-to-back Prediction

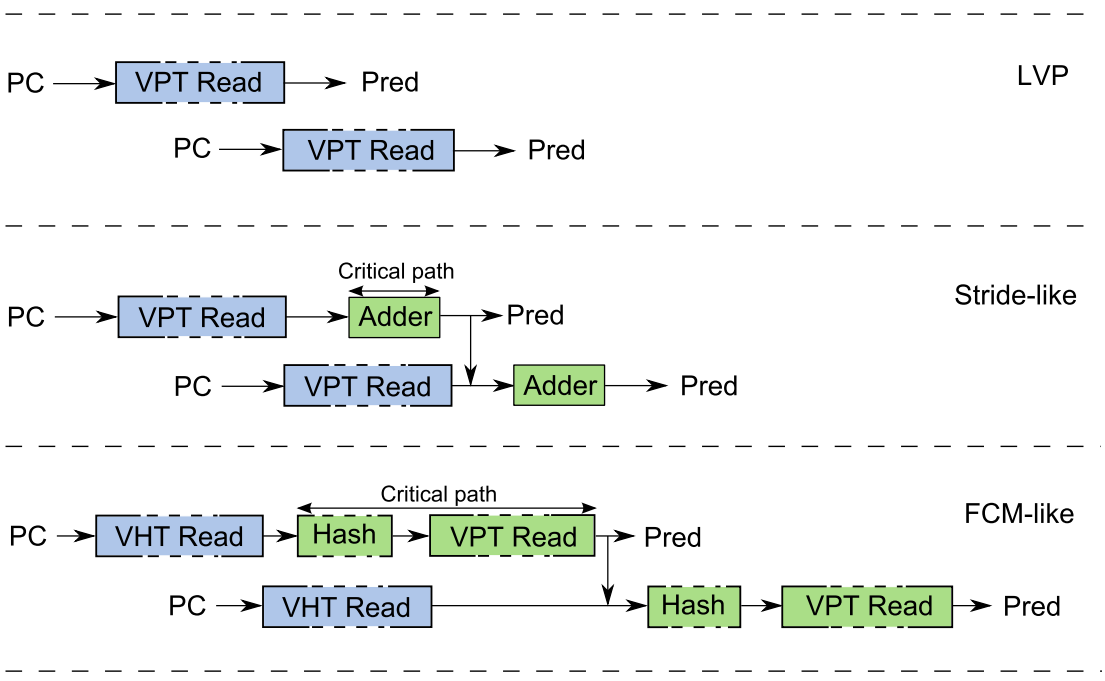


Figure 3.1: Prediction flow and critical paths for different value predictors when two occurrences of an instruction are fetched in two consecutive cycles.

Unlike a branch prediction, a value prediction is needed rather late in the pipeline (at dispatch time). Thus, at first glance, prediction latency does not seem to be a concern and long lookups in large tables and/or fairly complex computations could be tolerated. However, for most predictors, the outcomes of a few previous occurrences of the instruction are needed to perform a prediction for the current occurrence. Consequently, for those predictors, either the critical operation must be made short enough to allow for the prediction of close (possibly back-to-back) occurrences (e.g. by using small tables) or the prediction of tight loops must be given up. Unfortunately, tight loops with candidates for VP are quite abundant in existing programs. Experiments conducted with the

methodology we will introduce in Section 3.6 suggest that for a subset of the SPEC'00/'06 benchmark suites, fetched instructions eligible for VP for which the previous occurrence was fetched in the previous cycle (8-wide Fetch) represent up to 38.8% (4.4% a-mean) of the fetched instructions. We highlight such critical operations for each predictor in the subsequent paragraphs.

3.2.2.1 LVP

Despite its name, LVP does not require the previous prediction to predict the current instance as long as the table is trained. Consequently, LVP uses only the program counter to generate a prediction. Thus, successive table lookups are independent and can last until Dispatch, meaning that large tables can be implemented. The top part of Fig. 3.1 describes such behavior. Similarly, the predictor we introduce in Section 3.5, VTAGE, only uses control-flow information to predict, allowing it to seamlessly predict back-to-back occurrences. The same goes for the *gDiff* predictor itself [ZFC03] although a critical path may be introduced by the predictor providing speculative values for the global history.

3.2.2.2 Stride

Prediction involves using the result of the previous occurrence of the instruction. Thus, tracking the result of only the last speculative occurrence of the instruction is sufficient.

The Stride value predictor acts in two pipeline steps: 1) retrieval of the stride and the last value 2) computation of the sum. The first step induces a difficulty when the last occurrence of the instruction has not committed or is not even executed. One has to track the last occurrence in the pipeline and use the speculative value predicted for this occurrence. This tracking can span over several cycles. The critical operation is introduced by the need to bypass the result of the second step directly to the adder in case of fetching the same instruction in two consecutive cycles (e.g. in a very tight loop), as illustrated by the central part of Figure 3.1. The D-VTAGE predictor that we introduce in Section 3.5 possesses the same critical path. Regardless, a stride-based value predictor supporting the prediction of two consecutive occurrences of the same instruction one cycle apart could reasonably be implemented since the second pipeline step is quite simple. In other words, since back-to-back additions can be done in the execution units, stride-based predictors should be able to handle back-to-back occurrences.

3.2.2.3 Finite Context Method

The local value history predictor (n^{th}-order FCM) is a two-level structure. The first-level consists of a value history table accessed using the instruction address.

This history is then hashed and used to index the second level table.

As for the Stride value predictor, a difficulty arises when several occurrences of the same instruction are fetched in a very short interval since the predictor is indexed through a hash of the last n values produced by the instruction. In many cases, the previous occurrences of the instruction have not been committed, or even executed. The logic will be more complex than for the Stride predictor since one has to track the last n values instead of a single one.

However, the critical delay is on the second step. To predict consecutive occurrences of an instruction, the delay between the accesses to the second level table for the two occurrences is likely to be much shorter than the delay for computing the hash, reading the value table and then forward the predicted value to the second index hash/computation, as illustrated by the bottom part of Figure 3.1. This implies that FCM must either use very small tables to reduce access time or give up predicting tight loops altogether. Note that D-FCM [GVDB01], Last-n [BZ99] with SAg confidence estimation,[4] 2-level [WF97] and DDISC [TF01] also require two successive lookups. Consequently, these observations also stand for those predictors.

In summary, a hardware implementation of a local history value predictor would be very complex since 1) it involves tracking the n last speculative occurrences of each fetched instruction 2) the predictor cannot predict instructions in successive iterations of a loop body if the body is fetched in a delay shorter than the delay for retrieving the prediction and forwarding it to the second level index computation step.

3.2.2.4 Summary

Table lookup time is not an issue as long as the prediction arrives before Dispatch for LVP and Stride. Therefore, large predictor tables can be considered for implementation. For stride-based value predictor, the main difficulty is that one has to track the last (possibly speculative) occurrence of each instruction.

For local value history-based predictors the same difficulty arises with the addition of tracking the n last occurrences. Moreover the critical operations (hash and the 2nd level table read) lead to either using small tables or not being able to timely predict back-to-back occurrences of the same instruction. Implementing such predictors can only be justified if they bring significant performance benefit over alternative predictors.

The VTAGE predictor we introduce in this work is able to seamlessly predict back-to-back occurrences of the same instruction, thus its access can span over several cycles. VTAGE does not require any complex tracking of the last

[4]Due to the need to update the outcome history speculatively for the next access, depending on the values of the n counters.

occurrences of the instruction. Conversely, D-VTAGE requires to track the last instance due to its stride-based nature. We provide an actual mechanism to do so in Chapter 5, where we focus on a practical implementation of the value predictor itself.

3.3 Commit Time Validation and Hardware Implications on the Out-of-Order Engine

In the previous Section, we have pointed out that the hardware modifications induced by pipeline squashing at commit time on the Out-of-Order engine are limited. In practice, the only major modification compared to a processor without Value Prediction is that the predicted values must be written in physical registers at Dispatch.

At first glance, if every destination register has to be predicted for each fetch group, one would conclude that the number of write ports should double. In that case the overhead on the register file would be quite high. The area cost of a register file is approximately proportional to $(R + W) * (R + 2W)$, R and W respectively being the number of read ports and the number of write ports [ZK98]. Assuming $R = 2W$, the area cost without VP would be proportional to $12W^2$ and the one with VP would be proportional to $24W^2$, i.e. the double. Energy consumed in the register file would also be increased by around 50% (using very simple Cacti 5.3 [TMJHJ08] approximations).

For practical implementations, there exist several opportunities to limit this overhead. For instance one can limit the number of extra ports needed to write predictions. Each cycle, only a few predictions are used and the predictions can be known several cycles before Dispatch: One could limit the number of writes on each cycle to a certain limit, and buffer the extra writes, if there are any. Assuming only $\frac{W}{2}$ write ports for writing predicted values leads to a register file area of $\frac{35W^2}{2}$, saving half of the overhead of the naive solution. The same savings are observed for energy consumption (Cacti 5.3 estimations). Another opportunity is to allocate physical registers for consecutive instructions in different register file banks, limiting the number of write ports on the individual banks. One can also prioritize the predictions according to the criticality of the instruction and only use the most critical one, leveraging the work on criticality estimation of Fields et al. [FRB01], Tune et al. [TTC02] as well as Calder et al. [CRT99].

Exploring the complete optimization set available to reduce the overhead on the register file design is out of the scope of this Chapter. It would depend on the precise micro-architecture of the processor, but we have clearly shown that this overhead in terms of energy and silicon area can be reduced to less than 25% and 50% respectively. Moreover, this overhead is restricted to the register file and does

not impact the other components of the out-of-order engine. Similarly, thanks to commit time validation, the power overhead introduced by Value Prediction will essentially reside in the predictor table.

3.4 Maximizing Value Predictor Accuracy Through Confidence

As we already pointed out, the total misprediction recovery cost can be minimized through two vehicles: Minimizing the individual misprediction penalty and/or minimizing the total number of mispredictions.

When using the prediction is not mandatory (i.e., contrarily to branch prediction), an efficient way to minimize the number of mispredictions is to use saturating counters to estimate confidence and use the prediction only when the associated confidence is very high. For instance, for the value predictors considered in this study, a saturating 3-bit confidence counter per entry that is reset on each misprediction leads to an accuracy in the 95-99% range if the prediction is used only when the counter is saturated. However this level of accuracy is still not sufficient to avoid performance loss in several cases unless idealistic selective replay is used. To increase accuracy, Burtscher et al. proposed the SAg confidence estimation scheme to assign confidence to a history of outcomes rather than to a particular instruction [BZ99]. However, this entails a second lookup in the counter table using the outcome history retrieved in the predictor table with the PC of the instruction. A way to maximize accuracy without increasing complexity and latency would be preferable.

We actually found that simply using wider counters (e.g., 6 or 7 bits) leads to much more accurate predictors while the prediction coverage is only reduced slightly. Prediction is only used on saturated confidence counters and counters are reset on each misprediction. Interestingly, probabilistic 3-bit counters such as defined by Riley et al. [RZ06] augmented with reset on misprediction achieve the same accuracy for substantially less storage and a marginal increase in complexity.

We refer to these probabilistic counters as *Forward Probabilistic Counters* (FPC). In particular, each forward transition is only triggered with a certain probability. In this work, we will consider 3-bit confidence counters using a probability vector $v = \{1, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}\}$ for pipeline squashing at Commit and $v = \{1, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{16}, \frac{1}{16}\}$ for selective replay, respectively mimicking 7-bit and 6-bit counters. This generally prevents all the considered VP schemes to slow down execution while minimizing the loss of coverage (as opposed to using lower probabilities). The used pseudo-random generator is a simple Linear Feedback Shift Register.

Using FPC counters instead of full counters limits the overhead of confidence

estimation. It also opens the opportunity to adapt the probabilities at run-time as suggested in [Sez11c] and/or to individualize these probabilities depending on the criticality of the instructions.

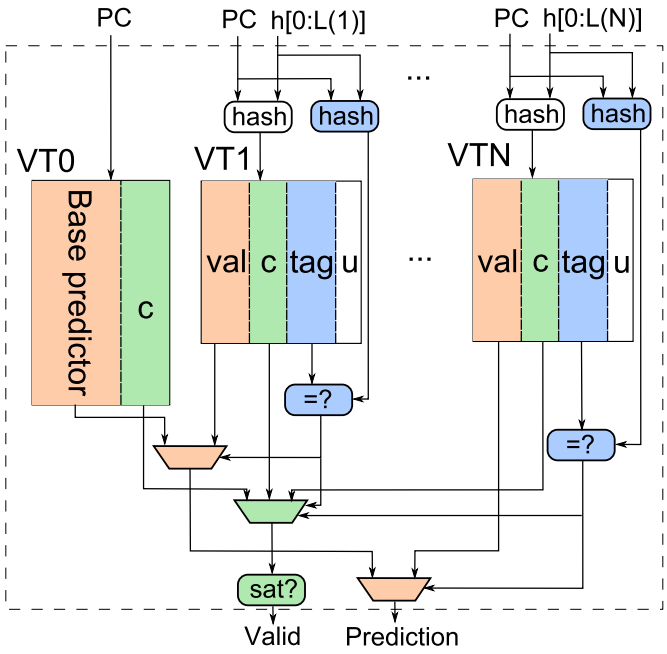


Figure 3.2: (1+N)-component VTAGE predictor. *Val* is the prediction, c is the hysteresis counter acting as confidence counter, u is the useful bit used by the replacement policy.

3.5 Using TAgged GEometric Predictors to Predict Values

3.5.1 The Value TAgged GEometric Predictor

Branch predictors exploiting correlations inside the global branch history have been shown to be efficient. Among these global history branch predictors, TAGE [SM06] is generally considered state-of-the-art. An indirect branch predictor ITTAGE [SM06] has been directly derived from TAGE. Since targets of indirect branches are register values, we have explored the possibility of adapting ITTAGE to Value Prediction. We refer to this modified ITTAGE as the *Value TAgged GEometric* history length predictor, VTAGE. To our knowledge, VTAGE is the first hardware value predictor to make extensive use of recent and less recent control-flow history. In particular, PS [NGS99] only uses a few bits of the global branch

history.

As it uses branch history to predict, we expect VTAGE to perform much better than other predictors when instruction results are indeed depending on the control flow. Nonetheless, VTAGE is also able to capture control-flow independent patterns as long as they are short enough with regard to the maximum history length used by the predictor. In particular, it can still capture short strided patterns, although space efficiency is not optimal since each value of the pattern will reside in an entry (contrarily to the Stride predictor where one pattern can be represented by a single entry).

Fig. 3.2 describes a (1+N)-component VTAGE predictor. The main idea of VTAGE (exactly like ITTAGE) is to use several value tables (VT), or *components*, storing predictions. Each table is indexed by a different number of bits of the global branch history, hashed with the PC of the instruction and a few bits of the path history (16 in our case). The different lengths form a geometric series (i.e. VT1 is accessed with two bits of the history, VT2 with four, VT3 with eight and so on). These tables are backed up by a base predictor – a tagless LVP predictor – which is accessed using the instruction address only. In VTAGE, an entry of a tagged component consists of a partial tag (blue), a 1-bit usefulness counter u (white) used by the replacement policy, a full 64-bit value *val* (orange), and a confidence/hysteresis counter c (green). An entry of the base predictor simply consists of the prediction (orange) and the confidence counter (green).

At prediction time, all components are searched in parallel to check for a tag match. The matching component accessed with the longest history is called the *provider* component as it will provide the prediction to the pipeline.

At update time, only the *provider* is updated. On either a correct or an incorrect prediction, c[5] and u[6] are updated. On a misprediction, *val* is replaced if c is equal to 0, and a new entry is allocated in a component using a longer history than the *provider*: All "upper" components are accessed to see if one of them has an entry that is not useful (u is 0). If none is found, the u bit of all matching entries in the upper components are reset, but no entry is allocated. Otherwise, a new entry is allocated in one of the components whose corresponding entry is not useful. The component is chosen randomly.

The main difference between VTAGE and ITTAGE is essentially the usage: The predicted value is used only if its confidence counter is saturated. We refer the reader to [SM06] for a detailed description of ITTAGE.

Lastly, as a prediction does not depend on previous values but only on previous control-flow, VTAGE can seamlessly predict instructions in tight loops and behaves like LVP in Fig. 3.1. However, due to index hash and multiplexing from multiple components, it is possible that its prediction latency will be higher,

[5] $c++$ if correct or $c = 0$ if incorrect. Saturating arithmetic.

[6] $u =$ *correct and the prediction of the next lower matching component is wrong.*

although this is unlikely to be an issue since prediction can span several cycles.

3.5.2 The Differential VTAGE Predictor

Aside from performance, VTAGE [PS14b] has several advantages over previously proposed value predictors. First, all predictions are independent, meaning that the predictor itself does not have to track in-flight predictions in the context of a wide instruction window processor. Second, and for the same reason, it does not suffer from a long prediction critical path and can handle back-to-back prediction of the same static instruction as is.

However, VTAGE is not suited to handle instructions whose results series exhibit strided patterns. For those instructions, each dynamic instance will occupy its own predictor entry, while a single entry is required for the whole pattern in a Stride predictor. As such, the prediction of strided patterns by VTAGE is not only unlikely but also space inefficient.

To overcome this limitation, VTAGE can be combined with a stride-based predictor (e.g., 2-delta Stride). However, the resulting hybrid predictor is space inefficient since both components are trained for all instructions. As a result, there is a call for a more efficient association of the two prediction schemes.

Moreover, a major shortcoming of VTAGE is that since it stores full 64-bit values in all its components, it requires a substantial amount of storage to perform well, i.e., to track enough instructions. This is not practical for implementation on actual silicon.

Based on these considerations, and building on the *Differential* FCM predictor of Goeman et al. [GVDB01], we propose the *Differential* VTAGE predictor, D-VTAGE. Instead of storing whole values, D-VTAGE stores – potentially much smaller – differences (strides) that will be added to last values in order to generate predictions.

In particular, VTAGE uses a tagless Last Value Predictor as its base component, while D-VTAGE uses a stride-based predictor (baseline Stride [GM98b] in our case). This component can be further divided into two parts. First, the Last Value Table (LVT) contains only retired last values. Second, the Base Predictor (VT0) contains the strides and a confidence estimation mechanism. Both tables are direct-mapped but we use small tags (5 bits) on the LVT to maximize accuracy.

A $1+n$-component D-VTAGE is depicted in Fig. 3.3, assuming one prediction per entry for clarity. Additional logic with regard to VTAGE is shaded in light blue on the figure. For each dynamic instruction, the last value will be read from the Last Value Table[7] (LVT, light blue). Then, the stride (orange) will be selected depending on whether a partially tagged component hits, following the regular

[7]Partial LVT tags are not shown for clarity.

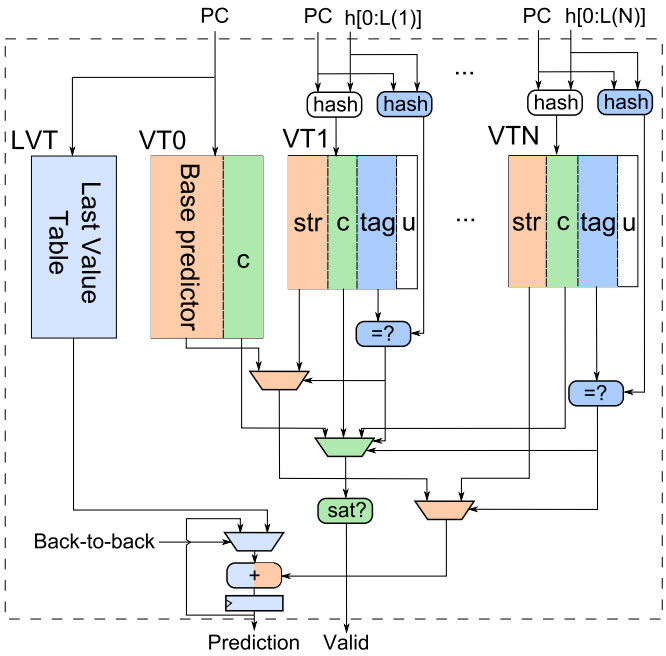


Figure 3.3: $1 + n$-component Differential Value TAGE predictor.

VTAGE operation [PS14b, SM06]. If the same instruction is fetched *twice in two consecutive cycles*, the prediction for the first one is bypassed to the input of the adders to be used as the last values for the second instance.

If the same instruction block is fetched *twice in a single cycle*, both instances can be predicted by using 3-input adders (not shown in Fig. 3.3).

The advantages of D-VTAGE are twofold. First, it is able to predict control-flow dependent patterns, strided patterns and *control-flow dependent strided patterns*. Second, it has been shown that short strides could capture most of the coverage of full strides in a stride-based value predictor [GVDB01]. As such using D-VTAGE instead of VTAGE would allow to greatly reduce storage requirements.

3.6 Evaluation Methodology

3.6.1 Value Predictors

3.6.1.1 Single Scheme Predictors

We study the behavior of several distinct value predictors in addition to VTAGE and D-VTAGE. Namely, LVP [LS96], the 2-delta Stride predictor (2D-Stride)

Table 3.1: Layout Summary. For VTAGE, *rank* is the position of the tagged component and varies from 1 to 6, 1 being the component using the shortest history length. History lengths are respectively 2, 4, 8, 16, 32 and 64.

Predictor	#Entries	Tag	Size (KB)
LVP [LS96]	8192	Full (51)	118.0
2D-Stride [EV93]	8192	Full (51)	246.0
o4-FCM [SS97a]	8192 (VHT)	Full (51)	115.0
	8192 (VPT)	-	67.0
o4-D-FCM [GVDB01]	8192 (VHT)	Full (51)	179.0
	8192 (VPT)	-	67.0
VTAGE [PS14b]	8192 (Base)	-	67.0
	6×1024	$12 + rank$	62.6
D-VTAGE [PS15a]	8192 (Base)	5	136.0
	6×1024	$12 + rank$	62.6

[EV93] as a representative of the stride-based predictor family[8] and a generic order-4 FCM predictor (o4-FCM) [SS97a], as well as an order-4 D-FCM predictor [GVDB01].

All predictors use 3-bit saturating counters as confidence counters. The prediction is used only if the confidence counter is saturated. Baseline counters are incremented by one on a correct prediction and reset on a misprediction. The predictors were simulated with and without FPC (See Section 3.4). We start from an 8K-entry LVP and derive the other predictors, each of them having 8K entries as we wish to gauge the prediction generation method, not space efficiency. Predictor parameters are illustrated in Table 3.1.

For VTAGE and D-VTAGE, we consider predictors featuring 6 tables in addition to a base component. The base component is a tagless LVP (respectively partially tagged stride) predictor. We use a single *useful* bit per entry in the partially tagged components and a 3-bit hysteresis/confidence counter c per entry in every component. The tag of tagged components is $12+rank$-bit long with *rank* varying between 1 and 6. The minimum and maximum history lengths are respectively 2 and 64 as we found that these values provided a good tradeoff in our experiments, for both predictors.

For o4-FCM and D-FCM, we use a hash function similar to those described in [SS98a] and used in [GVDB01]: For a n^{th} order predictor, we fold (XOR) each 64-bit history value upon itself to obtain a 16-bit index. Then, we XOR the

[8]To save space, we do not illustrate the Per-**P**ath **S**tride predictor [NGS99] that we initially included in the study. Performance was on par with 2D-Str.

most recent one with the second most recent one left-shifted by one bit, and so on. Even if it goes against the spirit of FCM, we XOR the resulting index with the PC in order to break conflicts as we found that too many instructions were interfering with each other in the VPT.

We consider that all predictors are able to predict instantaneously. As a consequence, they can seamlessly deliver their prediction before Dispatch. This also implies that o4-FCM and o4-D-FCM are – unrealistically – able to deliver predictions for two occurrences of the same instruction fetched in two consecutive cycles. Hence, their performance is most likely to be overestimated.

3.6.1.2 Hybrid Predictors

We also report numbers for a simple combination of VTAGE/2D-Str and o4-FCM/2D-Str, using the predictors described in Table 3.1. To select the prediction, we use a very simple mechanism: If only one component predicts (i.e., has high confidence), its prediction is naturally selected. When both predictors predict and if they do not agree, no prediction is made. If they agree, the prediction proceeds. Our hybrids also use the prediction of a component as the speculative last occurrence used by the other component to issue a prediction (i.e., use the last prediction of VTAGE as the next last value for 2D-Stride if VTAGE is more confident). When a given instruction retires, *all* components are updated with the committed value. To obtain better space efficiency, one can use a dynamic selection scheme such as the one Rychlik et al. propose. They assign a prediction to one component at most [RFK+98].

For the sake of clarity, we do not study *gDiff* [ZFC03] and DDISC [TF01] but we point out that they can be added on top of any other predictor. In particular, D-VTAGE, which, contrarily to FCM or any hybrid featuring FCM, would be feasible.

3.6.2 Simulator

In our experiments, we use a modified[9] version of the *gem5* cycle-accurate simulator [BBB+11]. Unfortunately, contrarily to modern x86 implementations, *gem5* does not support *move elimination* [FRPL05, JRB+98, PSR05], *μ-op fusion* [GRA+03a] and does not implement a *stack-engine* [GRA+03a]. Hence, its baseline IPC may be lower than what can commonly be observed on modern processors.

[9]Our modifications mostly lie with the ISA implementation. In particular, we implemented branches with a single μ-op instead of three and we removed some false dependencies existing between instructions due to the way flags are renamed/written.

Table 3.2: Simulator configuration overview. *not pipelined.

Front End	L1I 8-way 32KB, 1 cycle, Perfect TLB; 32B fetch buffer (two 16-byte blocks each cycle, potentially over one taken branch) w/ 8-wide fetch, 8-wide decode, 8-wide rename; TAGE 1+12 components 15K-entry total ($\simeq$ 32KB) [SM06], 20 cycles min. branch mis. penalty; 2-way 8K-entry BTB, 32-entry RAS.
Execution	192-entry ROB, 60-entry IQ unified, 72/48-entry LQ/SQ, 256/256 INT/FP registers; 1K-SSID/LFST Store Sets [CE98]; 6-issue, 4ALU(1c), 1MulDiv(3c/25c*), 2FP(3c), 2FP-MulDiv(5c/10c*), 2Ld/Str, 1Str; Full bypass; 8-wide WB, 8-wide retire.
Caches	L1D 8-way 32KB, 4 cycles, 64 MSHRs, 2 reads and 2 writes/cycle; Unified L2 16-way 1MB, 12 cycles, 64 MSHRs, no port constraints, Stride prefetcher, degree 8; All caches have 64B lines and LRU replacement.
Memory	Single channel DDR3-1600 (11-11-11), 2 ranks, 8 banks/rank, 8K row-buffer, tREFI 7.8us; Min. Read Lat.: 75 cycles, Max. 185 cycles.

We model a fairly aggressive pipeline: 4GHz, 8-wide, 6-issue[10] superscalar, out-of-order processor with a latency of 19 cycles. We chose a slow front-end (15 cycles) coupled to a swift back-end (4 cycles) to obtain a realistic misprediction penalty. Table 3.2 describes the characteristics of the pipeline we use in more details. As μ-ops are known at Fetch in *gem5*, all the width given in Table 3.2 are in μ-ops. We allow two 16-byte blocks of instructions to be fetched each cycle, potentially over a single taken branch. Independent memory instructions (as predicted by the Store Set predictor [CE98]) are allowed to issue out-of-order. Entries in the IQ are released upon issue (except with *selective replay*). Branches are resolved on data-speculative paths [SS98b].

3.6.2.1 Value Predictor Operation

The predictor makes a prediction at Fetch for every μ-op (we do not try to estimate criticality or focus only on load instructions) producing an integer register explicitly used by subsequent μ-ops. In particular, branches are *not* predicted with the value predictor but values feeding into branch instructions are. To index the predictors, we XOR the PC of the x86 instruction with the μ-op number inside the x86 instruction to avoid all μ-ops mapping to the same entry. This mechanism ensures that most μ-ops of a macro-op will generate a different predictor index and therefore have their own entry in the predictor. We assume that the predictor can deliver as many predictions as requested by the Fetch stage. A prediction is written into the register file and replaced by its non-speculative counterpart when it is computed. However, a real implementation of VP need not use this exact mechanism, as discussed in Section 3.3.

Finally, for validation at Commit, we assume the presence of a ROB-like queue (i.e., FIFO) where predictions are pushed at Fetch and popped at Commit. To

[10]On our benchmark set and with our baseline simulator, an 8-issue machine achieves only marginal speedup over this baseline.

validate, the actual result is read from the PRF and the prediction from the head of the queue. Note that **all** predictions are pushed in the queue, even if they are not used by the execution engine. This is required to train the predictor by determining if the prediction was correct at validation time, and update the associated confidence counter accordingly.

x86 Flags In the x86_64 ISA, some instructions write flags based on their results while some need them to execute (e.g., branches) [Int12]. Therefore, the flags should generally be predicted along the values to avoid stalling dependents. This is particularly true for the *Late Execution* hardware that we will present in the next Chapter. Therefore, we assume that flags are computed as the last step of Value Prediction, based on the predicted value. In particular, the *Zero Flag* (ZF), *Sign Flag* (SF) and *Parity Flag* (PF) can easily be inferred from the predicted result. Remaining flags – *Carry Flag* (CF), *Adjust Flag* (AF) and *Overflow Flag* (OF) – depend on the operands and cannot be inferred from the predicted result only. We found that always setting the *Overflow Flag* to 0 did not cause many mispredictions and that setting CF if SF was set was a reasonable approximation. The *Adjust Flag*, however, cannot be set to 0 or 1 in the general case. This is a major impediment to the value predictor coverage since we consider a prediction as incorrect if a single one of the derived flags – thus the flag register – is wrong. Fortunately, x86_64 forbids the use of decimal arithmetic instructions. As such, AF is not used and we can simply ignore its correctness when checking for a misprediction [Int12].

Free *Load Immediate* Prediction Our implementation of Value Prediction features write ports available at dispatch time to write predictions to the PRF. Thus, it is not necessary to predict *load immediate* instructions[11] since their *actual* result is available in the front-end. The predictor need not be trained for these instructions, and they need not be validated or even dispatched to the IQ. Their result simply has to be written to the PRF as if it were a value prediction. Due to this optimization, the addition of a specific table that would only predict small constants [SA02] may become much less interesting.

3.6.2.2 Misprediction Recovery

We illustrate two possible recovery scenarios, squashing at commit time and very idealistic selective replay. In both scenarios, recovery is unnecessary if the prediction of instruction I was wrong but no dependent instruction has been issued before the execution of I, since the prediction is replaced by the effective result at execution time. This removes useless squashes and is part of our implementation.

[11] With the exception of *load immediate* instructions that write to a partial register [Int12].

For selective replay, we assume an idealistic mechanism where recovery is immediate. All value-speculatively issued instructions stay in the IQ until they become non speculative (causing more pressure on the IQ). When a value misprediction is found, the IQ and LSQ are searched for dependents to reissue. Ready instructions can be rescheduled in the same cycle with respect to the issue width.

Yet, even such an idealistic implementation of selective replay does not only offer performance advantages. In particular, it can only inhibit an entry responsible for a wrong prediction at update time. Consequently, in the case of tight loops, several occurrences of the same instruction can be in-flight, and a misprediction for the first occurrence will often result in mispredictions for subsequent occurrences, causing multiple replays until the first misprediction retires.

In any case, the 0-cycle replay mechanism is overly optimistic because of the numerous and complex steps selective replay implies. Our experiments should be considered as the illustration that even a perfect mechanism would not significantly improve performance compared to squashing at commit time as long as the predictor is very accurate.

3.6.3 Benchmark Suite

We use a subset of the the SPEC'00 [Staa] and SPEC'06 [Stab] suites to evaluate our contributions as we focus on single-thread performance. Specifically, we use 18 integer benchmarks and 18 floating-point programs.[12] Table 3.3 summarizes the benchmarks we use as well as their input, which are part of the *reference* inputs provided in the SPEC software packages. To get relevant numbers, we identify a region of interest in the benchmark using Simpoint 3.2 [PHC03]. We simulate the resulting slice in two steps: 1) Warm up all structures (caches, branch predictor and value predictor) for 50M instructions, then collect statistics (speedup, coverage, accuracy) for 100M instructions.

3.7 Simulation Results

3.7.1 General Trends

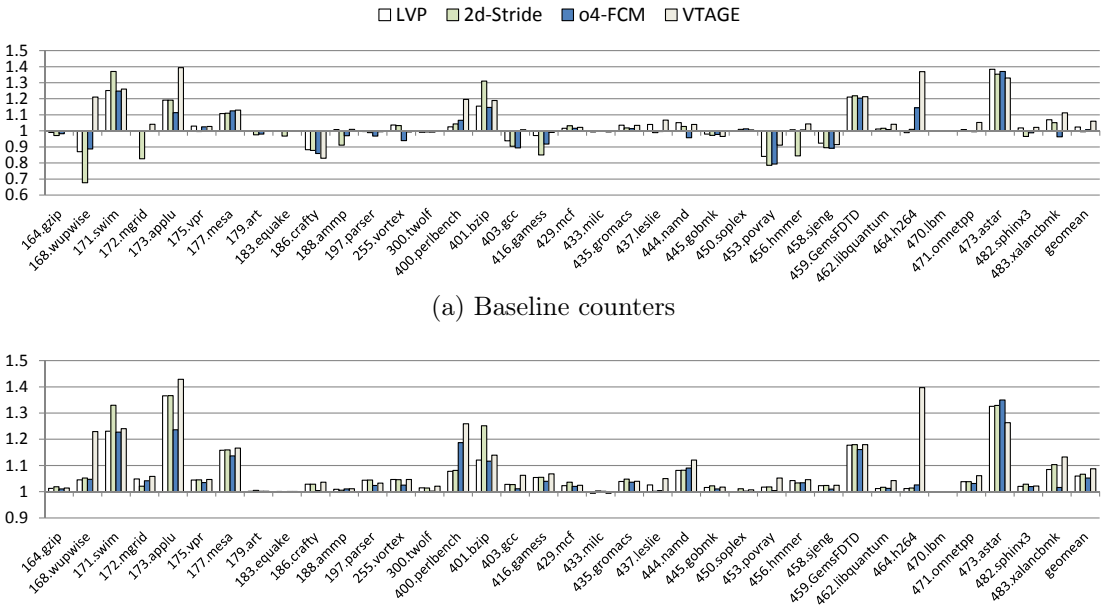
3.7.1.1 Forward Probabilistic Counters

Fig. 3.4 illustrates speedup over baseline using *squashing* at Commit to repair a misprediction. First, Fig. 3.4 (a) suggests that the simple 3-bit counter confidence estimation is not sufficient. Accuracy is generally comprised between 0.94 and almost 1.0, yet fairly important slowdowns can be observed. Much better

[12] We do not use the whole suites due to some currently missing system calls and x87 instructions in *gem5-x86*.

Table 3.3: Benchmarks used for evaluation. Top: CPU2000, Bottom: CPU2006. INT: 18, FP: 18, Total: 36.

Program	Input	IPC (6-issue)
164.gzip (INT)	input.source 60	0.841
168.wupwise (FP)	wupwise.in	1.303
171.swim (FP)	swim.in	1.745
172.mgrid (FP)	mgrid.in	2.364
173.applu (FP)	applu.in	1.481
175.vpr (INT)	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2	0.663
177.mesa (FP)	-frames 1000 -meshfile mesa.in -ppmfile mesa.ppm	0.951
179.art (FP)	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10	0.442
183.equake (FP)	inp.in	0.656
186.crafty (INT)	crafty.in	1.562
188.ammp (FP)	ammp.in	1.260
197.parser (INT)	ref.in 2.1.dict -batch	0.455
255.vortex (INT)	lendian1.raw	1.481
300.twolf (INT)	ref	0.259
400.perlbench (INT)	-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1	1.395
401.bzip2 (INT)	input.source 280	0.700
403.gcc (INT)	166.i	0.998
416.gamess (FP)	cytosine.2.config	1.697
429.mcf (INT)	inp.in	0.111
433.milc (FP)	su3imp.in	0.503
435.gromacs (FP)	-silent -deffnm gromacs -nice 0	0.770
437.leslie3d (FP)	leslie3d.in	2.157
444.namd (FP)	namd.input	1.760
445.gobmk (INT)	13x13.tst	0.716
450.soplex (FP)	-s1 -e -m45000 pds-50.mps	0.273
453.povray (FP)	SPEC-benchmark-ref.ini	1.468
456.hmmer (INT)	nph3.hmm	2.039
458.sjeng (INT)	ref.txt	1.145
459.GemsFDTD (FP)	/	2.148
462.libquantum (INT)	1397 8	0.459
464.h264ref (INT)	foreman_ref_encoder_baseline.cfg	0.977
470.lbm (FP)	reference.dat	0.381
471.omnetpp (INT)	omnetpp.ini	0.301
473.astar (INT)	BigLakes2048.cfg	1.165
482.sphinx3 (FP)	ctlfile . args.an4	0.802
483.xalancbmk (INT)	-v t5.xml xalanc.xsl	1.831



(a) Baseline counters

(b) FPC

Figure 3.4: Speedup over baseline for different predictors using validation at Commit and refetch to recover from a value misprediction.

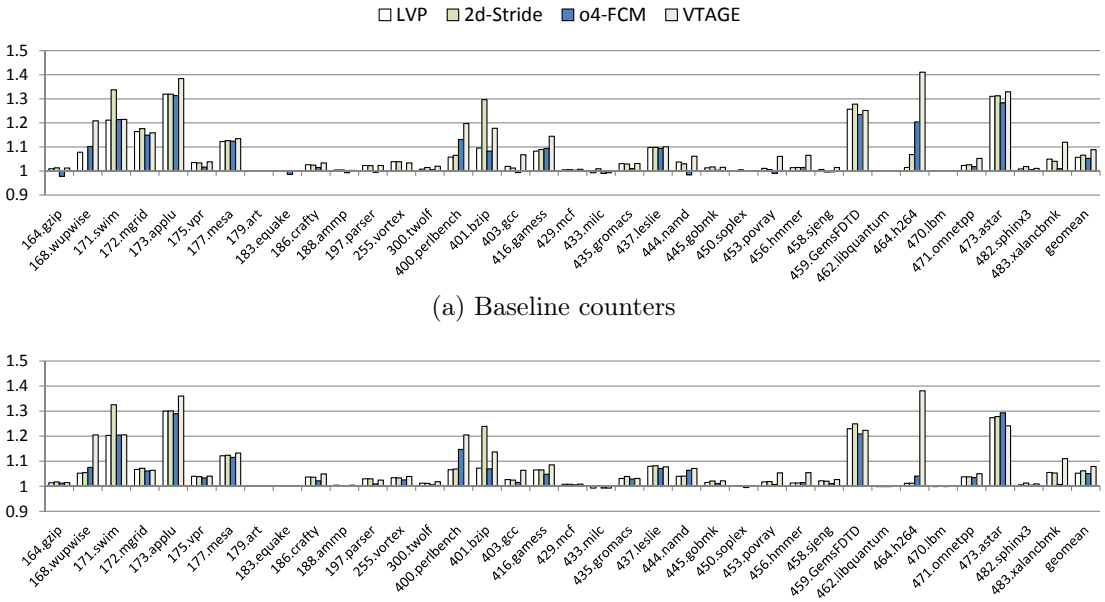
accuracy is attained in (b) using FPC (above 0.997 in all cases), and translates to performance gain when using VP. This demonstrates that by pushing accuracy up, VP yields performance increase even with a pessimistic – but much simpler – recovery mechanism.

Fig. 3.4 (b) also shows that from a performance standpoint, no single-scheme predictor plainly outperforms the others. In *bzip* and *astar*, 2D-Stride achieves higher performance than VTAGE, and in others, VTAGE performs better (*wupwise, applu, perlbench, gcc, gamess, namd* and *h264*). This advocates for hybridization.

Nonetheless, if VTAGE is outperformed by a computational predictor in some cases, it generally appears as a simpler *and* better context-based predictor than our implementation of o4-FCM. In our specific case, o4-FCM suffers mostly from a lack of coverage stemming from the use of a stricter confidence mechanism (FPC) and from needing more time to learn patterns.

3.7.1.2 Pipeline Squashing vs. selective replay

Fig. 3.5 illustrates experiments similar to those of Fig. 3.4 except selective replay is used to repair value mispredictions. On the one hand, selective replay significantly decreases the misprediction penalty, therefore, performance gains are generally obtained even without FPC because less coverage is lost due to the confidence



(a) Baseline counters

(b) FPC

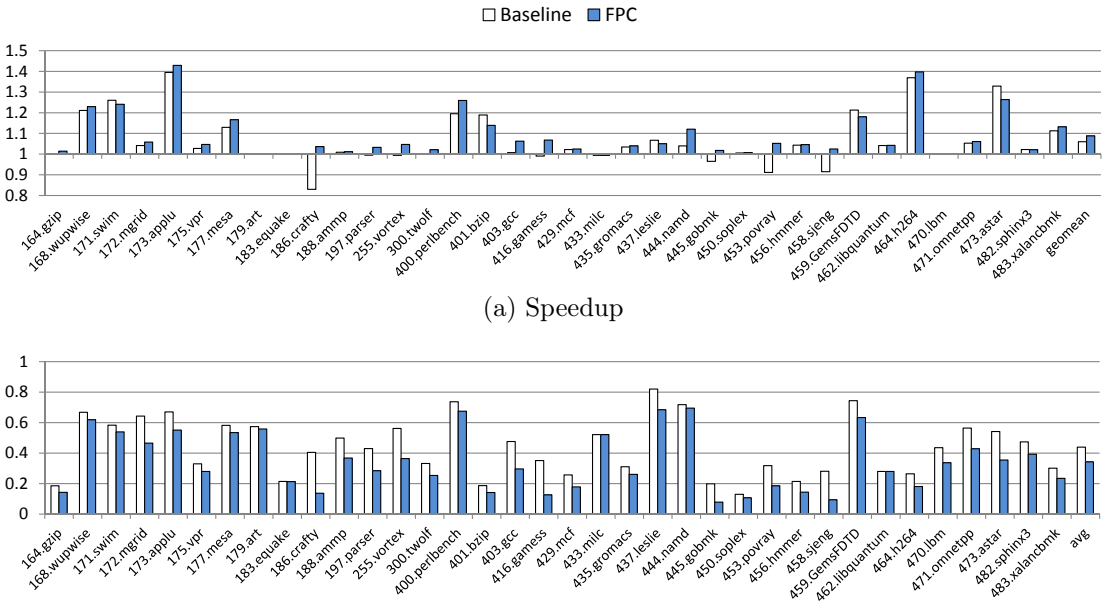
Figure 3.5: Speedup over baseline for different predictors using selective replay to recover from a value misprediction.

estimation mechanism. Yet, this result has to be pondered by the fact that our implementation of selective replay is idealistic.

On the other hand, with FPC, we observe that the recovery mechanism has little impact since the speedups are very similar in Fig. 3.4 (b) and Fig. 3.5 (b). In other words, provided that a confidence scheme such as FPC yields very high accuracy, even an optimistic implementation of a very complex recovery mechanism will not yield a significant performance increase. It is only if such confidence scheme is not available that selective replay becomes interesting. However, Zhou et al. state that even a single-cycle recovery penalty – which is more conservative than our implementation – can nullify speedup if accuracy is too low (under 85%) [ZYFRC00].

3.7.1.3 Prediction Coverage and Performance

Prediction accuracy when using FPC is widely improved with it being higher than 0.997 for every benchmark. This accuracy improvement comes with a reduction of the predictor coverage, especially for applications for which the accuracy of the baseline predictor was not that high. This is illustrated in Fig. 3.6 for VTAGE: The greatest losses of coverage in (b) correspond to the applications that have the lowest performance in (a), i.e. *crafty, vortex, gamess, gobmk, povray, sjeng.* These applications are also those for which a performance loss was encountered for the



(a) Speedup

(b) Coverage

Figure 3.6: Speedup and coverage of VTAGE with and without FPC. The recovery mechanism is squashing at Commit.

baseline while no performance is lost with FPC. For applications already having high accuracy and showing performance increase with the baseline, coverage is also slightly decreased, but in a more moderate fashion. Similar behaviors are observed for the other predictors.

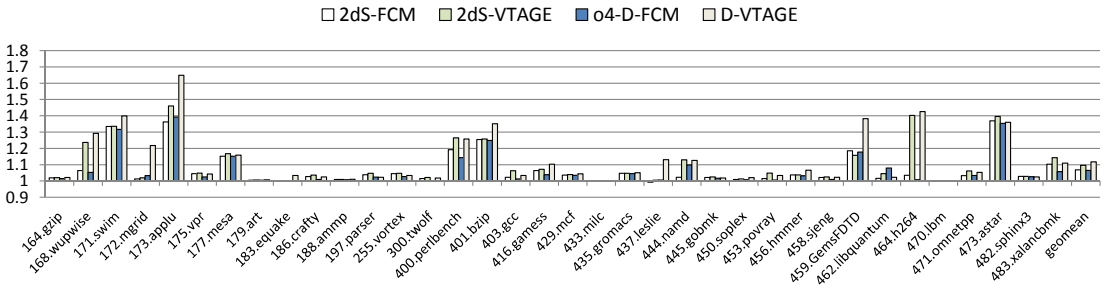
Note that high coverage does not always correlate with high performance, e.g. *leslie* exhibits 70-80% coverage but marginal speedup. Conversely, a smaller coverage may lead to significant speed-up e.g., *h264*.

3.7.2 Hybrid predictors

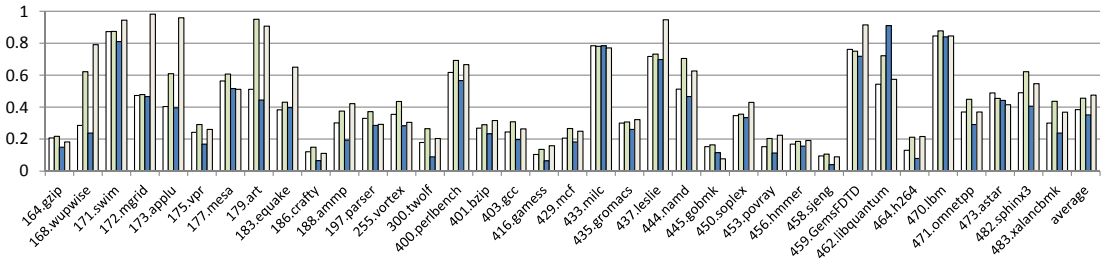
For the sake of readability, we only report results for the squashing recovery mechanism, but trends are similar for selective replay.

Fig. 3.7 (a) illustrates speedups for a combination of VTAGE and 2D-Stride as well as o4-FCM and 2D-Stride, and more tightly-coupled hybrid: D-FCM and D-VTAGE. As expected, simple hybrids yield slightly higher performance than single predictor schemes, generally achieving performance at least on par with the best of the two predictors on a given benchmark. Yet, improvement over this level is generally marginal.

Interestingly, we can observe that D-VTAGE is able to provide better performance than its naive hybrid equivalent. As a result, a more tightly-coupled hybrid



(a) Speedup



(b) Coverage

Figure 3.7: Speedup over baseline and coverage of a 2-component symmetric hybrid made of 2D-Stride and VTAGE/FCM, as well as D-FCM and D-VTAGE. All predictors feature FPC for confidence estimation. Recovery is done via squash at Commit.

of VTAGE and Stride seems more interesting from a performance standpoint.

Therefore, since D-FCM suffers from the same shortcomings as FCM (prediction critical loop and complex management of the speculative history), D-VTAGE appears as a better choice overall.

3.8 Conclusion

In this Chapter, we have first shown that the use of Value Prediction can be effective, even if prediction validation is performed at commit time. Using probabilistic saturating confidence counters with a low increment probability can greatly benefit any value predictor: A very high accuracy ($> 99.7\%$) can be ensured for all existing value predictors at some cost in coverage, using very simple hardware. Very high accuracy is especially interesting because it allows to tolerate the high individual average misprediction cost associated with validation at commit time. In this context, complex recovery mechanisms such as selective replay have very marginal performance interest. In other words, we claim that, granted a proper confidence estimation mechanism such as the FPC scheme, state-of-the-art value predictors can improve performance on a fairly wide and deep pipeline while the prediction validation is performed at commit time. In this case, the Value Prediction hardware is essentially restricted to the in-order front-end (prediction) and the in-order back-end (validation and training) and the modifications to the out-of-order engine are very limited.

Second, we have introduced a new context-based value predictor, the Value TAGE predictor, and an improved version able to capture strided patterns, D-VTAGE. We derived VTAGE from the ITTAGE predictor by leveraging similarities between Indirect Branch Target Prediction and Value Prediction. VTAGE uses the global branch history as well as the path history to predict values. We have shown that it outperforms previous generic context-based value predictors leveraging per-instruction value history (e.g., FCM). A major advantage of VTAGE is its tolerance to high latency which has to be contrasted with local value history based predictors as they suffer from long critical operation when predicting consecutive instances of the same instruction. They are highly sensitive to the prediction latency and cannot afford long lookup times, hence large tables.

On the contrary, the access to VTAGE tables can span over several cycles (from Fetch to Dispatch) and VTAGE can be implemented using very large tables. D-VTAGE follows VTAGE operation except only the stride is selected using the TAGE scheme. As a result, D-VTAGE possesses the same prediction critical path and requirement for a speculative window as any stride-based predictors. Fortunately, contrarily to FCM predictors, the critical path is only a

64-bit adder. Moreover, we propose a realistic implementation of the speculative window in Chapter 5. As a result, we claim that D-VTAGE is suited for actual implementations.

Through combining our two propositions, a practical value predictor improves performance while only requiring limited hardware modifications to the out-of-order execution engine. Our experiments show that while improvement might be limited in many applications – less than 5% on 21 out of 38 benchmarks –, encouraging performance gains are encountered in remaining applications: From 6.6% up to 64.8% on 17 benchmarks. Yet, this contribution does not address the requirements on the Physical Register File. Indeed, we have considered that up to 8 predictions were written to the PRF at Dispatch, and although banking the PRF can limit the number of write ports required per bank, those ports are still required *in addition* to those required by the execution engine. Moreover, validating predictions at Commit still implies that the actual result should be read from the PRF and compared against the associated prediction, that can sit in a FIFO queue similar to the ROB. Yet, this also entails additional read ports. As a result, and given the poor scalability of the PRF as the port count grows [ZK98], VP cannot be envisioned if the port requirements on the PRF is not kept reasonable. In the next Chapter, we will show that it is actually possible to implement VP with as many ports on the PRF as a regular out-of-order processor, paving the way for an actual implementation of VP on real silicon.

Chapter 4

A New Execution Model Leveraging Value Prediction: EOLE

This Chapter covers a publication in the International Symposium on Computer Architecture (ISCA 2014) [PS14a] that was also selected to appear in IEEE Micro as a Top Pick from Computer Architecture conferences of 2014 [PS15b].

4.1 Introduction & Motivations

Even in the multicore era, the need for higher single thread performance is driving the definition of new high-performance cores. Although the usual superscalar design does not scale, increasing the ability of the processor to extract Instruction Level Parallelism (ILP) by increasing the window size as well as the issue width[1] has generally been the favored way to enhance sequential performance. For instance, consider the recently introduced Intel Haswell micro-architecture that has 33% more issue capacity than Intel Nehalem.[2] To accommodate this increase, both the Reorder Buffer (ROB) and Scheduler size were substantially increased.[3] On top of this, modern schedulers must support complex mechanisms such as *speculative scheduling* to enable back-to-back execution of variable latency instructions and their dependents, and thus a form of replay to efficiently recover from schedule mispredictions [PSM+15, MBS07, MS01, MSBU00, SGC01b, MLO01, KL04].

In addition, the issue width impacts other structures: The Physical Register File (PRF) must provision more read/write ports as the width grows, while the number of physical registers must also increase to accommodate the ROB size.

[1] How many instructions can be sent to the functional units to be executed each cycle.
[2] State-of-the-art in 2009
[3] From respectively 128 and 36 entries to 192 and 60 entries.

Because of this, both latency and power consumption increase and using a monolithic register file rapidly becomes complexity-ineffective. Similarly, a wide-issue processor should provide enough functional units to limit resource contention. Yet, the complexity of the bypass network grows quadratically with the number of functional units and quickly becomes critical regarding cycle time [PJS97]. In other words, the out-of-order engine impact on power consumption and cycle time is ever increasing [EA02].

We previously mentioned that in the context of Value Prediction, validation and recovery from a misprediction can be done at commit time, but predictions must still be written to the PRF at Dispatch. As a result, additional ports are required on the register file, which may lead to an unbearably large and power-hungry structure, especially as the issue width is already high (e.g., 8 in Haswell).

To that extent, we propose a modified superscalar design, the *{Early | Out-of-Order | Late} Execution* microarchitecture, *EOLE*. EOLE builds on a pipeline implementing Value prediction and aims to further reduce both the complexity of the out-of-order (OoO) execution engine and the number of ports required on the PRF when VP is implemented. We achieve this reduction without significantly impacting overall performance. The contribution is therefore twofold: First, EOLE paves the way to truly practical implementations of VP. Second, it reduces complexity in the most complicated and power-hungry part of a modern OoO core.

To do so, we rely on two observations. First, when using VP, a significant amount of single-cycle instructions have their operands ready in the front-end thanks to the value predictor. As such, we introduce *Early Execution* to execute single-cycle ALU instructions in-order in parallel with Rename by using predicted and/or immediate operands. Early-executed instructions are **not** sent to the OoO Scheduler. Validation at Commit [PS14b] guarantees that the operands of committed early-executed instructions were the correct operands. Early Execution requires simple hardware and reduces pressure on the OoO instruction window.

Second, and also thanks to the fact that predicted results can be validated outside the OoO engine at commit time, we can offload the execution of predicted single-cycle ALU instructions to some dedicated in-order *Late Execution* pre-commit stage, where no dynamic scheduling has to take place. This does not hurt performance since instructions dependent on predicted instructions will simply use the predicted results rather than wait in the OoO Scheduler. Similarly, the resolution of high confidence branches can be offloaded to the Late Execution stage since they are very rarely mispredicted [Sez11c].

Overall, a total of 7% to 75% (37% on average) of the retired instructions can be offloaded from the OoO core, on our benchmark suite. As a result, EOLE should benefit from both the aggressiveness of modern OoO designs and the higher energy-efficiency of more conservative in-order designs.

We evaluate EOLE against a baseline OoO model featuring VP and show that it achieves similar levels of performance having only 66% of the baseline issue capacity and a significantly less complex physical register file. This is especially interesting since it provides architects extra design headroom in the OoO engine to implement new architectural features.

4.2 Related Work on Complexity-Effective Architectures

Many propositions aim to reduce complexity in modern superscalar designs. In particular, it has been shown that most of the complexity and power consumption reside in the OoO engine, including the PRF [WB96], Scheduler and bypass network [PJS97]. As such, previous studies focused on either reducing the complexity of existing structures, or in devising new pipeline organizations.

4.2.1 Alternative Pipeline Design

Farkas et al. propose the *Multicluster* architecture in which execution is distributed among several execution clusters, each of them having its own register file [FCJV97]. Since each cluster is simpler, cycle time can be decreased even though some inefficiencies are introduced due to the inter-cluster dependencies. In particular, inter-cluster data dependencies are handled by dispatching the same instruction to several clusters and having all instances but one serve as inter-cluster data-transfer instructions while a single instance actually computes the result. To enforce correctness, this instance is data-dependent on all others.

The Alpha 21264 [KMW98] is an example of real-world clustered architecture and shares many traits with the *Multicluster* architecture. It features two symmetric clusters sharing the frontend (i.e., Fetch to Rename) and the store queue. Instructions are then *steered* to one of them depending on a given heuristic.

Palacharla et al. introduce a *dependence-based* microarchitecture where the centralized instruction window is replaced by several parallel FIFOs [PJS97]. FIFOs are filled with independent instructions (i.e., all instructions in a single FIFO are dependent) to allow ILP to be extracted: since instructions at the head of the FIFOs are generally independent, they can issue in parallel.

This greatly reduces complexity since only the head of each FIFO has to be selected by the *Select* logic. They also study a *clustered dependence-based* architecture to reduce the amount of bypass and window logic by using clustering. That is, a few FIFOs are grouped together and assigned their own copy of the register file and bypass network, mimicking a wide issue window by having several smaller ones. Inter-cluster bypassing is naturally slower than intra-cluster

bypassing.

Tseng and Patt propose the *Braid* architecture [TP08], which shares many similarities with the *clustered dependence-based* architecture with three major differences: 1) Instruction partitioning – steering – is done at compile time via dataflow-graph coloring 2) Each FIFO is a narrow-issue (dual issue in [TP08]) in-order cluster called a *Braid Execution Unit* with its own local register file, execution units, and bypass network 3) A global register file handles inter-unit dependencies instead of an inter-cluster bypass network. As such, they obtain performance on-par with an accordingly sized OoO machine without using complicated scheduling logic, register file and bypass logic.

Austin proposes *Dynamic Implementation VAlidation* (DIVA) to check instruction results just before commit time, allowing the core to be faulty [Aus99]. DIVA can greatly reduce the complexity of *functionally validating* the core since only the checker has to be functionally correct. The core complexity itself can remain very high, however. An interesting observation is that the latency of the checker has very limited impact on performance. This hints that adding pipeline stages between Writeback and Commit does not actually impact performance much, as long as the instruction window is large enough.

Fahs et al. study *Continuous Optimization* where common compile-time optimizations are applied dynamically in the Rename stage [FRPL05]. This allows to early-execute some instructions in the front-end instead of the OoO core. Similarly, Petric et al. propose RENO which also dynamically applies optimizations at rename-time [PSR05].

4.2.2 Optimizing Existing Structures

Instead of studying new organizations of the pipeline, Kim and Lipasti present the *Half-Price Architecture* [KL03] in which regular structures are made operand-centric rather than instruction-centric. For instance, they argue that many instructions are single operand and that both operands of dual-operand instructions rarely become ready at the same time. Thus, the load capacitance on the tag broadcast bus can be greatly reduced by *sequentially waking-up* operands. In particular, the left operand is woken-up as usual, while the right one is woken-up one cycle later by inserting a latch on the broadcast bus. They use a criticality predictor to place the critical tag in the left operand field.

Similarly, Ernst and Austin propose *Tag Elimination* to limit the number of comparators used for Wakeup [EA02]. Only the tag of the – predicted – last arriving operand is put in the scheduler. This allows to only use one comparator per entry instead of one per operand (e.g., 4 per instruction in AMD Bulldozer [GAV11]). However, this also implies that the instruction must be replayed on a wrong last operand prediction.

Regarding the register file, Kim and Lipasti also observe in their *Half-Price Architecture* that many issuing instructions do not need to read both their operands in the register file since one or both will be available on the bypass network [KL03]. Thus, provisioning two read ports per issue slot is generally over-provisioning. In particular, for the small proportion of instructions that effectively need to read two operands in the RF, they propose to use a single port but use it for two consecutive cycles. Reducing the number of ports drastically reduces the complexity of the register file as ports are much more expensive than registers [ZK98].

Lastly, many studies on *heterogeneous* architectures have been done in the recent past, although most of them focus on heterogeneous multicores, which are out of the scope of this work. Nonetheless, Lukefahr et al. propose to implement two back-ends – in-order and OoO – in a single core [LPD+12] and to dynamically dispatch instructions to the most adapted one. In most cases, this saves power at a slight cost in performance. In a sense, EOLE has similarities with such a design since instructions can be executed in different locations. However, no decision has to be made as the location where an instruction will be executed depends only on its type and status (e.g., predicted or not).

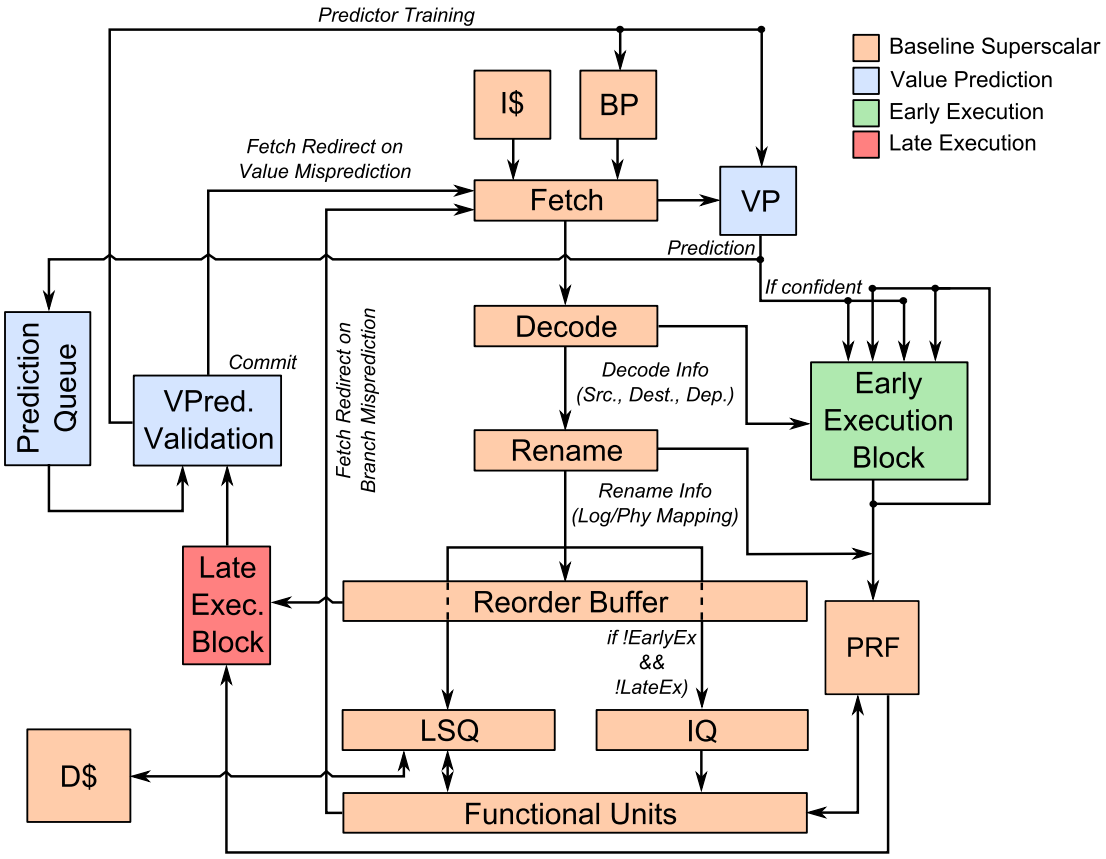
4.3 {Early | Out-of-order | Late} Execution: EOLE

4.3.1 Enabling EOLE Using Value Prediction

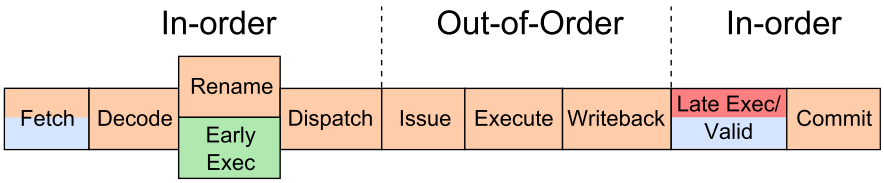
As previously described, EOLE consists of a set of simple ALUs in the in-order front-end to early-execute instructions in parallel with Rename, and a second set in the in-order back-end to late-execute instructions just before they are validated and committed. With EOLE, instructions are either executed early, as usual in the out-of-order engine, or late, but at most **once** (there is no re-execution).

While EOLE is heavily dependent on Value Prediction, they are in fact complementary features. Indeed, the former needs a value predictor to predict operands for Early Execution and provide temporal slack for Late Execution, while Value Prediction needs EOLE to reduce PRF complexity and thus become truly practical. Yet, to be implemented, EOLE requires prediction validation to be done at Commit since validating at Execute mechanically forbids Late Execution. Furthermore, using selective replay to recover from a value misprediction [MBS07, MS01, MSBU00, SGC01b, MLO01, KL04] nullifies the interest of both Early and Late Execution as all instructions must flow through the OoO Scheduler in case they need to be replayed. Hence, squashing must be used to recover from a misprediction so that early/late-executed instructions can safely bypass the Scheduler.

Fortunately, we previously described a confidence estimation mechanism greatly limiting the number of value mispredictions, *Forward Probabilistic Counters* (FPC)



(a) Block diagram.



(b) Pipeline diagram.

Figure 4.1: The EOLE μ-architecture.

[PS14b]. With FPC, the cost of a single misprediction can be high since mispredicting is very rare. Thus, validation can be done late – at commit time – and squashing can be used as the recovery mechanism. This enables the implementation of both Early and Late Execution, hence EOLE.

By eliminating the need to dispatch and execute many instructions in the OoO engine, EOLE substantially reduces the pressure on complex and power-hungry structures. Thus, those structures can be scaled down, yielding a less complex architecture whose performance is on par with a more aggressive design. Moreover, doing so is orthogonal to previously proposed mechanisms such as *clustering* [FCJV97, KMW98, PJS97, STR02] and does not *require* a centralized instruction window, even though this is the model we use in this work. Figure 4.1 depicts the EOLE architecture, implementing both Early Execution (green), Late Execution (red) and Value Prediction (blue). In the following paragraphs, we detail the two additional blocks required to implement EOLE and their interactions with the rest of the pipeline.

4.3.2 Early Execution Hardware

The core idea of Early Execution (EE) is to position one or more ALU stages in the front-end in which instructions with available operands will be executed. For complexity concerns, however, it seems necessary to limit Early Execution to single-cycle ALU instructions. Indeed, implementing complex functional units in the front-end to execute multi-cycle instructions does not appear as a worthy tradeoff. In particular, memory instructions are not early-executed (i.e., no memory accesses). Early Execution is done in-order, hence, it does not require renamed registers and can take place in parallel with Rename. For instance, Figure 4.2 depicts the Early Execution Block adapted to a 2-wide Rename stage.

Renaming can be pipelined over several cycles [SMV11]. Consequently, we can use several ALU stages and simply insert pipeline registers between each stage. The actual execution of an instruction can then happen in any of the ALU stages, depending on the readiness of its operands coming from Decode (i.e., immediate), the local[4] bypass network (i.e., from instructions early-executed in the previous cycle) or the value predictor. Operands are **never** read from the PRF.

In a nutshell, all eligible instructions flow through the ALU stages, propagating their results in each bypass network accordingly once they have executed. Finally, after the last stage, results as well as predictions are written into the PRF.

[4]For complexity concerns, we consider that bypass does not span several stages. Consequently, if an instruction depends on a result computed by an instruction located two rename-groups ahead, it will not be early-executed.

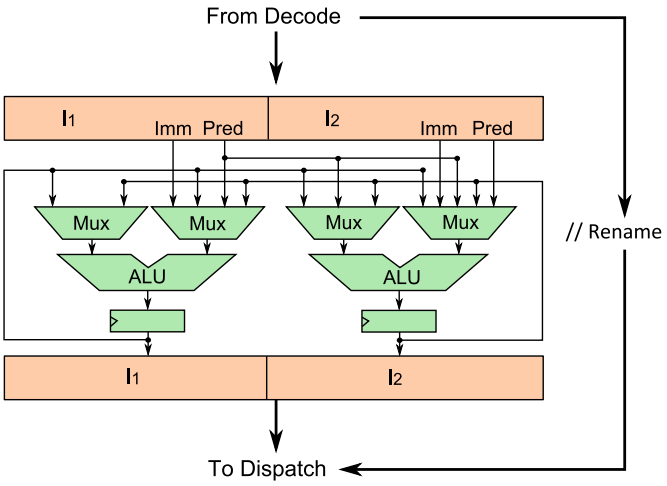


Figure 4.2: Early Execution Block. The logic controlling the ALUs and muxes is not shown for clarity.

An interesting design concern lies with the number of stages required to capture a reasonable proportion of instructions. We actually found that using more than a single stage was highly inefficient, as illustrated in Figure 4.3. This figure shows the proportion of committed instructions eligible for Early Execution for a baseline 8-wide rename, 6-issue model (see Table 3.2), using the D-VTAGE predictor described in Table 3.1. As a result, in further experiments, we consider a 1-deep Early Execution Block only.

To summarize, Early Execution only requires a single new computing block, which is shown in green in Figure 4.1. The mechanism we propose does not require any storage area for temporaries as all values are living inside the pipeline registers or the local bypass network. Finally, since we execute in-order, each instruction is mapped to a single ALU and scheduling is straightforward.

4.3.3 Late Execution Hardware

Late Execution (LE) targets instructions whose result has been predicted.[5] It is done in-order just before validation time, that is, outside of the execution engine. As for Early Execution, we limit ourselves to single-cycle ALU instructions to minimize complexity. That is, predicted loads are executed in the OoO engine, but validated at commit.

Interestingly, Seznec showed in [Sez11c] that conditional branch predictions flowing from the TAGE [SM06, Sez11b] branch predictor can be categorized such

[5]Instructions eligible for prediction are μ-ops producing a 64-bit or less result that can be read by a subsequent μ-op, as defined by the ISA implementation.

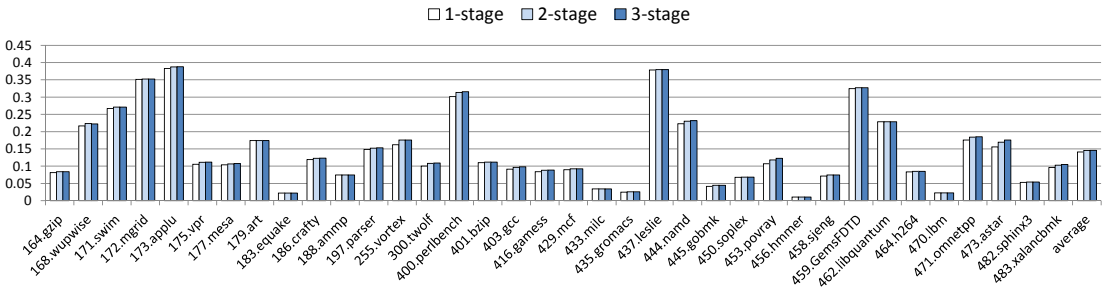


Figure 4.3: Proportion of committed instructions that can be early-executed, using one, two and three ALU stages and a D-VTAGE predictor.

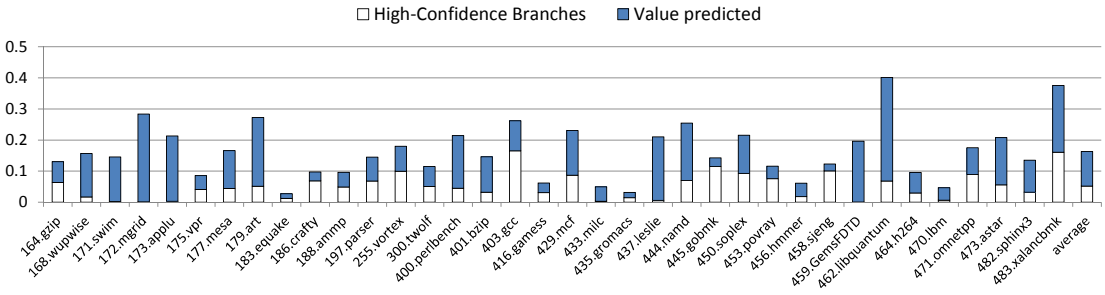


Figure 4.4: Proportion of committed instructions that can be late-executed using a D-VTAGE predictor. Late-executable instructions that can also be early-executed are not counted since instructions are executed once at most.

that very high confidence predictions are known. Since high confidence branches exhibit a misprediction rate generally lower than 0.5%, resolving them in the Late Execution block has a marginal impact on overall performance. Thus, we consider both single-cycle predicted ALU instructions and very high confidence branches[6] for Late Execution. In this study, we did not try to set confidence on the other types of branches (indirect jumps, returns). Yet, provided a similar high confidence estimator for these categories of branches, one could postpone the resolution of high confidence ones until the LE stage.

Furthermore, note that predicted instructions can also be early-executed. In that event, they only need to be validated in case another early-executed instruction from the same rename-group used the prediction as an operand.

In any case, Late Execution further reduces pressure on the OoO engine in terms of instructions dispatched to the Scheduler. As such, it also removes the need for predicting only critical instructions [FRB01, RFK+98, TTC02] since minimizing the number of instructions flowing through the OoO engine requires maximizing the number of predicted instructions. Hence, usually useless predictions from a performance standpoint become useful in EOLE. Figure 4.4 shows the proportion of committed instructions that can be late-executed using a baseline 6-issue processor with a D-VTAGE predictor (respectively described in Tables 3.2 and 3.1 in Section 3.6).

Late Execution needs to implement *commit width* ALUs and the associated read ports in the PRF. If an instruction I_1 to be late-executed depends on the result of instruction I_0 of the same commit group, and I_0 will also be late-executed, it does not need to wait as it can use the predicted result of I_0. In other words, all non-executed instructions reaching the Late Execution stage have all their operands ready by construction, as in DIVA [Aus99]. Due to the need to late-execute some instructions as well as validate predictions (including reading results to train the value predictor), at least one extra pipeline stage after Writeback is likely to be required in EOLE. In the remainder of this Chapter, we refer to this stage as the Late Execution/Validation and Training (LE/VT) stage.

As a result, the hardware needed for LE is fairly simple, as suggested by the high-level view of a 2-wide LE/VT Block shown in Figure 4.5. It does not even require a bypass network. In further experiments, we consider that LE and prediction validation can be done in the same cycle, before the Commit stage. EOLE is therefore only one cycle longer than the baseline superscalar it is compared to. While this may be optimistic due to the need to read from the PRF, this only impacts the value misprediction penalty and the pipeline fill delay. In particular, since low confidence branches are resolved in the same cycle as for the baseline, the average branch misprediction penalty will remain very similar.

[6] Predictions whose confidence counter is saturated [Sez11c].

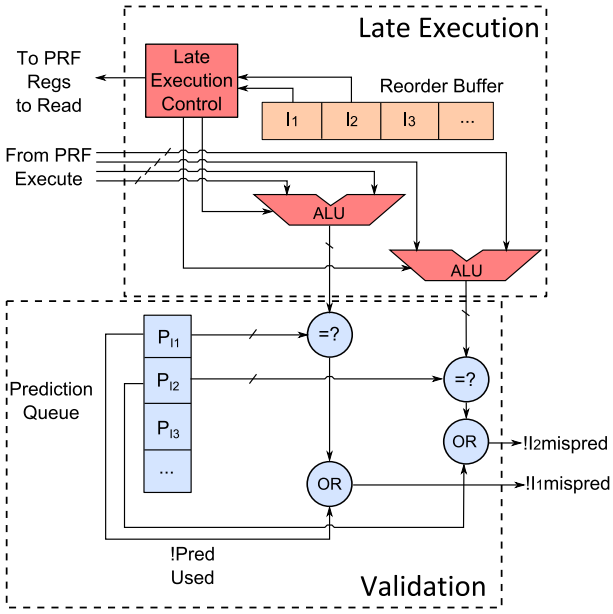


Figure 4.5: Late Execution Block for a 2-wide processor. The top part can late-execute two instructions while the bottom part validates two results against their respective predictions. Buses are *register-width*-bit wide.

Lastly, as a first step, we also consider that enough ALUs are implemented (i.e., as many as the commit width). As a second step, we shall consider reduced-width Late Execution.

4.3.4 Potential OoO Engine Offload

We obtain the ratio of retired instructions that can be offloaded from the OoO engine for each benchmark by summing the columns in Figure 4.3 and 4.4 (both sets are disjoint as we only count late executable instructions that cannot also be early-executed). We also add the *load immediate* instructions that were not considered in Figure 4.3 but that will be early-executed by construction. Those numbers are reported in Figure 4.6. We distinguish four types of instructions:

1. Early-executed because it is a *load immediate*, i.e., its operand is necessarily ready at Rename. In a regular Value Prediction pipeline, these instructions can also bypass the out-of-order engine, as mentioned in 3.6.2.1.

2. Early-executed because it is a ready single-cycle instruction.

3. Late-executed because it is a high-confidence branch, as predicted by the TAGE branch predictor described in 3.6.

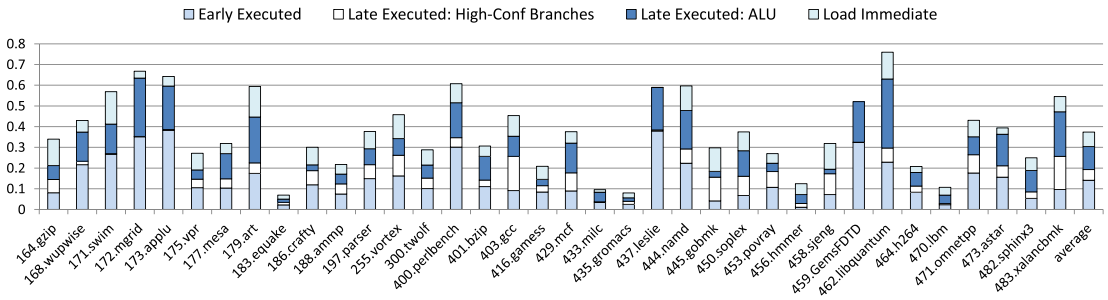


Figure 4.6: Overall portion of committed instructions that can be bypassed from the out-of-order engine, including early-executed, late-executed instructions and *Load Immediate* instructions that can be done for free thanks to the PRF write ports provisioned for VP.

4. Late-executed because it is a value prediction single-cycle instruction.

The ratio of instructions bypassing the OoO engine is very dependent on the application, ranging from around 10% for *equake, milc, gromacs, hmmer* and *lbm* to more than 60% for *mgrid, applu, art, perlbench, leslie* and *namd* and up to 75% for *libquantum*. Nonetheless, it generally represents a significant portion of the retired instructions.

4.4 Evaluation Methodology

4.4.1 Simulator

We reuse the simulator configuration presented in the previous Chapter (Table 3.2). That is, we consider a relatively aggressive 4GHz, 6-issue superscalar baseline with a fetch-to-commit latency of 19 cycles. Since we focus on the OoO engine complexity, both in-order front-end and in-order back-end are overdimensioned to treat up to 8 μ-ops per cycle. We model a deep front-end (15 cycles) coupled to a shallow back-end (4 cycles) to obtain realistic branch/value misprediction penalties. The OoO scheduler features a unified centralized 60-entry IQ and a 192-entry ROB on par with Haswell's, the latest commercially available Intel microarchitecture. We refer to this baseline as the *Baseline_6_60* configuration (6-issue, 60-entry IQ).

The only difference with the simulator configuration presented in Table 3.2 is that in the case where Value Prediction is used, we add a pre-commit stage responsible for validation/training and Late Execution when relevant: the LE/VT stage. This accounts for an additional pipeline cycle (20 cycles) and an increased value misprediction penalty (21 cycles min.). Minimum branch misprediction

latency remains unchanged except for mispredicted very high confidence branches when Late Execution is used. Note that the value predictor is effectively trained after commit, but the value is read from the PRF in the LE/VT stage.

4.4.1.1 Value Predictor Operation

As in the previous Chapter, the predictor makes a prediction at fetch time for every eligible μ-op (i.e. producing a 64-bit or less register that can be read by a subsequent μ-op, as defined by the ISA implementation). To index the predictor, we XOR the PC of the x86_64 instruction left-shifted by two with the μ-op number inside the x86_64 instruction. This avoids all μ-ops mapping to the same entry. We assume that the predictor can deliver as many predictions as requested by the Fetch stage.

In the previous Chapter, we assumed that a prediction is written into the PRF and replaced by its non-speculative counterpart when it is computed in the OoO engine [PS14b]. In parallel, predictions are put in a FIFO queue to be able to validate them – in-order – at commit time. In EOLE, we also use a queue for validation. However, instead of directly writing predictions to the PRF, we place predictions in the Early Execution units, which will in turn write the predictions to the PRF at Dispatch. By doing so, we can use predictions as operands in the EE units.

Finally, since we focus on single core, the impact of VP on memory consistency [MSC+01] in EOLE is left for future work. However, VP should not alter the behavior of processors implementing *Sequential Consistency* [Lam79] or *Total Store Ordering* [SSO+10] (which is used in x86).

Predictor Considered in this Study In this study, we focus on the tightly-coupled hybrid of VTAGE and a simple stride-based Stride predictor, D-VTAGE [PS15a]. For confidence estimation, we use Forward Probabilistic Counters [PS14b], that were described in the previous Chapter. In particular, we use 3-bit confidence counters whose forward transitions are controlled by the vector $v = \{1, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}\}$ as we found it to perform best with D-VTAGE. We use the same configuration as in the previous Chapter (see Table 3.1): a 8K-entry base predictor and 6 1K-entry partially tagged components. We do not try to optimize the size of the predictor (by using partial strides, for instance), but we will see in Chapter 5 that even 16 to 32KB of storage are sufficient to get good performance with D-VTAGE.

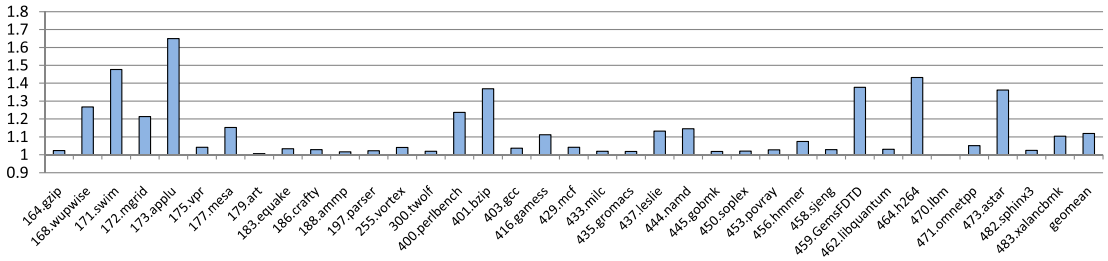


Figure 4.7: Speedup over *Baseline_6_60* brought by Value Prediction using D-VTAGE.

4.4.2 Benchmark Suite

We reuse the benchmarks used to evaluate the contribution of Chapter 3. Specifically, we use 18 integer benchmarks and 18 floating-point programs. Table 3.3 summarizes the benchmarks we use as well as their input, which are part of the *reference* inputs provided in the SPEC software packages. To get relevant numbers, we identify a region of interest in the benchmark using Simpoint 3.2 [PHC03]. We simulate the resulting slice in two steps: First, warm up all structures (caches, branch predictor and value predictor) for 50M instructions, then collect statistics for 100M instructions.

4.5 Experimental Results

In our experiments, we first use *Baseline_6_60* as the baseline to gauge the impact of adding a value predictor only. Then, in all subsequent experiments, we use said baseline augmented with the D-VTAGE predictor detailed in Table 3.1. We refer to it as the *Baseline_VP_6_60* configuration. Our objective is to characterize the potential of EOLE at decreasing the complexity of the OoO engine. We assume that Early and Late Execution stages are able to treat any group of up to 8 consecutive μ-ops every cycle. In Section 4.6, we will consider tradeoffs to enable realistic implementations.

4.5.1 Performance of Value Prediction

Figure 4.7 illustrates the performance benefit of augmenting the baseline processor with the D-VTAGE predictor. A few benchmarks present interesting potential e.g., *wupwise, swim, mgrid, applu, perlbench, bzip, gemsFDTD, h264* and *astar*, some a more moderate potential e.g., *mesa, gamess, leslie, namd, hmmer* and *xalancbmk*, and a few others low potential. No slowdown is observed.

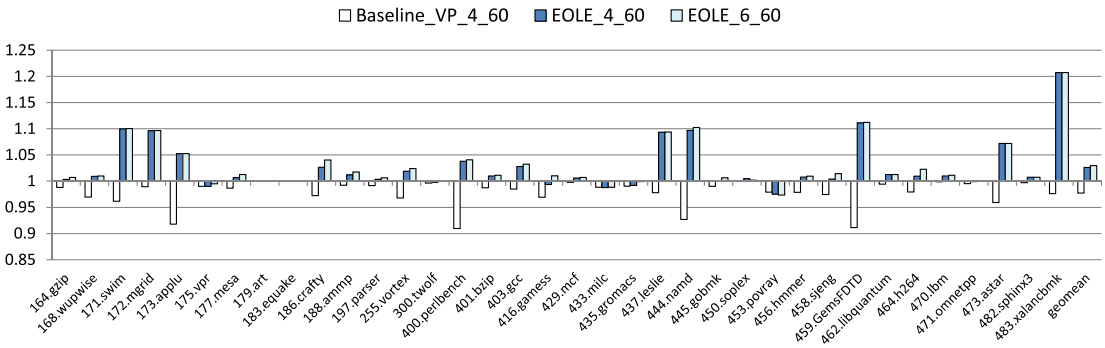


Figure 4.8: Performance of EOLE and the baseline with regard to issue width, normalized to *Baseline_ VP_ 6_ 60*.

In further experiments, figures will depict speedups over the baseline featuring D-VTAGE (*Baseline_ VP_ 6_ 60*).

4.5.2 Impact of Reducing the Issue Width

Applying EOLE without modifying the OoO core (*EOLE_ 6_ 60*) is illustrated in Figure 4.8. In this figure, we also illustrate the baseline, but with a 4-issue OoO engine (*Baseline_ VP_ 4_ 60*) and EOLE using a 4-issue OoO engine (*EOLE_ 4_ 60*).

By itself, EOLE slightly increases performance over the baseline, with a few benchmarks achieving 5% speedup or higher (20% in *xalancmbk*). These improvements come from the fact that we actually increase the number of instructions that can be executed each cycle through EOLE.

Shrinking the issue width to 4 reduces the performance of the baseline by a significant factor in many applications e.g., *applu, perlbench, namd* and *gemsFDTD* for which slowdown is worse than 5%. For EOLE, such a shrink only reduces performance slightly compared with *EOLE_ 6_ 60*. Furthermore, *EOLE_ 4_ 60* still performs slightly higher than *Baseline_ VP_ 6_ 60* in several benchmarks e.g., *swim, mgrid, applu, crafty, vortex, perlbench, gcc, leslie, namd, gemsFDTD, astar* and *xalancbmk*. Only a few slight slowdowns can be observed (e.g., *vpr, milc, gromacs*) and *povray*. Fortunately, even the worst slowdown (2.7% in *povray*) is acceptable.

As a result, EOLE can be considered as a means to reduce issue width without significantly impacting performance on a processor featuring VP.

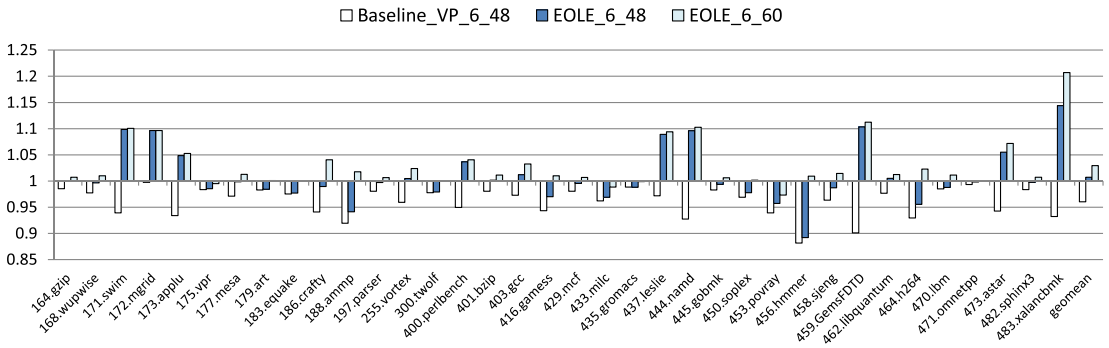


Figure 4.9: Performance of EOLE and the baseline with regard to the number of entries in the IQ, normalized to *Baseline_ VP_ 6_ 60*.

4.5.3 Impact of Shrinking the Instruction Queue

In Figure 4.9, we illustrate the respective performance of the baseline and EOLE when shrinking the instruction queue size from 60 to 48 entries.

Many benchmarks are quite sensitive to such a shrink for *Baseline_ VP_ 6_ 48* e.g., *swim, applu, crafty, ammp, vortex, perlbench, gcc, namd, hmmer, gemsFDTD, h264, astar* and *xalancbmk*. On the other hand, *EOLE_ 6_ 48* does not always exhibit the same behavior. Most applications encounter only minor losses with *EOLE_ 6_ 48* and higher losses with *Baseline_ VP_ 6_ 48*, e.g., *applu* with 5% speedup against 6.4% slowdown, or *namd* with around 9.6% speedup against 7.3% slowdown.

In practice, the benefit of EOLE is greatly influenced by the proportion of instructions that are not sent to the OoO engine. For instance *namd* needs a 60-entry IQ in the baseline case, but since it is an application for which many instructions are predicted or early-executed, it can deal with a smaller IQ in EOLE.

On the other hand, *hmmer*, the application that suffers the most from reducing the instruction queue size with EOLE, exhibits a relatively low coverage of predicted or early-executed instructions. Nonetheless, with *EOLE_ 6_ 48*, slowdown is limited to 3% at most for all but four benchmark, *ammp, povray, hmmer* and *h264*, for which slowdown is still around or worse than 5% (10.8% in *hmmer*).

4.5.4 Summary

EOLE provides opportunities for either slightly improving the performance over a VP-augmented processor without increasing the complexity of the OoO engine, or reaching the same level of performance with a significantly reduced OoO engine complexity. In the latter case, reducing the issue width is our favored direction as it addresses scheduler complexity, PRF complexity and bypass complexity. EOLE

also mitigates the performance loss associated with a reduction of the instruction queue size, but is not able to completely hide it.

In the next Section, we provide directions to limit the global hardware complexity and power consumption induced by the EOLE design and the overall integration of VP in a superscalar processor.

4.6 Hardware Complexity

We have shown that provided that the processor already implements Value Prediction, adopting the EOLE design may allow to use a reduced-issue OoO engine without impairing performance. On the other hand, extra complexity and power consumption are added in the Early Execution engine as well as the Late Execution engine.

In this Section, we first describe the potential hardware simplifications on the OoO engine enabled by EOLE. Then, we describe the extra hardware cost associated with the Early Execution and Late Execution engines. Finally, we provide directions to mitigate this extra cost. Note however that a precise evaluation would require a complete processor design and is beyond the scope of this work.

4.6.1 Shrinking the Out-of-Order Engine

4.6.1.1 Out-of-Order Scheduler

Our experiments have shown that with EOLE, the OoO issue width can be reduced from 6 to 4 without significant performance loss on our benchmark set. This would greatly impact Wakeup since the complexity of each IQ entry would be lower. Similarly, a narrower issue width mechanically simplifies Select. As such, both steps of the Wakeup & Select critical loop could be made faster and/or less power hungry.

Providing a way to reduce issue width with no impact on performance is also crucial because modern schedulers must support complex features such as speculative scheduling and thus selective replay to recover from scheduling mispredictions [PSM+15, MBS07, MS01, MSBU00, SGC01b, MLO01, KL04].

Lastly, to our knowledge, most scheduler optimizations proposed in the literature can be added on top of EOLE. This includes the *Sequential Wakeup* of Kim and Lipasti [KL03] or the *Tag Elimination* of Ernst and Austin [EA02]. As a result, power consumption and cycle time could be further decreased.

4.6.1.2 Functional Units & Bypass Network

As the number of cycles required to read a register from the PRF increases, the bypass network becomes more crucial. It allows instructions to "catch" their operands as they are produced and thus execute back-to-back. However, a full bypass network is very expensive, especially as the issue width – hence the number of functional units – increases. Ahuja et al. showed that partial bypassing could greatly impede performance, even for a simple in-order single-issue pipeline [ACR95]. Consequently, in the context of a wide-issue OoO superscalar with a multi-cycle register read, missing bypass paths may cripple performance even more.

EOLE allows to reduce the issue width in the OoO engine. Therefore, it reduces the design complexity of a full bypass by reducing the number of ALUs and thus the number of simultaneous writers on the network.

4.6.1.3 A Limited Number of Register File Ports dedicated to the OoO Engine

Through reducing the issue width of the OoO engine, EOLE mechanically reduces the number of read and write ports required on the PRF for regular OoO execution. Indeed, assuming 2 read ports and 1 write port per potentially issued instruction each cycle, reducing the issue width from 6 to 4 saves 4 read ports and 2 write ports.

4.6.2 Extra Hardware Complexity Associated with EOLE

4.6.2.1 Cost of the Late Execution Block

The extra hardware complexity associated with Late Execution consists of three main components. First, for validation at commit time, a prediction queue (FIFO) is required to store predicted results. This component is needed anyway as soon as VP associated with validation at commit time is implemented. Second, ALUs are needed for Late Execution. Last, the operands for the late-executed instructions must be read from the PRF. Similarly, the result of VP-eligible instructions must be read from the PRF for validation (predicted instructions only) and predictor training (all VP-eligible instructions).

In the simulations presented in Section 4.5, we have assumed that up to 8 μ-ops (i.e., commit width) could be late-executed per cycle. This would necessitate 8 ALUs and up to 16 read ports on the PRF (including ports required for validation and predictor training).

4.6.2.2 Cost of the Early Execution Block

A single stage of simple ALUs is sufficient to capture most of the potential benefits of Early Execution. The main hardware cost associated with Early Execution is this stage of ALUs and the associated full bypass. Additionally, the predicted results must be written on the register file.

Therefore, in our case, a complete 8-wide Early Execution stage necessitates 8 ALUs, a full 8-to-8 bypass network and 8 write ports on the PRF.

4.6.2.3 The Physical Register File

From the above analysis, an EOLE-enhanced core featuring a 4-issue OoO engine (*EOLE_4_60*) would have to implement a PRF with a total of 12 write ports (respectively 8 for Early Execution and 4 for OoO execution) and 24 read ports (respectively 8 for OoO execution and 16 for Late Execution, validation and training).

The area cost of a register file is approximately proportional to $(R+W)*(R+2W)$, R and W respectively being the number of read and write ports [ZK98]. That is, at equal number of registers, the area cost of the EOLE PRF would be 4 times the initial area cost of the 6-issue baseline (*Baseline_6_60*) PRF. Moreover, this would also translate in largely increased power consumption and longer access time, thus impairing cycle time and/or lengthening the register file access pipeline.

Without any optimization, *Baseline_VP_6_60* would necessitate 14 write ports (respectively 8 to write predictions and 6 for the OoO engine) and 20 read ports (respectively 8 for validation/training and 12 for the OoO engine), i.e., slightly less than *EOLE_4_60*. In both cases, this overhead might be considered as prohibitive in terms of silicon area, power consumption and access time.

However, simple solutions can be devised to reduce the overall cost of the PRF and the global hardware cost of Early/Late Execution without significantly impacting global performance. These solutions apply for EOLE as well as for a baseline implementation of VP. We describe said solutions below.

4.6.3 Mitigating the Hardware Cost of Early/Late Execution

4.6.3.1 Mitigating Early-Execution Hardware Cost

Because Early Executed instructions are treated in-order and are therefore consecutive, one can use a banked PRF and force the allocation of physical registers for the same dispatch group to different register banks. For instance, considering a 4-bank PRF, out of a group of 8 consecutive μ-ops, 2 could be allocated to each

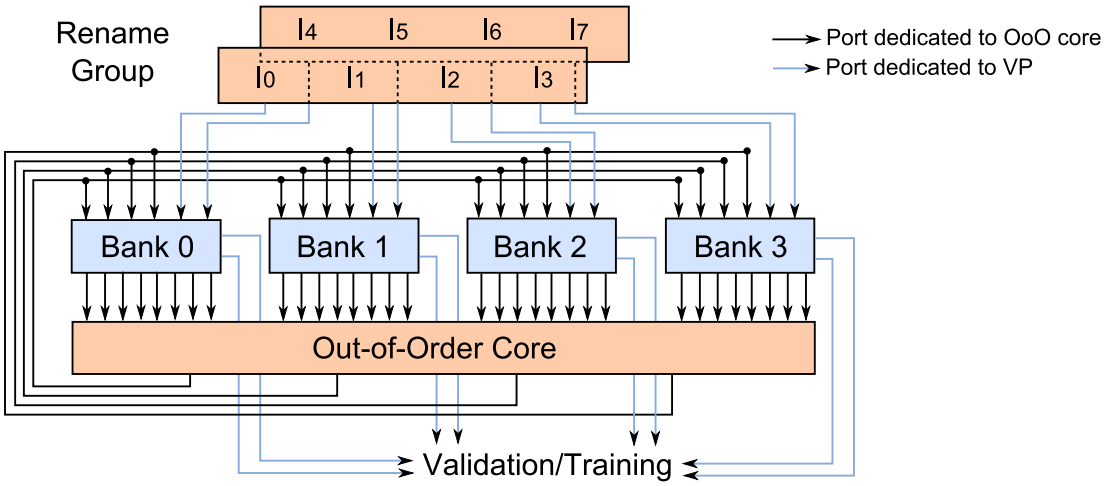


Figure 4.10: Organization of a 4-bank PRF supporting 8-wide VP/Early Execution and 4-wide OoO issue. The additional read ports per-bank required for Late Execution are not shown for clarity.

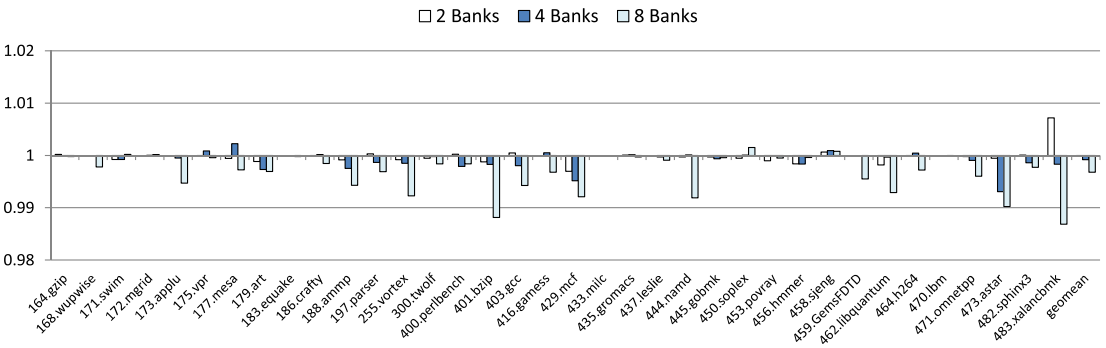


Figure 4.11: Performance of *EOLE_4_60* using a different number of banks in the PRF, normalized to *EOLE_4_60* with a single bank.

bank. A dispatch group of 8 consecutive μ-ops would at most write 2 registers in a single bank after Early Execution. Thus, Early Execution would necessitate only two extra write ports on each PRF bank, as illustrated in Figure 4.10 for an 8-wide Rename/Early Execute, 4-issue OoO core. Interestingly, this would add-up to the number of write ports required by a baseline 6-issue OoO Core.

In Figure 4.11, we illustrate simulation results with a banked PRF. In particular, registers from distinct banks are allocated to consecutive μ-ops and Rename is stalled if the current bank does not have any free register. We consider respectively 2 banks of 128 registers, 4 banks of 64 registers and 8 banks of 32 registers.[7] We observe that the performance loss associated with the unbalanced PRF load is quite limited for our benchmark set, and using 4 banks of 64 registers instead of a single bank of 256 registers appears as a reasonable tradeoff.

Note that register file banking is also a solution for a practical implementation of a core featuring Value Prediction without EOLE, although in that latter case ports are required **in addition** to those required by the baseline pipeline, while we will see that EOLE allows to implement VP with the **same** amount of PRF ports as the baseline OoO pipeline.

4.6.3.2 Narrow Late Execution and Port Sharing

Not all instructions are predicted or late-executable (i.e., predicted and simple ALU or high confidence branches). Moreover, entire groups of 8 μ-ops are rarely ready to commit. Therefore, one can limit the number of potentially late-executed instructions and/or predicted instructions per cycle. For instance, the maximum commit width can be kept to 8 with the extra constraint of using only 6 or 8 PRF read ports for Late Execution and validation/training.

Moreover, one can also leverage the register file banking proposed above to limit the number of read ports on each individual register file bank at Late Execution/Validation and Training. To only validate the prediction for 8 μ-ops and train the predictor, and assuming a 4-bank PRF, 2 read ports per bank would be sufficient. However, not all instructions need validation/training (e.g., branches and stores). Hence, some read ports may be available for LE, although extra read ports might be necessary to ensure smooth LE.

Our experiments showed that limiting the number of LE/VT read ports on each register file bank to 4 results in a marginal performance loss. Figure 4.12 illustrates the performance of *EOLE_4_60* with a 4-bank PRF and respectively 2, 3 and 4 ports provisioned for the LE/VT stage (per bank). As expected, having only two additional read ports per bank is not always sufficient. Having 4 additional read ports per bank, however, yields an IPC very similar to that of

[7] 8 banks were simulated. However, there could be rare situations where the whole set of architectural registers would be allocated to a single bank, leading to major functionality issues.

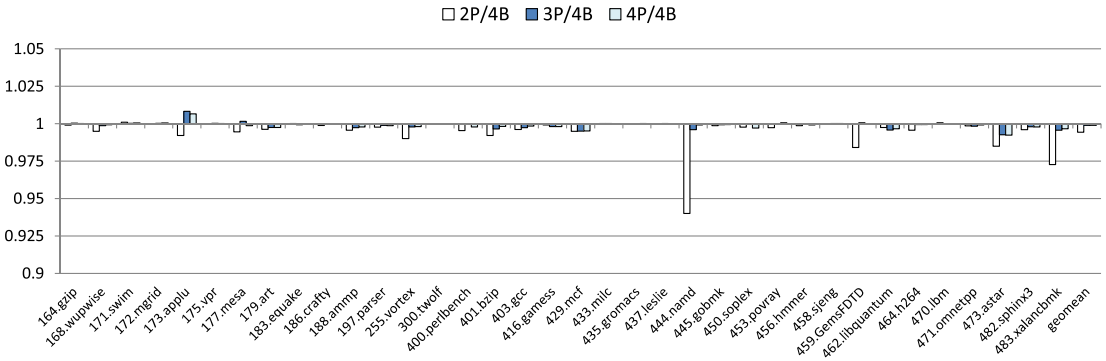


Figure 4.12: Performance of *EOLE_4_60* (4-bank PRF) when the number of read ports dedicated to Late Execution and validation/training is limited, normalized to *EOLE_4_60* (1-bank PRF) with enough ports for full width LE/validation.

EOLE_4_60. Interestingly, adding 4 read ports adds up to a total of 12 read ports per bank (8 for OoO execution and 4 for LE/VT), that is, the same amount of read ports as the baseline 6-issue configuration. Note that provisioning only 3 read ports per bank is also a possibility in that case, but a slightly higher slowdown was observed in [PS14a] when using 3 ports instead of 4. Since this past study used VTAGE and not D-VTAGE, this suggests that the exact amount of ports required may be contingent on the performance of the value predictor.

Regardless, it should be emphasized that the logic needed to select the group of μ-ops to be Late Executed/Validated on each cycle does not not require complex control and is not on the critical path of the processor. This could be implemented either by an extra pipeline cycle or speculatively after dispatch.

4.6.3.3 The Overall Complexity of the Register File

On *EOLE_4_60*, the register file banking proposed above leads to equivalent performance as a non-constrained register file. However, the 4-bank file has only 2 extra write ports per bank for Early Execution and prediction writing and 4 extra read ports for Late Execution/Validation/Training. That is a total of 12 read ports (8 for the OoO engine and 4 for LE/VT) and 6 write ports (4 for the OoO engine and 2 for EE/Prediction), just as the baseline 6-issue configuration without VP.

As a result, if the additional complexity induced on the PRF by VP is noticeable (as issue-width must remain 6), EOLE allows to virtually nullify this complexity by diminishing the number of ports required by the OoO engine. The only remaining difficulty comes from banking the PRF. Nonetheless, according to the previously mentioned area cost formula [ZK98], the total area and power

consumption of the PRF of a 4-issue EOLE core is similar to that of a baseline 6-issue core.

It should also be mentioned that the EOLE structure naturally leads to a distributed register file organization with one file servicing reads from the OoO engine and the other servicing reads from the LE/VT stage. The PRF could be naturally built with a 4-bank, 6 write/8 read ports file (or two copies of a 6 write/4 read ports) and a 4-bank, 6 write/4 read ports one. As a result, the register file in the OoO engine would be less likely to become a temperature hotspot than in a conventional design.

4.6.3.4 Further Possible Hardware Optimizations

It might be possible to further limit the number of effective write ports on the PRF required by Early-Execution and Value Prediction as many μ-ops are not predicted or early-executed. Hence, they do not generate any writes on the PRF and one could therefore limit the number of μ-ops that write to the PRF at the exit of the EE stage. The number of ALUs in said EE stage could also be limited. Specifically, μ-ops and their predicted/computed results could be buffered after the Early Execution/Rename stage. Dispatch groups of up to 8 μ-ops would be built, with the extra constraint of a limited number of at most 4 early-executions or prediction writes on the PRF per dispatch group.

As already mentioned, it is also possible to limit the number of read ports dedicated to LE/VT on each register file bank to 3 with only marginal performance loss, on our benchmark set.

4.6.4 Summary

Apart from the prediction tables and the update logic, the major hardware overhead associated with implementing VP and validation at commit time comes from the extra read and write ports on the register file [PS14b]. We have shown above that EOLE allows to get rid of this overhead on the PRF as long as enough banks can be implemented.

Specifically, EOLE allows to use a 4-issue OoO engine instead of a 6-issue engine. This implies a much smaller instruction scheduler, a much simpler bypass network and a reduced number of PRF read and write ports in the OoO engine. As a result, one can expect many advantages in the design of the OoO execution core: Significant silicon area savings, significant power savings in the scheduler and the register file and savings on the access time of the register file. Power consumption savings are crucial since the scheduler has been shown to consume almost 20% of the power of a modern superscalar core [EA02], and is often a temperature hotspot in modern designs. As such, even if global power savings

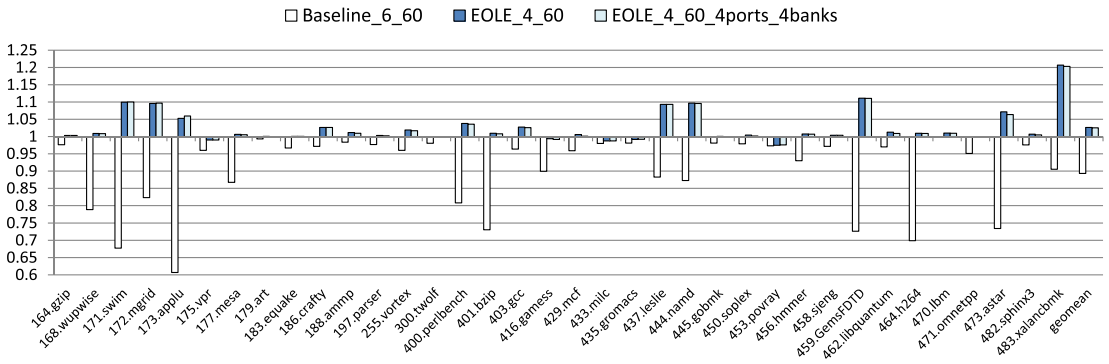


Figure 4.13: Performance of *EOLE_4_60* using 4 ports for Late Execution and validation/training and having 4 64-register banks, *EOLE_4_60* with 16 ports for LE/validation and a single bank and *Baseline_6_60*, normalized to *Baseline_VP_6_60*.

were not to be achieved due to the extra hardware required in EOLE, the power consumption is likely to be more distributed across the core.

On the other hand, EOLE requires some extra but relatively simple hardware for Early/Late Execution. Apart from some relatively simple control logic, this extra hardware consists of a set of ALUs and a bypass network in the Early Execution stage and a set of ALUs in the Late Execution stage. A full rank of ALUs is actually unlikely to be needed. From Figure 4.3, we presume that a rank of 4 ALUs would be sufficient.

Furthermore, implementing EOLE will not impair cycle time. Indeed, Early Execution requires only one stage of simple ALUs and can be done in parallel with Rename. Late Execution and validation may require more than one additional pipeline stage compared to a conventional superscalar processor, but this should have a fairly small impact since low-confidence branch resolution is not delayed. In fact, since EOLE simplifies the OoO engine, it is possible that the core could actually be clocked higher, yielding even more sequential performance.

Therefore, our claim is that EOLE makes a clear case for implementing VP on wide-issue superscalar processors. Higher performance is enabled thanks to VP (see Figure 4.13) while EOLE enables a much simpler and far less power hungry OoO engine. The extra hardware blocks required for EOLE are relatively simple: Sets of ALUs in Early Execution and Late Execution stages, and storage tables and update logic for the value predictor itself.

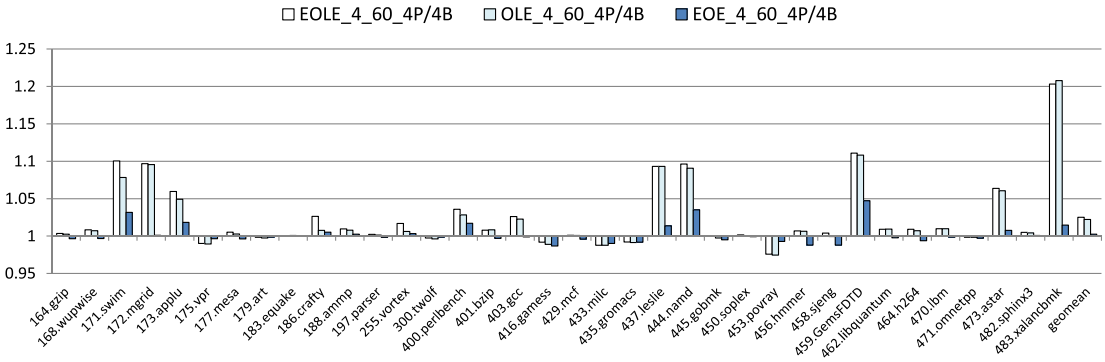


Figure 4.14: Performance of *EOLE_4_60*, *OLE_4_60* and *EOE_4_60* using 4 ports for LE/VT and having 4 64-register banks, normalized to *Baseline_VP_6_60*.

4.6.5 A Note on the Modularity of EOLE: Introducing OLE and EOE

EOLE need not be implemented as a whole. In particular, either Early Execution **or** Late Execution can be implemented, if the performance vs. complexity tradeoff is deemed worthy. Removing Late Execution can further reduce the number of read ports required on the PRF and the need to arbitrate PRF ports for this stage. Removing Early Execution saves on complexity since there is no need for an 8-to-8 bypass network anymore.

Figure 4.14 shows the respective speedups of *EOLE_4_60*, *OLE_4_60* (Late Execution only) and *EOE_4_60* (Early Execution only) over *Baseline_VP_6_60*. As in the previous paragraph, only 4 read ports are dedicated to Late Execution/Validation and Training, and the PRF is 4-banked (64 registers in each bank). The baseline has a single 256-register bank and enough ports to avoid contention.

We observe that some benchmarks are more sensitive to the absence of Late Execution (e.g., *swim, applu, gcc, leslie, namd, gemsFDTD* and *xalancbmk*) while a few are also sensitive to the absence of Early Execution (e.g., *crafty*). As a result, the performance impact of Late Execution appears as more critical in the general case.

Interestingly, slowdown over *Baseline_VP_6_60* remains under 5% in all cases. This suggests that when considering an effective implementation of VP using EOLE, an additional degree of freedom exists as either only Early or Late Execution may be implemented without losing the potential of VP.

4.7 Conclusion

Single thread performance remains the driving force for high-performance design. However, hardware complexity and power consumption remain major obstacles to the implementation of new architectural features.

Value Prediction (VP) is one of such features that has still not been implemented in real-world products due to those obstacles. Fortunately, the mechanisms we have presented in Chapter 3 partially addresses these issues [PS14b]. In particular, prediction validation can be performed at commit time without sacrificing performance. This greatly simplifies design, as the burdens of validation at execution-time and selective replay for VP in the OoO engine are eliminated.

Building on this work, we have proposed EOLE, an {Early | Out-of-Order | Late} Execution microarchitecture aiming at further reducing the hardware complexity and the power consumption of a VP-augmented superscalar processor.

With Early Execution, single-cycle instructions whose operands are immediate or predicted are computed in-order in the front-end and do not have to flow through the OoO engine. With Late Execution, predicted single-cycle instructions as well as very high confidence branches are computed in-order in a pre-commit stage. They also do not flow through the OoO engine. As a result, EOLE significantly reduces the number of instructions dispatched to the OoO engine.

Considering a 6-wide, 60-entry IQ processor augmented with VP and validation at commit time as the baseline, EOLE allows to drastically reduce the overall complexity and power consumption of both the OoO engine and the PRF. EOLE achieves performance very close to the baseline using a 6-issue, 48-entry IQ OoO engine. It achieves similar or higher performance when using a 4-issue, 60-entry IQ engine, with the exception of four benchmarks, *vpr, milc, gromacs* and *povray* (2.7% slowdown at worst).

With EOLE, the overhead over a 6-wide, 60-entry IQ processor (without VP) essentially consists of relatively simple hardware components, the two set of ALUs in the Early and Late Execution, a bypass network and the value predictor tables and update logic. The need for additional ports on the PRF is also substantially lowered by the reduction in issue width and some PRF optimizations (e.g., banking). Lastly, the PRF could be distributed into a copy in the OoO engine and a copy only read by the Late Execution/Validation and Training stage. Consequently, EOLE results in a much less complex and power hungry OoO engine, while generally benefiting from higher performance thanks to Value Prediction. Moreover, we hinted that Late Execution and Early Execution can be implemented separately, with Late Execution appearing as more cost-effective.

Further studies to evaluate the possible variations of EOLE designs may include the full range of hardware complexity mitigation techniques that were discussed in Section 4.6.3 for both Early and Late execution, and the exploration

of other possible sources of Late Execution e.g., indirect jumps, returns, but also store address computations. One can also explore the interactions between EOLE and previous propositions aiming at reducing the complexity of the OoO engine such as the *Multicluster* architecture [FCJV97] or register file-oriented optimizations [WB96]. Finally, there is still a need to look for practical and storage-effective value prediction schemes as well as even more accurate predictors. In the next Chapter, we propose such an infrastructure by relying on D-VTAGE and block-based instruction fetch mechanisms.

Chapter 5

A Realistic, Block-Based Prediction Infrastructure

This Chapter covers a publication[1] in the International Symposium on High Performance Computer Architecture (HPCA 2015) [PS15a].

5.1 Introduction & Motivations

Taking a step back, only two distinct components are responsible for the majority of the additional complexity brought by Value Prediction: 1) the out-of-order execution engine and 2) the intrinsic complexity of the Value Prediction data path.

The first source of complexity was studied in Chapter 3 [PS14b] and Chapter 4 [PS14a]. We showed that **the out-of-order engine does not need to be overhauled to support VP**. First, very high accuracy can be enforced at reasonable cost in coverage and minimal complexity [PS14b]. Thus, both prediction validation and recovery by squashing can be done outside the out-of-order (OoO) engine, at commit time instead of execution time. Moreover, we devised a new pipeline organization, EOLE (depicted in Figure 5.1), that leverages VP with validation at Commit to execute many instructions outside the OoO core, in-order [PS14a]. Those instructions are processed either as-soon-as-possible (*Early Execution* in parallel with Rename) using predicted/immediate operands or as-late-as-possible (*Late Execution* just before Commit) if they are predicted. With EOLE, the issue width can be reduced without sacrificing performance, mechanically decreasing the number of ports on the Physical Register File (PRF). Validation at commit time also allows to leverage PRF banking for prediction and validation, leading to an overall number of ports similar in EOLE with VP and in

[1] D-VTAGE was also introduced in this publication but was covered in Chapter 3.

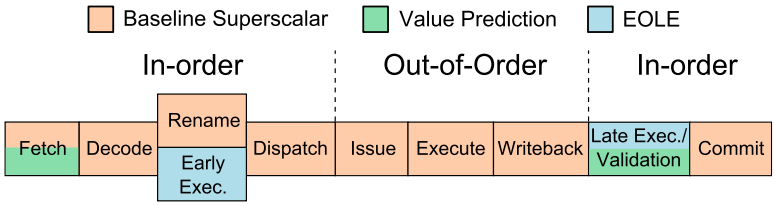


Figure 5.1: Pipeline diagram of EOLE.

the baseline case without VP. That is, EOLE achieves VP-enabled performance while the overall hardware cost of the execution engine is actually *lower* than in a baseline processor without VP.

In this Chapter, we address the remaining source of hardware complexity, which is associated with the value predictor infrastructure itself.

First, to handle the multiple value predictions needed each cycle in a wide issue processor, we propose **B**lock-**B**ased value **P**rediction (BBP or BeBoP). With this scheme, all predictions associated with an instruction fetch block are put in a single predictor entry. The predictor is therefore accessed with the PC of the instruction fetch block and the whole group of predictions is retrieved in a single read. BeBoP accommodates currently implemented instruction fetch mechanisms. However, it only addresses complexity of operation, but not storage requirement.

As a result, in a second step, we propose ways to reduce the storage footprint of the tightly-coupled hybrid predictor presented in Chapter 3, D-VTAGE. This predictor can be made very space-efficient as it can use partial strides (e.g., 8- and 16-bit). As a result, its storage cost is amenable to that of the I-Cache or the branch predictor.

Third, we devise a cost-effective checkpoint-based implementation of the speculative last-value window that was previously discussed in 2.2. Such a window is required due to the presence of a stride-based prediction scheme in D-VTAGE. Without it, instructions in loops of which there can be several instances in the pipeline are unlikely to be correctly predicted.

5.2 Block-Based Value Prediction

5.2.1 Issues on Concurrent Multiple Value Predictions

Modern superscalar processors are able to fetch/decode several instructions per cycle. Ideally, the value predictor should be able to predict a value for all instructions fetched during that cycle.

For predictors that do not use any local value history to read tables (e.g., the

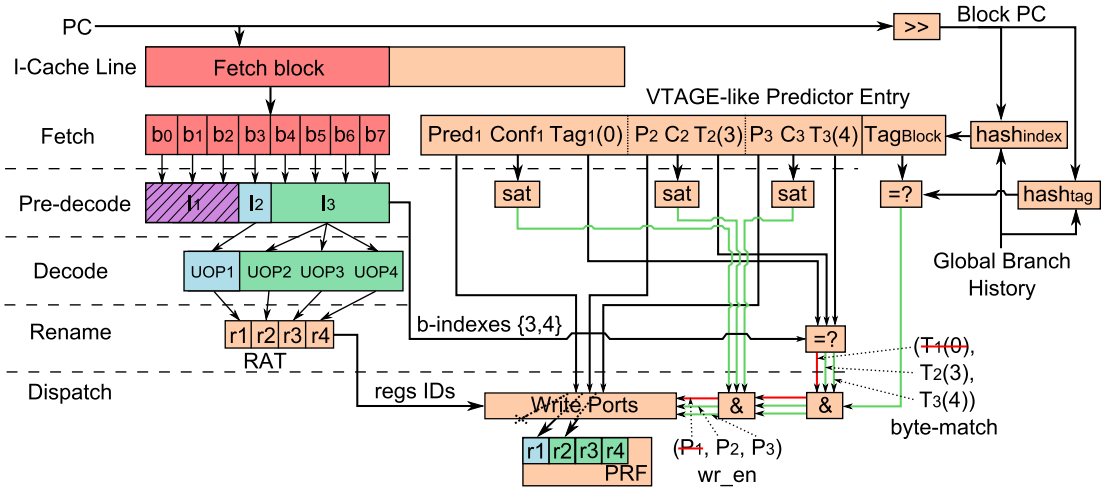


Figure 5.2: Prediction attribution with BeBoP and 8-byte fetch blocks. The predictor is accessed with the currently fetched PC and predictions are attributed using byte indexes as tags.

Last Value Predictor, the Stride predictor or VTAGE), and when using a RISC ISA producing at most one result per instruction (e.g., Alpha), a straightforward implementation of the value predictor consists in mimicking the organization of the instruction fetch hardware in the different components of predictor. Predictions for contiguous instructions are stored in contiguous locations in the predictor components and the same interleaving structures are used in the instruction cache, the branch predictor and the value predictor. However, for predictors using local value history (e.g., FCM [SS97a]), even in this very favorable scenario, some predictor components must be bank-interleaved at the instruction level and the banks individually accessed with different indexes.

Moreover, the most popular ISAs do not have the regularity property of Alpha. For x86 – that we consider in this study – some instructions may produce several results, and information such as the effective PC of instructions in the fetch block[2] and the number of produced values are known after several cycles (after pre-decoding at Fetch and after decoding, respectively). That is, **there is no natural way to associate a value predictor entry with a precise PC**: Smooth multiple-value prediction on a variable-length ISA remains a challenge in itself.

[2]Storing boundary bits in the I-cache can remedy this, but at high storage cost [AMD98].

5.2.2 Block-based Value-Predictor accesses

We remedy this fundamental issue by proposing Block-Based Value Prediction (BeBoP). To introduce this new access scheme and layout, we consider a VTAGE-like predictor, but note that block-based prediction can be generalized to most predictors.

The idea is to access the predictor using the *fetch-block PC* (i.e., the current PC right-shifted by $log_2(fetchBlockSize)$), as well as some extra global information as defined in VTAGE [PS14b]. Instead of containing a single value, the entry that is accessed now consists in N_{pred} values ($N_{pred} > 0$). The m^{th} value in the predictor entry is associated with the m^{th} result in the fetch block, and not with its precise PC.

The coupling of the Fetch/Decode pipeline and the value predictor is illustrated in more details in Figure 5.2, assuming an x86-like ISA and a VTAGE-like value predictor. Fetch first retrieves a chunk of n bytes from the I-cache, then pre-decodes it to determine instruction boundaries, to finally put it in a decode queue. This queue feeds the x86 decoders in charge of generating μ-ops. At the same time, the value predictor is read using the fetch-block PC and global information to retrieve N_{pred} predictions. After Decode, the predictions are attributed to the μ-ops in the fetch block by matching instruction boundaries with small per-prediction tags. The rationale behind this tagging scheme is highlighted in the next paragraph. Finally, predictions proceeds to the PRF depending on the saturation of the confidence counter associated with each of them.

5.2.2.1 False sharing issues

Grouping predictions introduces the issue of false prediction sharing if one uses the most significant bits of the PC to index the value predictor (e.g., removing the 4 last bits of the address when fetching 16-byte instruction blocks). False sharing arises when an instruction block is fetched with two distinct entry points e.g., instructions I_1 and I_2. In that case, the traces of μ-ops generated for this block at Decode are different, leading to complete confusion in the predicted values.

We avoid this issue by tagging each prediction in the block with the last $log_2(fetchBlockSize)$ bits of the instruction PC to which the prediction was attributed the last time the block was fetched.

That is, to access the predictor, we use the block address, and once N_{pred} predictions have flowed out of the predictor, they are attributed to each μ-op by matching the indexes given by the boundary bits against the per-prediction tags, as described in Figure 5.2. In this particular case, the first instruction of the block is I_2, while predictions correspond to the case where the block was entered through instruction I_1. If no specific action were taken, prediction P_1 would be attributed to *UOP1* of I_2. However, thanks to the use of tags to mask predictions,

prediction P_2 (with tag T_2 equals 3) is attributed to *UOP1* of I_2 (with boundary 3 since I_2 starts at byte 3). That is, P_1 is not shared between I_1 and I_2.

Tags are modified when the predictor is updated with the following constraint: a greater tag never replaces a lesser tag, so that the entry can learn the "real" location of instructions/μ-ops in the block. For instance, if the block was entered through I_2 but has already been entered through I_1 before, then the tag associated with P_1 is the address of the first byte of I_1 (i.e., 0). The address of the first byte of I_2 is greater than that of I_1, so it will not replace the tag associated with P_1 in the predictor, even though dynamically, for that instance of the block, I_2 is the first instruction of the block. As a result, the pairing P_1/I_1 is preserved throughout execution. This constraint does not apply when the entry is allocated.

5.2.2.2 On the Number of Predictions in Each Entry

The number of μ-ops producing a register depends on the instructions in the block, but only the size of the instruction fetch block is known at design-time. Thus, N_{pred} should be chosen as a tradeoff between coverage (i.e., provision enough predictions in an entry for the whole fetch block) and wasted area (i.e., having too many predictions provisioned in the entry). For instance, in Figure 5.2, since I_1 and I_2 both consume a prediction, only *UOP2* of instruction I_3 can be predicted.

In Section 5.6.2, we will study the impact of varying N_{pred} while keeping the size of the predictor constant. Specifically, a too small N_{pred} means that potential is lost while a too big N_{pred} means that space is wasted as not all prediction slots in the predictor entry are used. Additionally, at constant predictor size, a smaller N_{pred} means more entries, hence less aliasing.

5.2.2.3 Multiple Blocks per Cycle

In order to provide high instruction fetch bandwidth, several instruction blocks are fetched in parallel on wide-issue processors. For instance, on the EV8 [SFKS02], two instruction fetch blocks are retrieved each cycle. To support this parallel fetch, the instruction cache has to be either fully dual-ported or bank-interleaved. In the latter case, a single block is fetched on a conflict unless the same block appears twice. Since fully dual-porting is much more area- and power-consuming, bank interleaving is generally preferred.

In BeBoP, the (potential) number of accesses per cycle to the predictor tables is similar to the number of accesses made to the branch predictor tables, i.e., up to 3 accesses per fetch block: Read at fetch time, second read at commit time and write to update. This adds up to 6 accesses per cycle if two instruction blocks are fetched each cycle.

However, for value predictor or branch predictor components that are indexed using only the block PC (e.g., Last Value component of VTAGE and Stride predictors), one can replicate the same bank interleaving as the instruction cache in the general case. If the same block is accessed twice in a single cycle, stride-based predictors must provision specific hardware to compute predictions for both blocks. This can be handled through computing both *(Last Value + stride)* and *(Last Value + stride1 + stride2)* using 3-input adders.

For value or branch predictor components that are indexed using the global branch history or the path history, such as VTAGE tagged components, one can rely on the interleaving scheme that was proposed to implement the branch predictor in the EV8 [SFKS02]. By forcing consecutive accesses to map to different banks, the 4-way banked predictor components are able to provide predictions for any two consecutively fetched blocks with a single read port per bank. Moreover, Seznec states that for the TAGE predictor [Sez11b], one can avoid the second read at update. The same can be envisioned for VTAGE. Lastly, since they are more infrequent than predictions, updates can be performed through cycle stealing. Therefore, tagged components for VTAGE and TAGE can be designed with single ported RAM arrays.

That is, in practice, the global history components of a hybrid predictor using BeBoP such as VTAGE-Stride or D-VTAGE – that was presented in Chapter 3 – can be built with the same bank-interleaving and/or multi-ported structure as a multiple table global history branch predictor such as the EV8 branch predictor or the TAGE predictor. Only the Last Value Table (LVT) of (D-)VTAGE must replicate the I-Cache organization.

5.3 Implementing a Block-based D-VTAGE Predictor

In 3.5.2, we presented D-VTAGE, an efficient, tightly-coupled hybrid value predictor. Its advantages are twofold. First, it is able to predict control-flow dependent patterns, strided patterns and *control-flow dependent strided patterns*. Second, it has been shown that short strides could capture most of the coverage of full strides in a stride-based value predictor [GVDB01]. As such, using D-VTAGE instead of VTAGE, for instance, would allow to greatly reduce storage requirements. Partial strides should be sign-extended when they flow out of the predictor to be added to the base value. This is required to enable the use of negative strides.

Nonetheless, a concrete implementation of D-VTAGE is not straightforward as it should be practical enough but still worth the transistors spent from a performance standpoint. This is true for an instruction-based D-VTAGE predictor, but also for a block-based implementation, as BeBoP uncovers some new challenges

for the architect. In this section, we describe the main implementation-related issues that must be addressed.

5.3.1 Complexity Intrinsic to D-VTAGE

5.3.1.1 Prediction Critical Path

Because of its stride-based prediction mechanism, D-VTAGE possesses a prediction critical path. Specifically, the prediction for a given instance must be made available fast enough so that a subsequent instance can use it as its last value. In other words, bypassing the output of the adder to its input in a single cycle is required to predict back-to-back[3] occurrences [PS14b], as previously shown in Figure 3.1.

While this might prove challenging, it actually appears feasible as 64-bit addition and bypassing are already performed in a single cycle in the out-of-order engine.

In the case where two instances of the same instruction are fetched in the *same* cycle, a 3-input adder can provide the prediction for the second instance.

5.3.1.2 Associative Read

D-VTAGE features several tagged components that are accessed in parallel to find the relevant stride. In nature, this is an associative read, a usually costly operation. However, since the value predictor is not on the critical path because the prediction is only needed at Dispatch,[4] the data does not need to flow out of the components in an associative fashion. Rather, only the tags need to be read at the same time, while the stride can be fetched directly from the correct component, once identified. Moreover, the components are partially tagged, e.g., tags are in the range of 10-15 bits instead of full width, hence comparison energy and latency are lower than for D-Cache tag matching, for instance.

5.3.1.3 Speculative History

Because of its stride-based nature, D-VTAGE relies on the value produced by the most recent instance of an instruction to compute the prediction for the newly fetched instance. As many instances of the same instruction can coexist in the instruction window at the same time, the hardware should provide some support to grab the predicted value for the most recent speculative instance. Such a hardware structure could be envisioned as a chronologically ordered associative buffer whose size is roughly that of the ROB. For instruction-based VP, such a

[3]Fetched in two consecutive cycles, e.g., in tight loops

[4]Rename if Early Execution is implemented [PS14a].

design should prove slow and power hungry, e.g., more than ROB-size entries at worst and 8 parallel associative searches each cycle for the simulation framework we are considering (8-wide, 6-issue superscalar).

Fortunately, BeBoP allows to greatly reduce the number of required entries as well as the number of parallel accesses each cycle. We develop such a design of the speculative window in Section 5.4.

5.3.2 Impact of Block-Based Prediction

5.3.2.1 Predictor Entry Allocation

As for VTAGE and TAGE, the allocation policy in the tagged components of D-VTAGE is driven by the *usefulness* of a prediction [SM06]. The allocation of a new entry also depends on whether the prediction was correct or not. However, we consider a block-based predictor. Therefore, the allocation policy needs to be modified since there can be correct and incorrect predictions in a single entry, and some can be *useful* or not.

In our implementation, a new entry is allocated if at least one prediction in the block is wrong. However, the confidence counter of predictions from the providing entry are propagated to the newly allocated entry. This allows the predictor to be efficiently trained (allocate on a wrong prediction) while preserving coverage since high confidence predictions are duplicated. The *usefulness* bit is kept per block and set if a single prediction of the block is useful as defined in [PS14b], that is if it is correct and the prediction in the *alternate*[5] component is not.

5.3.2.2 Prediction Validation

The EOLE processor model assumes that predictions are used only if they are very high confidence. Therefore, unused predictions must be stored to wait for validation in order to train the predictor. To that extent, we assume a FIFO Update Queue where prediction blocks are pushed at prediction time and popped at validation time. This structure can be read and written in the same cycle, but reads and writes are guaranteed not to conflict by construction. Specifically, this queue should be able to contain all the predictions potentially in flight after the cycle in which predictions become available. It would also be responsible for propagating any information visible at prediction time that might be needed at update time (e.g., predictor entry, confidence, etc.).

In essence, this structure is very similar to the speculative window that should be implemented for D-VTAGE. Indeed, it contains all the inflight predictions for each block, and it should be able to rollback to a correct state on a pipeline flush

[5]Next lower matching component.

(we describe recovery in the context of BeBoP in the next Section). However, to maximize coverage, the FIFO update queue must be large enough so that prediction information is never lost due to a shortage of free entries. On the contrary, we will see that we can implement the speculative window with much less entries than the theoretical number of blocks that can be in flight at any given time. Moreover, the speculative window must be associatively searched every time a prediction block is generated while the FIFO does not need associative lookup (except potentially for rollback). As a result, if both structures essentially store the same data, it is still interesting to implement them as two separate items.

5.4 Block-Based Speculative Window

5.4.1 Leveraging BeBoP to Implement a Fully Associative Structure

In the context of superscalar processors where many instructions can be in flight, computational predictors such as D-VTAGE cannot only rely on their Last Value Table. Indeed, in the case where several instances of a loop body are live in the pipeline, the last value required by the predictor may not have been retired, or even computed. As a result, a speculative LVT is required so that the predictor can keep up with the processor. Given the fact that an instruction is allowed to use its prediction only after several tens of previous instances have been correctly predicted (because of confidence estimation), keeping the predictor synchronized with the pipeline is even more critical. As a result, the third and last contribution of this Chapter deals with the specific design of the *Speculative Window*.

An intuitive solution would consist in using an associative buffer that can store all the in-flight predictions. On each lookup, this buffer would be probed and provide the most recent prediction to be used as the last value, if any. Unfortunately, its size would be that of the ROB plus all instructions potentially in flight between prediction availability (Rename) and Dispatch. Associative structures of this size – such as the Instruction Queue – are known to be slower than their RAM counterparts as well as power hungry [EA02, KL03, PJS97].

Nonetheless, thanks to BeBoP, the number of required entries is actually much smaller than with instruction-based VP, and the number of accesses is at most the number of fetch blocks accessed in a given cycle. Furthermore, although this buffer behaves as a fully associative structure for *reads*, it acts as a simple circular buffer for *writes* because it is chronologically ordered. That is, when the predictor provides a new prediction block, it simply adds it at the head of the buffer, without having to match any tag. If the head overlaps with the tail (i.e., if the buffer was dimensioned with too few entries), both head and tail are

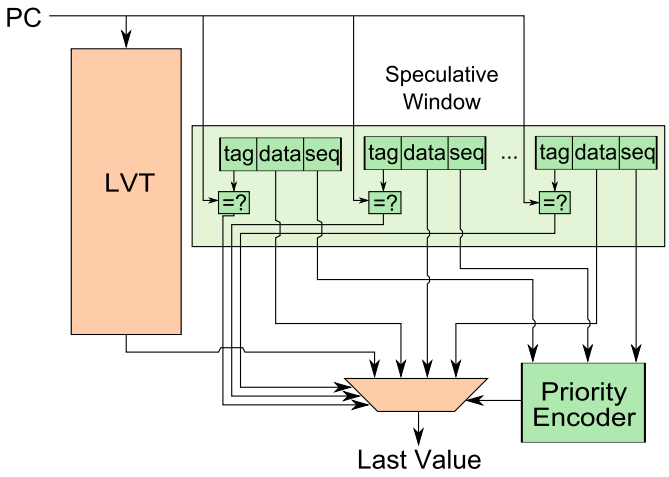


Figure 5.3: N-way speculative window. The priority encoder controls the multiplexer by prioritizing the matching entry corresponding to the most recent sequence number.

advanced. Lastly, partial tags (e.g., 15 bits) can be used to match the blocks, as VP is speculative by nature (false positive is allowed).

To order the buffer and thus ensure that the most recent entry is providing the last values if multiple entries hit, we can simply use internal sequence numbers. In our experiments, we use the sequence number of the first instruction of the block. A block-diagram of our speculative window is shown in Figure 5.3.

5.4.2 Consistency of the Speculative History

Block-based VP entails intrinsic inconsistencies in the speculative window. For instance, if a branch is predicted as taken, some predictions are computed while the instructions corresponding to them have been discarded (the ones in the same block as the branch but located *after* the branch). Therefore the speculative history becomes inconsistent even if the branch is correctly predicted.

Moreover, pipeline squashing events may also cause inconsistencies in the speculative window. The same stands for the FIFO update queue, as it is organized in a similar fashion as the speculative window[6]. To illustrate why, let us denote the first instruction to be fetched after the pipeline flush as I_{new} and the instruction that triggered the flush as I_{flush}. Let us also denote the fetch block address of I_{new} as B_{new} and that of I_{flush} as B_{flush}.

On a pipeline flush, all the entries whose associated sequence number is strictly greater than that of I_{flush} are discarded in both the speculative window and

[6]In our implementation, each entry of the FIFO update queue is also tagged with the internal sequence number of the first instruction in the fetch-block to enable rollback.

the FIFO update queue. The block at the head of both structures therefore corresponds to B_{flush}. We must then consider if whether I_{new} belongs to the same fetch block as I_{flush} or not. If not, the predictor should operate as usual and provide a prediction block for the new block, B_{new}. If it does belong to the same block (i.e. B_{flush} equals B_{new}), several policies are available to us.

Do not Repredict and Reuse (*DnRR*) This policy assumes that all the predictions referring to B_{flush}/B_{new} are still valid after the pipeline flush. That is, I_{new} and all subsequent instructions belonging to B_{flush} will be attributed predictions from the prediction block that was generated when B_{flush}/B_{new} was first fetched. Those predictions are available at the head of the FIFO update queue. With this policy, the heads of the speculative history and FIFO update queue are not discarded.

Do not Repredict and do not Reuse (*DnRDnR*) This policy is similar to *DnRR* except that all newly fetched instruction belonging to B_{flush}/B_{new} will be forbidden to use their respective predictions. The reasoning behind this policy is that the case where B_{new} equals B_{flush} typically happens on a value misprediction, and if a prediction was wrong in the block, chances are that the subsequent ones will be wrong too.

Repredict (*Repred*) This policy squashes the heads of the speculative history and FIFO update queue (the blocks containing I_{flush}). Then, once I_{new} is fetched, a new prediction block for B_{flush}/B_{new} is generated. In general, this provides the same predictions for the block, however, more up-to-date predictions may be generated if the entry has been updated since it was accessed the first time by this block.

Keep Older, Predict Newer (*Ideal*) This idealistic policy is able to keep the predictions pertaining to instructions of block B_{flush} older than I_{flush} while generating new predictions for I_{new} and subsequent instructions belonging to B_{new}. In essence, this policy assumes a speculative window (resp. FIFO update queue) that tracks predictions at the instruction level, rather than the block level. As a result, the speculative window (resp. FIFO update queue) is always consistent.

5.5 Evaluation Methodology

5.5.1 Simulator

We use the framework described in 3.6 with the modifications made to evaluate EOLE detailed in 4.4. Table 3.2 describes our model: A relatively aggressive 4GHz, 6-issue superscalar pipeline. The fetch-to-commit latency is 20 cycles as we are dedicating one stage to validation and late-execution, as in the previous Chapters. The in-order front-end and in-order back-end are overdimensioned to treat up to 8 μ-ops per cycle. We model a deep front-end (15 cycles) coupled to a shallow back-end (5 cycles) to obtain a realistic branch/value misprediction penalty: 20/21 cycles minimum. We allow two 16-byte blocks of instructions to be fetched each cycle, potentially over a single taken branch. The OoO scheduler is dimensioned with a unified centralized 60-entry IQ and a 192-entry ROB on par with the latest commercially available Intel microarchitecture.

In the first experiment, we consider as a baseline a pipeline with Value Prediction without EOLE. The processor is therefore similar to the one described in Table 3.2, except the pipeline latency is 20 cycles instead of 19 cycles, as we still dedicate a stage to validation. Moreover, the register file is 4-banked to model a realistic pipeline with VP without EOLE (only 2 additional write ports and 2 additional read ports per bank). We refer to this model as the *Baseline_VP_6_60* configuration (6-issue, 60-entry IQ).

In the last experiment, we go back to a baseline without Value Prediction at all. In that case, the processor is exactly the one depicted in Table 3.2: the Validation and Late Execution stage is removed, yielding a fetch-to-commit latency of 19 cycles. We refer to this model as the *Baseline_6_60* configuration.

In the remaining experiments, our baseline uses EOLE: we consider a 1-deep, 8-wide Early Execution stage and an 8-wide Late Execution/Validation stage limited to 4 read ports per bank, with a 4-bank register file and a 4-issue out-of-order engine, as described in 4.6.3. We refer to this model as the *EOLE_4_60* configuration.

5.5.1.1 Value Predictor Operation

In its baseline version, the predictor makes a prediction at Fetch for every eligible μ-op (i.e., producing a 64-bit or less register that can be read by a subsequent μ-op, as defined by the ISA implementation). To index the predictor, we XOR the PC of the x86_64 instruction with the μ-op index inside that x86_64 instruction [PS14a]. This avoids all μ-ops mapping to the same entry, but may create aliasing. We assume that the predictor can deliver *fetch-width* predictions each cycle. The predictor is updated in the cycle following retirement.

When using block-based prediction, the predictor is simply accessed with the PC of each fetch block and provides several predictions at once for those blocks. Similarly, update blocks are built after retirement and an entry is updated as soon as an instruction belonging to a block different than the one being built is retired. Update can thus be delayed for several cycles.

We use the D-VTAGE configuration depicted in Table 3.1: 8K-entry base component with 6 1K-entry partially tagged components. The partial tags are 13 bits for the first component, 14 for the second, and so on. Histories range from 2 to 64 bits in a geometric fashion. We also use 3-bit *Forward Probabilistic Counters* with probabilities $v = \{1, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}\}$. Unless specified otherwise, we use 64-bit strides in the predictor. Finally, to maximize accuracy, we tag the LVT with the 5 higher bits of the fetch block PC.

5.5.2 Benchmark Suite

We reuse the benchmarks used to evaluate the contribution of Chapter 3. Specifically, we use 18 integer benchmarks and 18 floating-point programs. Table 3.3 summarizes the benchmarks we use as well as their input, which are part of the *reference* inputs provided in the SPEC software packages. To get relevant numbers, we identify a region of interest in the benchmark using Simpoint 3.2 [PHC03]. We simulate the resulting slice in two steps: First, warm up all structures (caches, branch predictor and value predictor) for 50M instructions, then collect statistics for 100M instructions.

5.6 Experimental Results

In further experiments, when we do not give speedup for each individual benchmark, we represent results using the geometric mean of the speedups on top of the $[Min, Max]$ box plot of the speedups.

5.6.1 Baseline Value Prediction

Figure 5.4 depicts the speedup of a 4-issue, 60-entry IQ, 4-bank PRF, 12-read ports per bank EOLE architecture featuring D-VTAGE over *Baseline_ VP_ 6_ 60*. As expected, very little slowdown is observed by scaling down the issue width from 6 to 4 (at worst 0.982 in *povray*). We refer to this configuration as *EOLE_ 4_ 60* and use it as our baseline in further experiments.

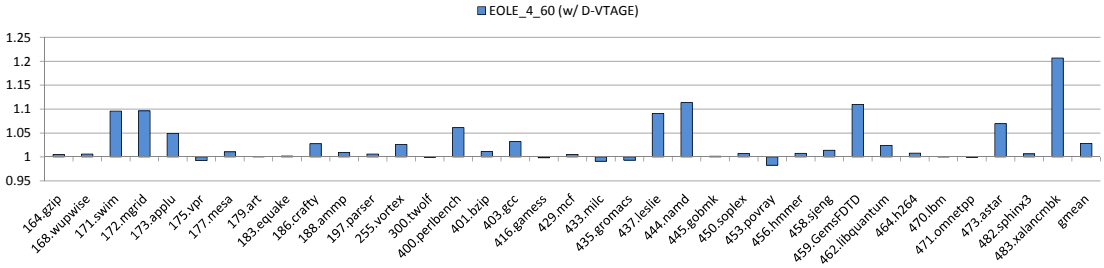
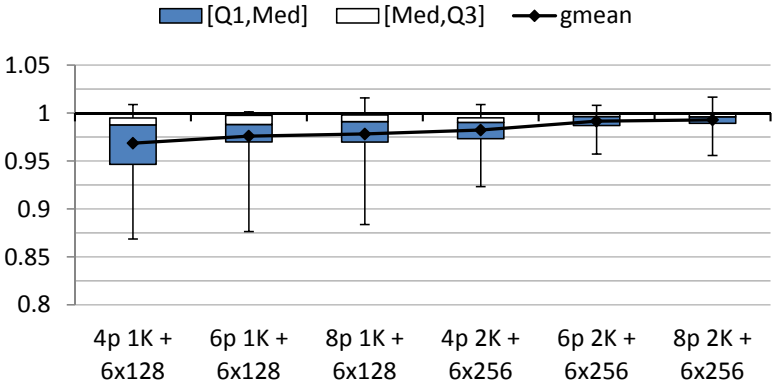
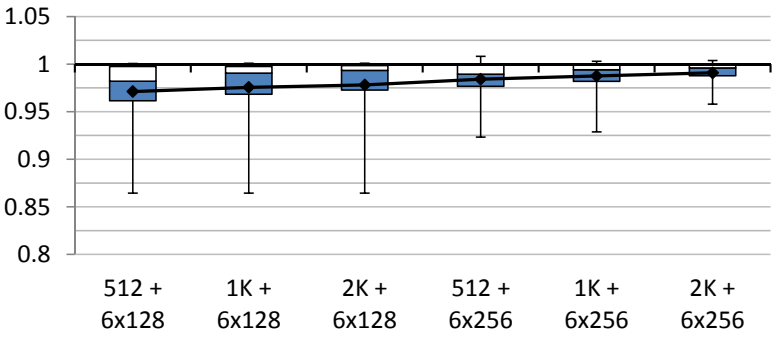


Figure 5.4: Speedup of bank/port constrained *EOLE_4_60* w/ D-VTAGE over *Baseline_VP_6_60*.



(a) Impact of the number of predictions per entry in D-VTAGE.



(b) Impact of the key structures sizes for a 6 predictions per entry D-VTAGE.

Figure 5.5: Performance of D-VTAGE with BeBoP normalized to the performance of *EOLE_4_60*.

5.6.2 Block-Based Value Prediction

5.6.2.1 Varying the Number of Entries and the Number of Predictions per Entry

Figure 5.5 (a) shows the impact of using BeBoP on top of *EOLE_4_60*. We respectively use 4, 6 and 8 predictions per entry in the predictor, while keeping the size of the predictor roughly constant. For each configuration, we study a predictor with either a 2K-entry base predictor and six 256-entry tagged components or a 1K-entry base predictor and six 128-entry tagged components. The *Ideal* policy is used to handle recovery in the infinite speculative window.

We first make the observation that 6 predictions per 16-byte fetch block appear sufficient. Second, we note that reducing the size of the structures plays a key role in performance. For instance, maximum slowdown is 0.876 for *{6p 1K + 6x128}*, but only 0.957 for *{6p 2K + 2x256}*.

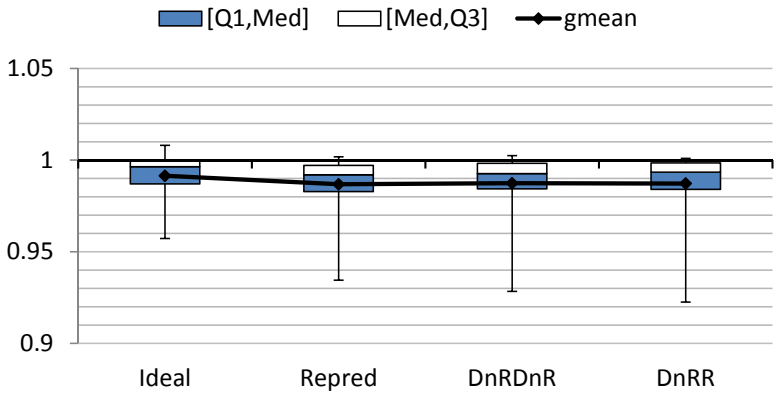
To gain further insight, in Figure 5.5 (b), we focus on a D-VTAGE predictor with 6 predictions per entry and we vary the number of entries in the base component while keeping the number of entries in the tagged components constant, and vice-versa. The results hint that decreasing the sizes of the tagged components from 256 entries to 128 has more impact than reducing the number of entries of the base predictor.

In further experiments, we consider an optimistic predictor configuration with a 2K-entry base predictor and six 256-entry tagged components. Each entry contains six predictions. Such an infrastructure has an average slowdown over *EOLE_4_60* of 0.991, a maximum slowdown of 0.957, and its size is around 290KB assuming 64-bit strides are stored in the predictor.

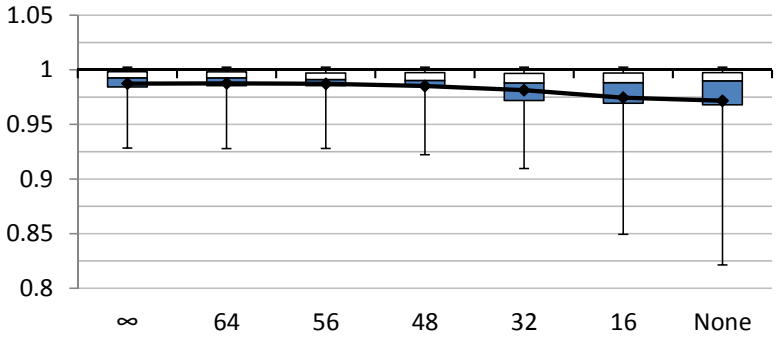
5.6.2.2 Partial Strides

To predict, D-VTAGE adds a constant, the *stride* to the last value produced by a given instruction. In general, said constant is not very large. As most of the entries of D-VTAGE only contain strides and not last values, reducing the size of strides from 64 bits to 16 or 8 bits would provide tremendous savings in storage.

Using the baseline D-VTAGE configuration, but varying the size of the strides used to predict, we found that average speedup (gmean) ranges from 0.991 (64-bit strides) to 0.985 (8-bit strides), while maximum slowdown increases from 0.957 (64-bit) to 0.927 (8-bit). However, results for 16-, 32- and 64-bit strides are very similar. In other words, performance is almost entirely conserved even though the size of the predictor has been reduced from around 290KB (64-bit) to respectively 203KB (32-bit), 160KB (16-bit) and 138KB (8-bit). That is, it makes little doubt that by combining partial strides and a reduction in the number of entries of the base predictor, good performance can be attained with a storage budget that is



(a) Impact of the recovery policy of the speculative window and FIFO update queue on speedup over *EOLE_4_60*.



(b) Impact of the size of the speculative window on speedup over *EOLE_4_60*. The recovery policy is *DnRDnR*.

Figure 5.6: Impact of the speculative window on performance for D-VTAGE with BeBoP. Speedup over *EOLE_4_60*.

comparable to that of the L1 D/I-Cache or the branch predictor.

5.6.2.3 Recovery Policy

Figure 5.6 (a) gives some insight on the impact of the speculative window recovery policy used for D-VTAGE with BeBoP, assuming an infinite window. In general the differences between the realistic policies are marginal, and on average, they behave equivalently. As a result, we only consider *DnRDnR* as it reduces the number of predictor accesses versus *Repred* and it marginally outperforms *DnRR*.

5.6.2.4 Speculative Window Size

Figure 5.6 (b) illustrates the impact of the speculative window size on D-VTAGE. When last values are not speculatively made available, some benchmarks are not accelerated as much as in the infinite window case e.g., *wupwise* (0.914 vs. 0.984),

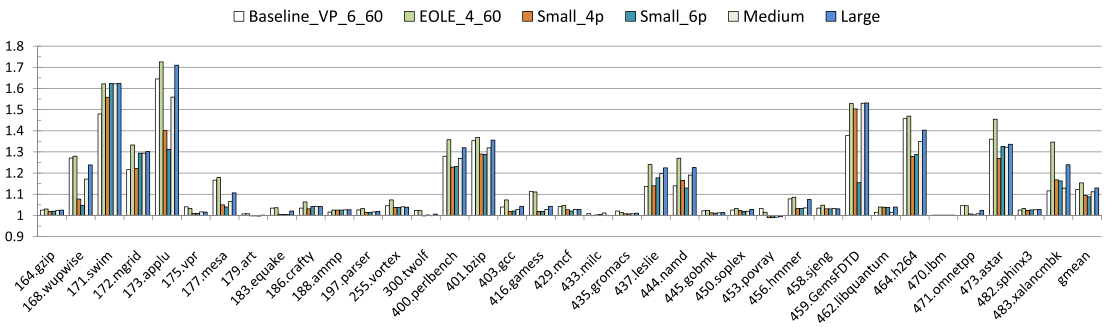


Figure 5.7: Speedup brought by different D-VTAGE configurations on top of an *EOLE_4_60* pipeline over *Baseline_6_60*.

applu (0.866 vs. 0.996), *bzip* (0.820 vs. 0.998) and *xalancbmk* (0.923 vs. 0.973). Having 56 entries in the window, however, provides roughly the same level of performance as an infinite number of entries, while using only 32 entries appears as a good tradeoff (average performance is a slowdown of 0.980 for 32-entry and 0.988 for ∞).

5.6.3 Putting it All Together

In previous experiments, we used a baseline *EOLE_4_60* model having a D-VTAGE predictor with 6 predictions per entry, a 2K-entry base component and six 256-entry tagged components. Because it also uses 64-bit strides, it requires roughly 290KB, not even counting the speculative window. Fortunately, we saw that the size of the base predictor could be reduced without too much impact on performance. Moreover, a speculative window with only a small number of entries performs well enough. Therefore, in this Section, we devise three predictor configurations based on the results of previous experiments as well as the observation that partial strides can be used in D-VTAGE [GVDB01]. To obtain a storage budget we consider as reasonable, we use 6 predictions per entry and six 128/256-entry tagged components. We then vary the size of the base predictor, the speculative window, and the stride length. Table 5.1 reports the resulting configurations.

In particular, for *Small* ($\simeq$ 16KB), we also consider a version with 4 predictions per entry but a base predictor twice as big (*Small_4p*). For *Medium* ($\simeq$ 32KB), we found that both tradeoffs have similar performance on average; for *Large* ($\simeq$ 64KB), 4 prediction per entry perform worse than 6 on average, even with a 1K-entry base predictor. Thus, we do not report results for the hypothetical *Medium_4p* and *Large_4p* for the sake of clarity.

To summarize, the implementation cost – i.e., storage budget, but also RAM structure (see Section 5.2.2.3) – is in the same range as that of the TAGE branch

Table 5.1: Final predictor configurations.

Predictor	#Base ent.	#Tagged ent.	#Spec. win. ent.	Str. len.	Size (KB)
Small_4p	256 (5b tag)	6×128	32 (15b tag)	8 bits	**17.26KB**
Small_6p	128 (5b tag)	6×128	32 (15b tag)	8 bits	**17.18KB**
Medium	256 (5b tag)	6×256	32 (15b tag)	8 bits	**32.76KB**
Large	512 (5b tag)	6×256	56 (15b tag)	16 bits	**61.65KB**

predictor considered in the study. In particular, even if we omit the – marginal – cost of sequence numbers and that of the FIFO update queue (around 5.6KB at worst[7]), the sizes reported in Table 5.1 have to be contrasted with the 44KB of the EV8 branch predictor [SFKS02] that was planned to be implanted on actual silicon.

Figure 5.7 reports the speedups obtained with these four predictor configurations on top of *EOLE_4_60*, over *Baseline_6_60*. We also show performance for *Baseline_VP_6_60* (first bar, white) and *EOLE_4_60* with an instruction-based D-VTAGE (second bar, green). We observe that even with around 32KB of storage and practical mechanisms, we manage to preserve most of the speedup associated with an idealistic implementation of VP. Maximum speedup decreases from 1.726 (*applu*) for *EOLE_4_60* to 1.622 (*swim*) for *Medium* and 1.710 (*applu*) for *Large*. Average speedup goes from 1.154 for *EOLE_4_60* down to 1.112 for *Medium* and 1.130 for *Large*. A noteworthy data point is *GemsFDTD*, where *Medium* and *Large* perform slightly better than *EOLE_4_60*. This is due to accuracy being slightly higher in *Medium* and *Large*, even if coverage is lower.

Allocating around 17KB of storage to the predictor, however, hurts performance noticeably: Average speedup over *Baseline_6_60* is only 1.088 for *Small_6p* (although max. is 1.622 in *swim*). Yet, it still represents a noticeable improvement. Interestingly, for this storage budget, using only 4 predictions per entry but a bigger base predictor (*Small_4p*) provides better performance on average: 1.095. This is because a smaller base predictor suffers from aliasing in *wupwise*, *applu*, *namd* and *GemsFDTD*.

As a result, we claim that BeBoP combined to D-VTAGE is an appealing possibility for an actual implementation of VP, even at reasonable storage budget (e.g., 16/32KB).

[7] 116 blocks in flight at worst in our simulation framework, if a block is an instruction and a branch.

5.7 Related Work on Predictor Storage & Complexity Reduction

To further improve space-efficiency, Sato et al. propose to use two tables for the Last Value Predictor (*2-mode scheme*) [SA00]. One contains full-width values and the other 8- or 16-bit values. At prediction time, both tables are accessed and the one with a tag match provides the prediction. Using 0/1-bit values is also a very space-efficient alternative [SA02].

Loh extends the 2-mode scheme by implementing several tables of differing width [Loh03]. The width of the result is predicted by a simple classifier at prediction time. By serializing width-prediction and table access, a single prediction table has to be accessed in order to predict.

Executing *load immediate* instructions for free in the front-end as suggested in 3.6.2.1 overlaps with the two propositions mentioned above, at lower cost.

Burtscher et al. reduce the size of a Last n value predictor by noting that most of the high-order bits are shared between recent values [BZ00]. Hence, they only keep one full-width value and the low-order bits of the $n-1$ previous ones. Moreover, they remark that if $n > 1$, then a stride can be computed on the fly, meaning that stride prediction can be done for nearly free.

Finally, Lee et al. leverage trace processors to reduce complexity [LWY00]. They decouple the predictor from the Fetch stage: Predictions are attributed to traces by the fill unit. Thus, they solve the access latency and port arbitration problems on the predictor. They also propose some form of block-based prediction by storing predictions in the I-cache [LY01]. However, a highly ported centralized structure is still required to build predictions at retire-time. To address this, Bhargava et al. replace the value predictor by a *Prediction Trace Queue* that requires a single read port and a single write port [BJ02]. Lastly, Gabbay and Mendelson also devise a hardware structure to handle multiple prediction per cycle by using highly interleaved tables accessed with addresses of instructions in the trace cache [GM98a]. Unfortunately, the scope of all these studies except [LY01] is limited to trace processors.

5.8 Conclusion

Value Prediction is a very attractive technique to enhance sequential performance in a context where power efficiency and cycle time entail that the instruction window cannot scale easily. However, many implementation details make VP hard to imagine on real silicon.

In this Chapter, we first described BeBoP, a block-based prediction scheme adapted to complex variable-length ISAs such as x86 as well as usually imple-

mented fetch mechanism. BeBoP contributes to reducing the number of ports required on the predictor by allowing several instructions to be predicted in a single access. Then, to reduce the footprint of the value predictor, we provided solutions for an actual implementation of D-VTAGE, a space-efficient tightly-coupled hybrid of VTAGE and a Stride predictor. In particular, a small speculative window to handle in-flight instructions that only requires an associative search on *read* operations, and a reduction in size thanks to partial strides and smaller tables. In a nutshell, the hardware complexity of D-VTAGE is similar to that of an aggressive TAGE branch predictor.

As a result, by combining the work presented in Chapters 3 and 4 with BeBoP, we provide one possible implementation of VP that requires around 32KB to around 64KB of additional storage. The issue width is reduced when compared to the baseline 6-issue model. Nonetheless, this implementation of VP is able to speed up execution by up to 62.2% with an average speedup of 11.2% (for the 32KB predictor version).

Conclusion

Over the past decade, computer architects have especially focused on efficient multi-core architectures and the challenges they offer, whether it be the memory model, coherency, the interconnect, shared caches management, and so on. Currently, the community appears to have reached a consensus in the fact that future multi- and many-cores will be *heterogeneous*. The rationale behind this choice is that even for parallel programs, the sequential portion caps the speedup that can be obtained by having more cores, as expressed by Amdahl's Law. Therefore, a processor should feature a sea of small cores to provide throughput (i.e., parallel performance) to parallel programs, and a few big cores to provide good performance on sequential code.

As a result, continuing to improve the performance of big cores is paramount as it will dictate the performance of sequential programs as well as parallel programs that are able to saturate the small cores. In the uni-processor era, increasing sequential performance could be done by increasing clock speed and implementing more complex micro-architectures. However, due to the increase in power consumption, scaling the frequency is no longer considered an option. Moreover, scaling the processor structures able to leverage ILP increases cycle time, power consumption and complexity. Therefore, alternative ways of improving sequential performance must be considered.

In this thesis work, we have revisited Value Prediction (VP), a less intuitive means to increase performance. Indeed, instead of being able to uncover more ILP from the program by being able to look further ahead (i.e., by having a bigger instruction window), the processor *creates* more ILP by breaking some true dependencies in the existing instruction window. As a result, performance increases without lengthening the processor electric critical path, usually located in the out-of-order scheduling logic.

In particular, we have addressed the main remaining concerns that were preventing Value Prediction from being implemented on actual silicon. We have first shown that prediction validation can be done at Commit, meaning that the VP hardware is conscripted to the in-order frontend and the in-order backend, with few interactions with the out-of-order engine. This is of particular importance since validating predictions in the out-of-order engine adds some strong

constraints on an already very complex piece of hardware. In the meantime, we have proposed a new, more practical value predictor leveraging the global branch history to generate predictions, VTAGE. We showed that it could be cleverly hybridized with a pre-existing Stride predictor to form D-VTAGE in order to improve prediction coverage even more. Although this hybrid requires a speculative window as it needs the result of the previous instance to predict for the current one, it does not suffer from shortcomings of other context-based predictors, such as a long prediction critical path and the need to manage local value histories.

Second, we have shown that Value Prediction can in fact be leveraged to reduce the aggressiveness of the out-of-order engine, by executing many instructions in-order either in the frontend or at Commit. As a direct result, the number of physical register file (PRF) ports required by the out-of-order engine is reduced. Moreover, since prediction and validation are done in-order, banking the register file can greatly decrease the number of PRF ports required by the VP hardware. Consequently, we can obtain performance on-par with a wide processor implementing VP but using a smaller out-of-order engine and a PRF of similar complexity as a processor without VP. This is especially interesting as it should reduce the impact of the out-of-order engine on cycle time and the overall processor power consumption.

Third, and last, we have devised a prediction scheme able to provide several predictions each cycle using only single-ported arrays. Block-based prediction leverages the block-based fetch mechanism of modern processors by associating a block of predictions to a block of instructions. Then, for a given block of instructions, a single read has to be performed to retrieve predictions for that block. Furthermore, by banking the D-VTAGE predictor in an efficient manner, it is even possible to support two instruction blocks each cycle with only the Last Value Table being dual-ported (only if the I-Cache is also dual ported). Finally, we take advantage of block-based prediction to implement the speculative window required for stride-based predictors. We propose to use a small fully-associative speculative window containing inflight predictions that is managed at the block level rather than the instruction level.

Although we do not claim to have addressed every implementation issues associated with Value Prediction, we are confident that our solutions tackle major flaws that can be observed in pre-existing Value Prediction design. We also make the case that Value Prediction need not be seen as a mechanism that would increase complexity in every stage of the pipeline. Rather, it allows to increase performance while re-distributing execution to different stages of the pipeline, reducing the out-of-order engine complexity in the process. As a result, we hope that Value Prediction will be re-considered by the industry in the near future, since some programs do not benefit from throwing more cores at them or putting a GPU on the die.

Perspectives

As previously mentioned, the ability to know the result of an instruction before it is executed opens the door to many optimizations. In the future, we aim to leverage the Value Prediction infrastructure to improve on two mechanisms.

First, even in the absence of a value predictor (but in the presence of a validation mechanism), we aim to improve on memory dependency prediction by revisiting memory renaming. In particular, by identifying producer/consumer pairs of stores and loads, the produced value can be forwarded to the destination register of the load speculatively, before usual store-to-load forwarding. This can be leveraged to increase the store-to-load forwarding capacity of the processor without increasing the size of the store queue. In particular, a simple FIFO buffer can hold older (i.e., written-back) store instructions and the memory renaming scheme pinpoint which entry to access for a given load.

Using this specific flavor of value prediction, we also aim to improve the accuracy of the branch predictor for some hard-to-predict branches. For instance, by looking at the sign of a forwarded value that is close to a branch, we might be able to cancel a wrongly predicted branch quicker than it is usually done (at Execute). This has already been generalized to all value predictions produced by a regular value predictor, but since both TAGE and (D-)VTAGE use the global branch history to predict, we do not expect that the value predictor can help the branch predictor when state-of-the art hardware is considered.

Second, we will study using value prediction to skip the execution of whole functions, as long as they are *pure*. In particular, by predicting the parameters of a function, a small memoization table can be accessed, and the result fetched before the parameters are actually available. In that case, the instruction stream can be redirected to the call site using the address present on the stack. The main challenge will be the identification of pure functions, since only these functions can be skipped. On the one hand, software can identify pure functions more easily but requires ISA changes to convey the information to the hardware. On the other hand, the hardware may have to be too conservative when identifying pure functions because of the various calling conventions used by the software.

Author's Publications

[PS14a] A. Perais and A. Seznec. EOLE: Paving the way for an effective implementation of value prediction. In *Proceedings of the International Symposium on Computer Architecture*, 2014.

[PS14b] A. Perais and A. Seznec. Practical data value speculation for future high-end processors. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2014.

[PS15a] A. Perais and A. Seznec. BeBoP: A cost effective predictor infrastructure for superscalar value prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2015.

[PS15b] A. Perais and A. Seznec. EOLE: Toward a practical implementation of value prediction. *IEEE Micro's Top Picks from Computer Architecture Conferences*, 2015.

[PSM+15] A. Perais, A. Seznec, P. Michaud, A. Sembrant, and E. Hagersten. Cost-effective speculative scheduling in high performance processors. In *Proceedings of the International Symposium on Computer Architecture*, 2015.

Bibliography

[ACR95] P.S. Ahuja, D.W. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. In *Proceedings of the International Symposium on Microarchitecture*, pages 36–45, 1995.

[AGGG01] J. L. Aragón, J. González, J. M. García, and A. González. Selective branch prediction reversal by correlating with data values and control flow. In *Proceedings of the International Conference on Computer Design*, pages 228–233, 2001.

[Amd67] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, pages 483–485, 1967.

[AMD98] Advanced Micro Devices. AMD K6-III Processor Data Sheet. pages 11–12, 1998.

[Aus99] T. M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the International Symposium on Microarchitecture*, pages 196–207, 1999.

[BBB+11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, Aug. 2011.

[BJ02] R. Bhargava and L. K. John. Latency and energy aware value prediction for high-frequency processors. In *Proceedings of the International Conference on Supercomputing*, pages 45–56, 2002.

[BJ03] M. Burtscher and M. Jeeradit. Compressing extended program traces using value predictors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 159–169, 2003.

[Bur00] M. Burtscher. *Improving Context-Based Load Value Prediction*. PhD thesis, University of Colorado, 2000.

[Bur04] M. Burtscher. VPC3: A fast and effective trace-compression algorithm. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 167–176, 2004.

[BZ99] M. Burtscher and B. G. Zorn. Exploring last-n value prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 66–76, 1999.

[BZ00] M. Burtscher and B. G. Zorn. Hybridizing and coalescing load value predictors. In *Proceedings of the International Conference on Computer Design*, pages 81–92, 2000.

[cbp14] Championship Branch Prediction. *Held in conjunction with the International Symposium on Computer Architecture, http://www.jilp.org/cbp2014/program.html*, 2014.

[CE98] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the International Symposium on Computer Architecture*, pages 142–153, 1998.

[CRT99] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *Proceedings of the International Symposium on Computer Architecture*, pages 64–74, 1999.

[CSK+99] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the International Symposium on Computer Architecture*, pages 186–195, 1999.

[EA02] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *Proceedings of the International Symposium on Computer Architecture*, pages 37–46, 2002.

[EV93] R. J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37(4):547–564, Jul. 1993.

[FCJV97] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster architecture: reducing cycle time through partitioning. In *Proceedings of the International Symposium on Microarchitecture*, pages 149–159, 1997.

[Fra93] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin at Madison, 1993.

[FRB01] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. In *Proceedings of the International Symposium on Computer Architecture*, pages 74–85, 2001.

[FRPL05] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta. Continuous optimization. In *Proceedings of the International Symposium on Computer Architecture*, pages 86–97, 2005.

[GAV11] M. Golden, S. Arekapudi, and J. Vinh. 40-entry unified out-of-order scheduler and integer execution unit for the AMD Bulldozer x86_64 core. In *Proceedings of the International Conference on Solid-State Circuits Conference, Digest of Technical Papers*, pages 80–82, 2011.

[GBJ98] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Proceedings of the Design Automation Conference*, pages 726–731, 1998.

[GG97] J. González and A. González. Speculative execution via address prediction and data prefetching. In *Proceedings of the International Conference on Supercomputing*, pages 196–203, 1997.

[GG98b] J. González and A. González. The potential of data value speculation to boost ILP. In *Proceedings of the International Conference on Supercomputing*, pages 21–28, 1998.

[GG99] J. González and A. González. Control-flow speculation through value prediction for superscalar processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 57–65, 1999.

[GM96] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Technical Report TR-1080, Technion-Israel Institute of Technology, 1996.

[GM98a] F. Gabbay and A. Mendelson. The effect of instruction fetch bandwidth on value prediction. In *Proceedings of the International Symposium on Computer Architecture*, pages 272 –281, 1998.

[GM98b] F. Gabbay and A. Mendelson. Using value prediction to increase the power of speculative execution hardware. *ACM Transactions on Computer Systems*, pages 234–270, Aug. 1998.

[GRA+03a] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7, May 2003.

[GVDB01] B. Goeman, H. Vandierendonck, and K. De Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 207–216, 2001.

[Int12] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2012.

[JRB+98] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proceedings of the International Symposium on Microarchitecture*, pages 216–225, 1998.

[JRS96] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the International Symposium on Microarchitecture*, pages 142–152, 1996.

[KL03] I. Kim and M. H. Lipasti. Half-price architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 28–38, 2003.

[KL04] I. Kim and M. H. Lipasti. Understanding scheduling replay schemes. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 198–209, 2004.

[KMW98] R. E. Kessler, E. J. Mclellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the International Conference on Computer Design*, pages 90–95, 1998.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sep. 1979.

[Loh03] G. H. Loh. Width prediction for reducing value predictor size and power. *First Value Prediction Workshop, held in conjunction with the International Symposium on Computer Architecture*, 2003.

[LPD+12] A. Lukefahr, S. Padmanabha, R. Das, F.M. Sleiman, R. Dreslinski, T.F. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the International Symposium on Microarchitecture*, pages 317–328, 2012.

[LS96] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the International Symposium on Microarchitecture*, pages 226–237. IEEE Computer Society, 1996.

[LWS96] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[LWY00] S-J. Lee, Y. Wang, and P-C. Yew. Decoupled value prediction on trace processors. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 231–240, 2000.

[LY00] S-J. Lee and P-C. Yew. On some implementation issues for value prediction on wide-issue ILP processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 145–156, 2000.

[LY01] S-J. Lee and P-C. Yew. On table bandwidth and its update delay for value prediction on wide-issue ILP processors. *IEEE Transactions on Computers*, 50(8):847–852, Aug. 2001.

[LY02] S-J. Lee and P-C. Yew. On augmenting trace cache for high-bandwidth value prediction. *IEEE Transactions on Computers*, 51(9):1074–1088, Sep. 2002.

[MBS07] A. A. Merchant, D. D. Boggs, and D. J. Sager. Processor with a replay system that includes a replay queue for improved throughput, 2007. US Patent 7,200,737.

[MLO01] E. Morancho, J. M. Llabería, and À. Olivé. Recovery mechanism for latency misprediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 118–128, 2001.

[Moo98] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86:82–85, Jan. 1998.

[MS01] A. A. Merchant and D. J. Sager. Computer processor having a checker, 2001. US Patent 6,212,626.

[MSBU00] A. A. Merchant, D. J. Sager, D. D. Boggs, and M. D. Upton. Computer processor with a replay system having a plurality of checkers, 2000. US Patent 6,094,717.

[MSC+01] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *the International Symposium on Microarchitecture*, pages 328–337, 2001.

[NGS99] T. Nakra, R. Gupta, and M. L. Soffa. Global context-based value prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 4–12, 1999.

[PHC03] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 244–, 2003.

[PJS97] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the International Symposium on Computer Architecture*, pages 206–218, 1997.

[PSR05] V. Petric, T. Sha, and A. Roth. RENO: a rename-based instruction optimizer. In *Proceedings of the International Symposium on Computer Architecture*, pages 98–109, 2005.

[RFK+98] B. Rychlik, J. W. Faistl, B.P. Krug, A. Y. Kurland, J. J. Sung, M. N. Velev, and J. P. Shen. Efficient and accurate value prediction using dynamic classification. Technical report, Carnegie Mellon University, 1998.

[RFKS98] B. Rychlik, J. W. Faistl, B. Krug, and J. P. Shen. Efficacy and performance impact of value prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 148–154, 1998.

[RZ06] N. Riley and C. B. Zilles. Probabilistic counter updates for predictor hysteresis and stratification. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 110–120, 2006.

[SA00] T. Sato and I. Arita. Table size reduction for data value predictors by exploiting narrow width values. In *Proceedings of the International Conference on Supercomputing*, pages 196–205, 2000.

[SA02] T. Sato and I. Arita. Low-cost value predictors using frequent value locality. In *Proceedings of the International Symposium on High Performance Computing*, pages 106–119, 2002.

[SBP00] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the International Symposium on Microarchitecture*, pages 57–66, 2000.

[SC02] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 25–34, 2002.

[Sez11b] A. Seznec. A new case for the TAGE branch predictor. In *Proceedings of the International Symposium on Microarchitecture*, pages 117–127, 2011.

[Sez11c] A. Seznec. Storage free confidence estimation for the TAGE branch predictor. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 443–454, 2011.

[SFKS02] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha EV8 conditional branch predictor. In *Proceedings of the International Symposium on Computer Architecture*, pages 295–306, 2002.

[SGC01b] Dave Sager, Desktop Platforms Group, and Intel Corporation. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 1:2001, 2001.

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

[Smi81] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the International Symposium on Computer Architecture*, pages 135–148, 1981.

[SMV11] E. Safi, A. Moshovos, and A. Veneris. Two-stage, pipelined register renaming. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(10):1926–1931, Oct. 2011.

[SPR00] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the International Conference on Architectural Support*

for Programming Languages and Operating Systems, pages 257–268, 2000.

[SS97a] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the International Symposium on Microarchitecture*, pages 248–258, 1997.

[SS98a] Y. Sazeides and J. E. Smith. Implementations of context based value predictors. Technical report, University of Wisconsin at Madison, 1998.

[SS98b] A. Sodani and G. S. Sohi. Understanding the differences between value prediction and instruction reuse. In *Proceedings of the International Symposium on Microarchitecture*, pages 205–215, 1998.

[SSO+10] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, Jul. 2010.

[Staa] Standard Performance Evaluation Corporation. CPU2000.

[Stab] Standard Performance Evaluation Corporation. CPU2006.

[STR02] A. Seznec, E. Toullec, and O. Rochecouste. Register write specialization, register read specialization: a path to complexity-effective wide-issue superscalar processors. In *Proceedings of the International Symposium on Microarchitecture*, pages 383–394, 2002.

[TF01] R. Thomas and M. Franklin. Using dataflow based context for accurate value prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 107–117, 2001.

[TMJHJ08] S. Thoziyoor, N. Muralimanohar, A. Jung Ho, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Laboratories, 2008.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, pages 25–33, Jan. 1967.

[TP08] F. Tseng and Y. N. Patt. Achieving out-of-order performance with almost in-order complexity. In *Proceedgins of the International Symposium on Computer Architecture*, pages 3–12, 2008.

[TS99] D. M. Tullsen and J. S. Seng. Storageless value prediction using prior register values. In *Proceedings of the International Symposium on Computer Architecture*, pages 270–279, 1999.

[TTC02] E. S. Tune, D. M. Tullsen, and B. Calder. Quantifying instruction criticality. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 104–113, 2002.

[WB96] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 179–184, 1996.

[WF97] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the International Symposium on Microarchitecture*, pages 281–290, 1997.

[YERJ] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the International Symposium on Computer Architecture*.

[YP93] T-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the International Symposium on Computer Architecture*, pages 257–266, 1993.

[ZFC03] H. Zhou, J. Flanagan, and T. M. Conte. Detecting global stride locality value streams. In *Proceedings of the International Symposium on Computer Architecture*, pages 324–335, 2003.

[ZK98] V. Zyuban and P. Kogge. The energy complexity of register files. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 305–310, 1998.

[ZYFRC00] H. Zhou, C. Ying Fu, E. Rotenberg, and T. Conte. A study of value speculative execution and misspeculation recovery superscalar microprocessors. Technical report, North Carolina State University, 2000.

List of Figures

Abstract

Although currently available general purpose microprocessors feature more than 10 cores, many programs remain mostly sequential. This can either be due to an inherent property of the algorithm used by the program, to the program being old and written during the uni-processor era, or simply to time to market constraints, as writing and validating parallel code is known to be hard. Moreover, even for parallel programs, the performance of the sequential part quickly becomes the limiting improvement factor as more cores are made available to the application, as expressed by Amdahl's Law. Consequently, increasing sequential performance remains a valid approach in the multi-core era.

Unfortunately, conventional means to do so – increasing the out-of-order window size and issue width – are major contributors to the complexity and power consumption of the chip. In this thesis, we revisit a previously proposed technique that aimed to improve performance in an orthogonal fashion: Value Prediction (VP). Instead of increasing the execution engine aggressiveness, VP improves the utilization of existing resources by increasing the available Instruction Level Parallelism.

In particular, we address the three main issues preventing VP from being implemented. First, we propose to remove validation and recovery from the execution engine, and do it in-order at Commit. Second, we propose a new execution model that executes some instructions in-order either before or after the out-of-order engine. This reduces pressure on said engine and allows to reduce its aggressiveness. As a result, port requirement on the Physical Register File and overall complexity decrease. Third, we propose a prediction scheme that mimics the instruction fetch scheme: Block Based Prediction. This allows predicting several instructions per cycle with a single read, hence a single port on the predictor array. This three propositions form a possible implementation of Value Prediction that is both realistic and efficient.

Résumé

Bien que les processeurs actuels possèdent plus de 10 cœurs, de nombreux programmes restent purement séquentiels. Cela peut être dû à l'algorithme que le programme met en œuvre, au programme étant vieux et ayant été écrit durant l'ère des uni-processeurs, ou simplement à des contraintes temporelles, car écrire du code parallèle est notoirement long et difficile. De plus, même pour les programmes parallèles, la performance de la partie séquentielle de ces programmes devient rapidement le facteur limitant l'augmentation de la performance apportée par l'augmentation du nombre de cœurs disponibles, ce qui est exprimé par la loi d'Amdahl. Conséquemment, augmenter la performance séquentielle reste une approche valide même à l'ère des multi-cœurs.

Malheureusement, la façon conventionnelle d'améliorer la performance (augmenter la taille de la fenêtre d'instructions) contribue à l'augmentation de la complexité et de la consommation du processeur. Dans ces travaux, nous revisitons une technique visant à améliorer la performance de façon orthogonale: La prédiction de valeurs. Au lieu d'augmenter les capacités du moteur d'exécution, la prédiction de valeurs améliore l'utilisation des ressources existantes en augmentant le parallélisme d'instructions disponible.

En particulier, nous nous attaquons aux trois problèmes majeurs empêchant la prédiction de valeurs d'être mise en œuvre dans les processeurs modernes. Premièrement, nous proposons de déplacer la validation des prédictions depuis le moteur d'exécution vers l'étage de retirement des instructions. Deuxièmement, nous proposons un nouveau modèle d'exécution qui exécute certaines instructions dans l'ordre soit avant soit après le moteur d'exécution dans le désordre. Cela réduit la pression exercée sur ledit moteur et permet de réduire ses capacités. De cette manière, le nombre de ports requis sur le fichier de registre et la complexité générale diminuent. Troisièmement, nous présentons un mécanisme de prédiction imitant le mécanisme de récupération des instructions: La prédiction par blocs. Cela permet de prédire plusieurs instructions par cycle tout en effectuant une unique lecture dans le prédicteur. Ces trois propositions forment une mise en œuvre possible de la prédiction de valeurs qui est réaliste mais néanmoins performante.