Shared Frontend for Manycore Server Processors

THÈSE N^{O} 6669 (2015)

PRÉSENTÉE LE 3 SEPTEMBRE 2015

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ARCHITECTURE DE SYSTÈMES PARALLÈLES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Ilknur Cansu KAYNAK

acceptée sur proposition du jury:

Prof. V. Cevher, président du jury
Prof. B. Falsafi, Prof. B. R. Grot, directeurs de thèse
Prof. J. Emer, rapporteur
Prof. A. Seznec, rapporteur
Prof. P. Ienne, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2015

Acknowledgements

First and foremost, I would like to thank my parents, Nadide and Fatih, for valuing my education more than anything else and always giving me and my sister the highest priority in their lives. Since my childhood, my parents have always broadened my horizons, set the next challenging goal for me to achieve and then endlessly supported me to achieve the goals that I set for myself as I grew up. They tirelessly put up with my whining without a single complaint and supported me whenever I was lacking confidence and afraid that I was going to fail. I could not have asked for any better parents and I will always feel in debt to them. My sister, Duygu, has always been a source of joy for me since the day she was born. Along with my parents, she is my biggest supporter. I will always be proud to be part of this great team of four.

I would like to thank Babak Falsafi, my advisor, for giving me the opportunity to start this journey. I learned countless things from Babak throughout this journey, the most important being perfectionism. Babak always seeks the very best both at work and in life, never settles for mediocre things, and always has the passion and excitement to motivate people around him to work to achieve the perfect. I will always try to emulate his perfectionism for the rest of my life. I hope, one day, I will be able to achieve his level of perfectionism. Thank you, Babak, for your support and every single bit you have taught me.

Boris Grot, my co-advisor, has been a source of inspiration for me. Although I met Boris half way through my PhD journey, I have incredibly benefited from his knowledge and experience. Working with Boris has always been a pleasure as he is always excited about discovering new

things and pursuing new directions fearlessly. I will always try to emulate his positiveness, friendliness and excitement for the rest of my life.

I would like to thank Joel Emer, Andre Seznec and Paolo Ienne for serving on my thesis committee. Their feedback and suggestions improved this thesis tremendously.

Mike Ferdman, as the most senior member of the group at the time of my arrival at EPFL, was my guide to grad school. Mike tirelessly tried to convey his knowledge to me and was always there to answer my questions and brainstorm with me. He taught me countless things ranging from systems programming to paper writing. I would like to thank Mike for being so patient, willing to teach, and holding my hand in the first few years of this journey. And of course, thanks for being a great friend and entertaining me during Onur's absences, along with Alisa.

Djordje Jevdjic and Stavros Volos have been great classmates, collaborators, friends, and travel companions. These two guys were always there to help me whenever I needed them, have endless discussions about research, politics and everything else and go for beer. As a group of four, we shared so many fun moments both at work and outside work that I will never forget. Javier Picorel, the latest addition to the team, brought along more fun. This journey would not have been so much fun without these three guys. They will always remain as my brothers.

I would like to thank every single member of the PARSA group, Sotiria Fytraki, Pejman Lotfi-Kamran, Almutaz Adileh, Evangelos Vlachos, Jennifer Sartor, Alexandros Daglis, Stanko Novakovic, Yorgos Psaropoulos, Nooshin Mirzadeh, Mario Drumond, Dmitrii Ustiugov and Nancy Chong, for their friendship and contributions to this long journey in various ways. I will always miss the PARSA spirit; the unconditional help and support. A special member of the PARSA group, Effi Georgala, has been a great friend to chat about girlie books, fashion, food and many other things. I would like to thank Effi for being such an intimate friend and introducing me to the nitty-gritty of phonetics. Special thanks to Damien Hilloulin for translating the abstract of this thesis into French. Many thanks to Stephanie Baillargues and Valerie Locca for taking

care of all the administrative work. Thanks to Rodolphe Buret for express technical support whenever needed.

I am grateful to Oguz Ergin, my undergrad advisor, for making me like computer architecture and supervising my senior project, which in turn led me to pursue a PhD in computer architecture. I would like to thank Oguz for his continuous support and friendship since my undergrad years.

During my six years in Lausanne, the Turkish gang has been my only connection to Turkey and the Turkish culture. I was extremely lucky to have Duygu Ceylan and Pinar Tozun as my closest Turkish friends in Lausanne. I would like to thank Duygu and Pinar for their close friendship and hospitality, the Turkish meals and parties we had together. Baris Can Kasikci and Kerem Kapucu brothers were the source of ceaseless laughter and fun nights. I will really miss this very special group of people.

Last but not least, I would like to thank Onur Kocberber, for always being there for me. I would not have been able to make it this far, if I had not had Onur's full support from the beginning till the end. Onur has been an excellent collaborator, lab-mate, friend, and travel companion throughout this journey. Most importantly, I could not have asked for a better partner than Onur. I feel very lucky that our ways crossed long before this journey, we finished this journey and will be moving on to new journeys together.

I am grateful to IBM and Google for valuing my research and supporting me with an IBM PhD fellowship and Google Anita Borg scholarship.

Abstract

Instruction-supply mechanisms, namely the branch predictors and instruction prefetchers, exploit recurring control flow in an application to predict the application's future control flow and provide the core with a useful instruction stream to execute in a timely manner. Consequently, instruction-supply mechanisms aggressively incorporate control-flow condition, target, and instruction cache access information (i.e., control-flow metadata) to improve performance. Despite their high accuracy, thus performance benefits, these predictors lead to major silicon provisioning due to their metadata storage overhead. The storage overhead is further aggravated by the increasing core counts and more complex software stacks leading to major metadata redundancy: (i) across cores as the metadata of cores running a given server workload significantly overlap, (ii) within a core as the control-flow metadata maintained by disparate instruction-supply mechanisms overlap significantly.

In this thesis, we identify the sources of redundancy in the instruction-supply metadata and provide mechanisms to share metadata across cores and unify metadata for disparate instruction-supply mechanisms. First, homogeneous server workloads running on many cores allow for metadata sharing across cores, as each core executes the same types of requests and exhibits the same control flow. Second, the control-flow metadata maintained by individual instruction-supply mechanisms, despite being at different granularities (i.e., instruction vs. instruction block), overlap significantly, allowing for unifying their metadata. Building on these two observations, we eliminate the storage overhead stemming from metadata redundancy in manycore server processors through a specialized shared frontend, which enables sharing metadata across cores and unifying metadata within a core without sacrificing the performance benefits provided by private and disparate instruction-supply mechanisms.

Abstract

Key words: server workloads, manycore server processors, instructions, prefetching, branch prediction, shared predictor metadata.

Résumé

Les structures d'approvisionnement en instructions, à savoir les prédicteurs de branchement et les prefetchers d'instructions, exploitent les flux de contrôle récurrent dans une application pour prédire le flux de contrôle futur et ainsi fournir à temps au cœur du processeur un flux d'instructions utiles. Par conséquent, ces mécanismes intègrent des informations sur le flot de contrôle tels que les conditions, la cible, et l'accès au cache d'instructions (c'est à dire, des métadonnées sur le flot de contrôle) pour améliorer les performances. En dépit de leur grande précision, améliorant ainsi les performances, ces prédicteurs nécessitent beaucoup de place et de silicium à cause de la taille de ces métadonnées. Cela est encore aggravé par l'augmentation constante du nombre de cœurs dans les processeurs et de la complexité des piles logicielles, amenant à des métadonnées de plus en plus redondantes : (i) entre les cœurs car les métadonnées des cœurs exécutant une même application serveur donnée se recoupent de façon significative, (ii) dans chaque cœur car les métadonnées stockées par les différentes structures sont elles aussi très redondantes.

Dans ce travail, nous identifions les sources de redondance dans les métadonnées et fournissons des mécanismes pour partager celles-ci entre les cœurs et au sein du même cœur entre des structures disparates. Tout d'abord, nous observons que les applications serveur homogènes s'exécutant sur un processeur comportant de nombreux cœurs permettent le partage des métadonnées entre les cœurs, étant donné que chaque cœur exécute les mêmes types de requêtes et suit le même flux de contrôle. Deuxièmement, nous observons que les métadonnées sur le flux de contrôle maintenues individuellement par les différentes structures, en dépit d'être à différentes granularités (instruction versus blocs d'instructions par exemple), se chevauchent de façon significative, ce qui permet de les rassembler. En s'appuyant sur

ces deux observations, nous minimisons le coût de stockage découlant de la redondance des métadonnées dans les processeurs multi-cœurs pour serveurs en partageant ces métadonnées entre les cœurs et au sein d'un même cœur entre les différentes structures, sans pour autant sacrifier les gains associés à l'utilisation de structures privées et spécialisées.

Mots clefs : applications serveur, processeurs serveur multi-cœurs, instructions, prefetching, prédiction de branchement, prédicteur à métadonnées partagées.

Contents

List of Figures

List of Tables

1 Introduction

Servers power today's information-centric world. Server workloads include but are not limited to business analytics, online transaction processing, media streaming and web search. The steadily increasing demand for these services from their global user base mandates constant performance increase in servers.

While the demand for server performance continues to grow, the breakdown of Dennard scaling (i.e., the ability to increase performance per unit area while keeping power density constant) has put an end to the continuous performance increase with each new technology node [24, 32]. As a result, improving efficiency has become the primary challenge in meeting the ever-increasing performance requirements of server processors. Consequently, both prior research and industry efforts have attempted to address the inefficiencies stemming from the mismatch between what server workloads demand and what commodity server processors provide.

Existing server chips from the major processor vendors, Intel and AMD, rely on architectures developed for the general-purpose market, with only minor modifications to the cache hierarchy to accommodate larger working sets in server workloads. These general-purpose commodity processors are typically designed for computation-intensive desktop and engineering workloads, featuring only a handful of wide-issue and high-frequency cores. An

important class of emerging server workloads, however, spend most of their execution time waiting for memory while performing pointer chasing and exhibit moderate computational intensity, while enjoying abundant request-level parallelism [21, 26, 53], similar to conventional database workloads such as transaction processing and data analytics [2, 45, 63, 82]. As a result, server workloads severely underutilize the computation resources in general-purpose processors, leading to extreme inefficiency.

To maximize the throughput within a given area and power budget, recent research has advocated the use of processors with many cores that are specialized to match the minimal computation needs of server workloads without sacrificing single-threaded performance, just a few MBs of on-chip cache to maintain only the shared instruction working set (due to low on-chip data reuse) and a specialized on-chip interconnect to provide fast access from cores to the instructions in the shared cache, essentially streamlining the instruction-supply path [17, 29, 53].

Even after these first steps of specialization to improve server efficiency, there is still much more room for improvement in the specialized server design space to optimize the instruction-supply path, which is overwhelmed by the large instruction working sets of server workloads. We observe that server workloads are homogeneous, meaning that each core executes the same types of tasks as all the other cores running the same workload. However, all cores are treated as standalone IP blocks, without taking into account the homogeneity of a given server workload. This results in abundant redundancy in the overall chip stemming from the same control-flow information being maintained privately in each core's instruction-supply path. Therefore, instruction supply still remains as a key efficiency bottleneck in terms of real-estate, performance and energy in manycore server processors.

This thesis is the first to propose and evaluate the next-generation of specialized processors for server workloads with a shared frontend based on the observation that the control-flow information within a core and across cores running a given homogeneous workload is replicated redundantly.

1.1 Instruction Supply in Servers

One of the distinguishing features that is common across many conventional and emerging server workloads is the large instruction working sets. Server software implements complex functionality in a stack of over a dozen layers of services with well-defined abstraction and interfaces from the application all the way through the system. Control flow of each individual request is quite complex and typically spans multiple layers of the software stack, including the application itself, a database engine, a web server, various libraries and the operating system, resulting in multi-megabyte instruction working sets [26, 31, 38, 43, 82].

Continuously supplying a core with a useful instruction stream to execute requires predicting the future flow of instructions correctly and fetching those instructions in a timely manner. The shortage of a continuous and useful instruction stream significantly degrades the overall system performance in several ways. First, the lack of useful instructions to execute results in core stalls preventing the core from making forward progress. Consequently, because the number of instructions that co-exist in the instruction window of a core decreases, the number of independent memory accesses that can be concurrently issued also decreases. As a result, the lack of concurrent memory accesses exposes the core to the entire latency of each long-latency memory access. Overall, the absence of useful instructions to execute and concurrent memory accesses to issue causes severe underutilization of processor resources and significantly hurts overall performance.

Eliminating the performance degradation caused by the lack of a continuous instruction stream necessitates accurate and effective instruction-supply mechanisms to predict and fetch the upcoming stream of correct-path instructions. The instruction-supply mechanisms, namely the branch predictors and instruction prefetchers, exploit the recurring control flow in applications to predict future control flow and proactively fetch instructions respectively. To do so, they maintain the history of an application's control flow and leverage this history to predict the future control flow.

The lack of a continuous and useful instruction stream to execute can be due to two reasons:

(i) instruction cache misses. (ii) branch mispredictions and branch target misses.

In order to mitigate the stalls caused by instruction cache misses, researchers have proposed various instruction prefetching techniques to predict future instruction cache misses and fetch the instructions from the lower levels of cache hierarchy ahead of core demand to hide the long access latencies from the core frontend. Simple next-line prefetchers [72] that are common in today's processors are effective at predicting misses to sequential instruction cache blocks without any storage overhead. However, they fail to predict misses caused by diverging control flow upon taken branches and transitions between application, library and OS codes. Moreover, next-line prefetchers are not timely enough to fully hide the access latency of lower level caches and increasing their lookahead causes pollution due to frequent control flow divergences.

More advanced prefetchers leverage the branch predictor to predict misses to non-sequential blocks in addition to misses to sequential blocks [14, 20, 65, 77, 86]. To do so, these prefetchers let the branch predictor run ahead of the fetch unit to explore the future control flow and prefetch the instruction blocks that are not present in the cache along the way. Although these prefetchers can eliminate more misses than the simple next-line prefetcher, they have two major limitations. First, because the branch predictor can generate one or two predictions per cycle, its lookahead is substantially limited to a few basic blocks, especially when the branch predictor is exploring the local control flow spanning cache-resident instruction blocks (i.e., typically due to loop iterations). Second, because the branch predictor speculatively runs ahead of the fetch unit to provide sufficient prefetch lookahead, its miss rate geometrically compounds, increasingly predicting the wrong-path instructions along the way until receiving feedback from the core backend upon the resolution of the first mispredicted branch. As a result, instruction prefetchers leveraging branch predictors provide only marginal performance benefits.

Because server workloads exhibit high recurrence in processing a large number of similar requests, their control flow tends to repeat significantly. To overcome the limitations of next-

line and branch-predictor-directed prefetchers, the state-of-the-art instruction prefetchers for server workloads rely on temporal streaming to record, and subsequently replay, entire sequences of instructions [27, 28].These instruction streaming mechanisms have been shown to be highly effective at eliminating the vast majority of frontend stalls stemming from instruction cache misses. However, due to the large instruction footprints and small, yet numerous differences in the control flow among the various types of requests of a given workload, instruction streaming mechanisms incur significant storage overheads to accommodate the instruction stream history.

Similar to instruction prefetching, branch prediction is a vital component for high performance in today's processors. While the instruction prefetcher predicts and fetches the instruction blocks that will be needed by the core frontend ahead of time, the branch predictor predicts the future instruction flow long before the branch instructions are actually executed to provide the core with a continuous useful instruction stream to execute. The branch direction predictor predicts whether a branch will be taken or not, while the branch target buffer (BTB) provides the address of the next instruction to execute (i.e., target instruction) for predicted taken branches. The BTB essentially caches the target address of each taken branch upon its retirement and uses the cached target address as the predicted target address for subsequent predictions of the same branch.

BTB capacity is limited due to the low access latency constraints as it is required to provide a prediction every cycle. As a result, the large instruction working sets of server workloads defy the limited BTB capacities, causing frequent target address mispredictions (i.e., misfetches) and providing the core with the wrong-path instruction flow (i.e., sequential) upon taken branches [4, 11, 12, 35].

To mitigate the performance penalty of misfetches due to frequent misses in the BTB, prior research has proposed employing hierarchical (i.e., two-level) BTBs [64]. The first-level BTB exhibits low hit rate and low access latency, while the second-level BTB has higher hit rate but higher latency due to its larger capacity. The second-level BTB can later override a wrong

target prediction provided by the first level upon the completion of the corresponding lookup. Although the hierarchical BTB design provides a trade-off between hit rate and access latency, it still exposes the core to the high access latency of the second-level BTB in the case of misses in the first-level BTB.

More recent work has proposed various prefetching schemes to hide the access latency of the second-level BTB. The existing prefetching mechanisms exploit either temporal or spatial correlation between BTB entries to predict future BTB misses. Techniques exploiting temporal correlation [12, 23] form groups of BTB entries that miss consecutively in the BTB and record these groups in a secondary storage. Upon a miss in the predefined code region surrounding the first branch in a group, they prefetch all the entries in a group from the secondary storage into the BTB. Because a branch might exist in multiple paths in the control-flow graph due to various diverges in the control flow (e.g., data dependent if-else construct), it is not guaranteed to co-exist with the same branches in a group every time it is accessed. Such divergences substantially limit the accuracy of the BTB prefetching mechanisms relying on temporal correlation.

Techniques exploiting spatial correlation [11], on the other hand, prefetch all the BTB entries that fall into a contiguous region of code (i.e., 4KB, 64 instruction blocks) upon a BTB miss in that region. Such techniques automatically fail to eliminate the first miss in every region, because the first miss is the prefetch trigger. Moreover, as prior research has shown, because consecutive accesses do not always fall into large contiguous code regions because of divergences (i.e., due to taken branches whose targets are in a different region), there is only limited spatial locality that can be exploited by these BTB prefetchers.

As summarized above, the instruction-supply mechanisms incorporate increasingly aggressive control-flow condition [41, 68], target [11, 12], as well as miss [28, 47] and cache reference [27] information, so-called *predictor metadata*, to improve prediction accuracy, thus performance. Furthermore, the storage requirements to store predictor metadata scale with the instruction working set size of an application. The storage requirements are further exacerbated by

the increasing core counts, as each prediction mechanism is private per core, resulting in abundant metadata redundancy in manycore server processors. The metadata redundancy in the instruction-supply mechanisms is twofold: (i) Inter-core redundancy as the predictor metadata of many cores running the same server workload overlap. (ii) Intra-core redundancy as the predictor metadata for different instruction-supply mechanisms overlap significantly.

1.2 Thesis and Dissertation Goals

Given the potential performance benefits and storage overheads of instruction-supply mechanisms, the next logical step toward higher efficiency is a specialized frontend that minimizes the overhead without sacrificing performance improvements provided by instruction-supply mechanisms. Therefore, the focus of this dissertation is to first identify the sources of redundancy in metadata and then eliminate this redundancy. The statement of this thesis is as follows:

The redundancy in metadata maintained by instruction-supply mechanisms can be eliminated through sharing and unifying metadata across cores and within a core for resource-efficient single-threaded performance improvement in manycore server processors.

In this thesis, we focus on two performance-critical and storage-intensive instruction-supply mechanisms: the instruction streaming mechanism and the branch target buffer (BTB).

1.3 Shared Instruction Streaming Metadata

We observe that individual server workloads are *homogeneous*, meaning that each core executes the same tasks as all other cores. Consequently, over time, all the cores of a processor executing a common homogeneous server workload tend to generate similar instruction streams. Building on this observation, we make a critical insight that commonality and recurrence in the instruction-level behavior across cores can be exploited to generate common temporal instruction streams, which can then be shared by all of the cores running a given

workload.

In this thesis, we explore hardware mechanisms to effectively share temporal instruction stream history across cores running a given homogeneous server workload. We find that a shared instruction stream history generated by a single core can provide almost the same prediction accuracy and performance improvement for all the cores running the same workload by effectively eliminating the majority of instruction cache misses. Moreover, we also describe mechanisms to maintain per-workload temporal instruction stream history in the presence of multiple server workloads consolidated on a CMP.

1.4 Unifying Instruction Streaming with BTB

We observe that in both instruction streaming and BTB prefetching, the required metadata contains a record of the application's control flow history. In the case of the former, the history is at the instruction block granularity; for the latter, it is at the granularity of individual branches. Because of the different granularities at which history is maintained, existing schemes require dedicated histories and prefetchers for both the instruction cache and the BTB.

The temporal knowledge of the control flow at instruction-block granularity in an instruction streaming mechanism incorporates the branches whose BTB entries that will be soon required, obviating the need for a separate BTB prefetcher. If an instruction block is likely to be accessed, the BTB entries of the branches within that block are also very likely to be accessed. Hence, the prediction of an access to a block is a good indicator of the accesses to the BTB entries of branches in that block too.

Accordingly, we propose a lightweight BTB design whose content is maintained in sync with the instruction cache backed by an instruction streaming mechanism. While the temporal instruction streams provided by the instruction streaming mechanism hint the instruction blocks that are likely to be touched, the BTB eagerly installs all the branches (along with their type information and target addresses) within a block upon the insertion of the block into

the instruction cache. By doing so, the BTB exploits the fact that most instructions (including branches) in a block are likely to be touched due to spatial locality during the residency of a block in the instruction cache. By synergistically combining the coarse-grain control-flow information supplied by temporal streams with fine-grain spatial locality in instruction blocks, the BTB maintains only entries within the active instruction working set of an application, while enjoying a high hit ratio just like the instruction cache.

1.5 Contributions

In this thesis, we explore and propose ways to eliminate redundancy in storage-intensive metadata maintained by instruction-supply mechanisms through a specialized frontend design. We begin by quantifying the metadata storage overhead and describing the redundancy problem in the context of manycore server processors running homogeneous server workloads. We then quantify the commonality of metadata maintained by storage-intensive instruction-supply mechanisms demonstrating the opportunity for metadata sharing in manycore server processors. Based on our findings, we propose low-overhead but effective metadata sharing and unifying mechanisms that attain almost the same performance as private per-core instruction-supply mechanisms without incurring their storage overheads.

Through a combination of trace-driven simulation and cycle-accurate modeling of manycore processors running a variety of traditional and emerging server workloads, we demonstrate:

- **Commonality of metadata maintained by instruction-supply mechanisms across cores:** We demonstrate significant commonality in the metadata maintained by per-core instruction-supply mechanisms, the branch predictors and the instruction streaming mechanism, across multiple cores running a common server workload.

- **Design for shared instruction streaming history:** We find that among many cores running a server workload, one core picked at random can generate the instruction history to be leveraged by all cores running the same workload. achieving similar

instruction miss coverage as compared to private history of the same size. Based on this observation, We propose Shared History Instruction Fetch, SHIFT, which maintains an instruction history embedded in the LLC and enables history sharing across all cores. We show that SHIFT provides performance benefits similar to that of private history, despite its low storage overhead that is amortized across many cores. Furthermore, we demonstrate SHIFT's effectiveness in the presence of multiple workloads consolidated on a CMP via per-workload history.

- **Unifying Instruction Streaming with BTB:** We observe the control-flow history represented with coarse-grain temporal streams leveraged for instruction streaming encapsulates the history of fine-grain branch accesses, eliminating the need for a separate BTB prefetcher. In order to effectively leverage the temporal stream history maintained for instruction streaming, we propose a lightweight BTB design whose content mirrors the instruction cache content by maintaining only the BTB entries that are in the active instruction working set (i.e., L1-resident). We show that our lightweight BTB design backed by our shared-history instruction streaming mechanism, SHIFT, provides performance benefits similar to a perfect BTB.

The rest of this thesis is organized as follows. In Chapter 2, we quantify the storage overheads of the state-of-the-art instruction-supply mechanisms and the commonality of their metadata across cores running homogeneous server workloads. In Chapter 3, we propose SHIFT as a hardware mechanism to enable sharing instruction stream history. In Chapter 4, we present Confluence, which maintains a BTB design mirroring the content of the instruction cache and leverages the low-overhead instruction streaming mechanism, SHIFT, to populate the BTB content in parallel with the instruction cache. We discuss related work in Chapter 5 and conclude the thesis with future directions in Chapter 6.

2 Why a Specialized Processor with a Shared Frontend?

In this chapter, we motivate the need for a specialized processor with a shared frontend for traditional and emerging server workloads. First, we describe the common characteristic features of server workloads and explain how the technology trends have been shaping the server design space. Then, we present the storage overheads of instruction-supply mechanisms employed in each core's frontend and explain why they are amenable to sharing. Next, we quantify the sharing opportunities for these storage-intensive instruction-supply mechanisms across many cores. Finally, we touch on the challenges and opportunities associated with a shared frontend design for manycore server processors.

2.1 Server Workloads and Processors

The server workload space has been expanding steadily due to the ever-increasing number of services becoming online. The server software constituting the backbone of these services includes but is not limited to traditional relational databases (e.g., IBM DB2, Oracle, MySQL), modern NoSQL databases (e.g., Cassandra, MongoDB), web servers (e.g., Apache, nginx), in-memory caching systems (e.g., memcached, Redis), streaming servers (e.g., Apple Darwin), and web-search engines (e.g., Apache Nutch). Typical server software relies heavily on third-party libraries and operating system functionalities (e.g., to handle network interrupts, access

the file system). Moreover, server applications are increasingly written in high-level languages to make programming simpler, necessitating interpreters and virtual machines to generate native code at runtime. Such a deep software stack constituting a server workload leads to multi-megabyte instruction working sets [26, 27, 38, 82, 43].

In server workloads, a single client request can be parallelized leveraging multiple threads or processes to minimize response latency (e.g., database systems parallelize the execution of a single analytics query for lower response time as such queries necessitate processing tremendous amounts of data). Furthermore, server workloads inherently exhibit high request-level parallelism as they simultaneously serve a large client base, necessitating use of multiple threads or processes running concurrently. In both cases, intra- and inter-request parallelism, each thread or process executes similar tasks or requests, but only over a small portion of the application data. We call such workloads *homogeneous* workloads.

2.1.1 Common Microarchitectural Characteristics of Server Workloads

Prior work has demonstrated a number of common microarchitectural characteristics distinguishing server workloads from other workload classes such as desktop, engineering, and scientific workloads.

First, the large instruction working sets dwarf the limited capacities of instruction cache and branch predictors, which have stringently low access-latency requirements [2, 26, 63, 82]. Consequently, the frequent misses in the instruction cache and branch predictors introduce one of the major performance bottlenecks in server processors causing frequent misfetches or fetch stalls [2, 4, 11, 12, 35]. Even the mid-level caches in three-level cache hierarchies fall short of accommodating the instruction working sets of server workloads, resulting in frequent accesses to LLC for instruction fetch [26, 38, 43].

Second, server workloads operate on massive datasets that are beyond the reach of limited on-chip last-level caches (LLC). Datasets served by these workloads (i.e., a single server typically accommodates tens of GBs of data in memory) are a few orders of magnitude larger than the

on-chip LLC capacities available in today's processors. Because server applications process many requests in parallel, the temporal locality that can be captured by the LLC is very limited. As a result, even large on-chip LLCs whose capacities are a few tens of MBs can capture only the instruction working set of a server workload and the operating system data, which exhibit high reuse as opposed to application data. Consequently, server workloads frequently access main memory for application data. Moreover, server workloads typically perform irregular memory accesses due to the use of pointer-intensive data structures (e.g., hash tables, B-tree indices, inverted indices). Because of the dependencies between subsequent memory operations, server workloads exhibit poor memory-level parallelism (MLP) (i.e., cannot overlap multiple memory accesses), exposing the core to the full memory access latency [26, 73, 87]. Therefore, server workloads spend most of their execution time waiting for memory.

Due to frequent memory stalls, server workloads are characterized by their low instruction-level parallelism (ILP). Regardless of how large the instruction window of the underlying core is, the core cannot utilize its resources by extracting independent instructions in the instruction window because of long-latency memory instructions and the subsequent instructions that depend on them. As a result, the data-intensive nature of server workloads results in low ILP and MLP, leading to severe underutilization of computation resources.

2.1.2 Specialized Server Processors

Today's mainstream server processors, such as Intel Xeon and AMD Opteron, feature a handful of powerful cores targeting high single-threaded performance for computationally intensive workloads. These processors employ wide-issue and high-frequency out-of-order (OoO) cores with large instruction windows to extract as many independent instructions as possible from the incoming instruction stream and execute independent instructions in parallel to boost performance. However, as summarized in Section 2.1.1, due the low ILP and MLP that can be extracted from server workloads, the core resources remain severely underutilized throughout the execution [21, 26, 53]. This underutilization results in considerable energy inefficiency as all the core resources consume power without being fully and effectively utilized. Furthermore,

the large area footprints of wide-issue OoO cores constrain the number of cores that can be integrated within a given area budget, which in turn limits the amount of parallelism that can be exploited.

The mismatch between the server workload characteristics and general-purpose purpose architectures calls for specialized processors that are tuned for the needs of server workloads to maximize efficiency. Accordingly, recent research has demonstrated that server workloads are best served by manycore processors with cores that are specialized for the relatively low computation needs of server workloads, without sacrificing single-threaded performance [29, 53]. The cores employed in such manycore processors typically have narrower issue widths and employ shallower instruction windows. As a result, they consume significantly less power compared to wide-issue OoO cores. Despite their low power consumption, such manycore processors deliver competitive performance when executing server workloads compared to wide-issue OoO cores as server workloads exhibit low ILP. Furthermore, smaller area footprints of these specialized cores allow for a higher core count within a given area budget, further boosting the throughput by exploiting the request-level parallelism prevalent in server workloads. A number of processors in today's server space exemplify the manycore server processors architected for server workloads. These include Cavium ThunderX [17], Applied Micro's XGene [6], and EZchip's Tile-MX series [30], all featuring tens of ARM Cortex cores.

A number of techniques further specialize for server workloads to provide low-latency instruction delivery from the LLC to the cores. Cavium ThunderX series employ a flat two-level on-chip cache hierarchy to minimize the access latency to LLC-resident instructions, as suggested by recent research results [26, 53], as opposed to deeper three-level cache hierarchies in Intel Xeon and AMD Opteron. By doing so, Cavium ThurderX eliminates the latency associated with accessing a mid-level cache in three-level hierarchies, which cannot accommodate the instruction working sets of server workloads and only adds extra latency to frequent LLC accesses for instructions. To further mitigate the inefficiencies in the cache hierarchy, scale-out processors [53] shrinks the LLC to the extent that it can accommodate only the shared

instruction working set, which in turn reduces the access latency to instructions and eliminates the silicon waste stemming from low data reuse. NoC-Out [52] proposes a specialized on-chip interconnect for server workloads that do not exhibit inter-core communication by eliminating inter-core links, thus the unnecessary area overhead associated with them, and organizing the cores around the small LLC to minimize the latency of accesses to instructions.

While these approaches propose specialization for high-performance instruction delivery for server workloads, they do not address the inefficiencies associated with the instruction-supply mechanisms employed in the frontend of each core in a manycore server processor for high single-threaded performance. As we show in the rest of this section, further improving the efficiency in server processors mandates specializing the server processors to address remaining inefficiencies in the instruction-supply path.

2.2 Frontend Metadata Overhead and Commonality

As explained in Section 2.1.1, server workloads are characterized by their large instruction working sets. These large instruction working sets defy the capacities of instruction-supply mechanisms of a core, namely the branch predictors and the instruction prefetcher, whose functionality is to provide the core with a continuous and useful instruction stream to execute. The instruction-supply mechanisms exploit the recurring control flow in applications to predict future control flow, thus the upcoming instruction stream. To do so, they collect and record application metadata that is used in identifying certain patterns during an application's execution. Upon recurrence of a particular pattern, the metadata recorded is leveraged to predict future patterns of the application. Higher prediction accuracy necessitates maintaining metadata for the entire instruction working set of an application to capture all the possible recurring patterns. Therefore, the required metadata storage capacity scales with the instruction working set size of an application.

To make matters worse, each core maintains its own private metadata storage, as the instruction-supply mechanisms are latency-sensitive. As a result, the aggregate area footprint

OLTP – Online Transaction Processing (TPC-C)	
DB2	*IBM DB2 v8 ESE Database Server* 100 warehouses (10GB), 64 clients, 2GB buffer pool
Oracle	*Oracle 10g Enterprise Database Server* 100 warehouses (10GB), 16 clients, 1.4 GB SGA
DSS – Decision Support Systems (TPC-H)	
Qry 2, 8, 17, 20	*IBM DB2 v8 ESE* 480MB buffer pool, 1GB database
Media Streaming	
Darwin	*Darwin Streaming Server 6.0.3* 7500 clients, 60GB dataset, high bitrates
Web Frontend (SPECweb99)	
Apache	*Apache HTTP Server v2.0* 16K connections, fastCGI, worker threading model
Web Search	
Nutch	*Nutch 1.2/Lucene 3.0.1,* 230 clients, 1.4 GB index, 15 GB data segment

Table 2.1: Server workload parameters.

dedicated to metadata storage scales with the number of cores in manycore processors. However, in a manycore server processor where each core executes a given homogeneous server workload, the metadata privately maintained by each core overlaps significantly as each core exhibits the same control flow, leading to significant waste of real-estate due to redundancy.

In homogeneous workloads, every thread and/or process executes the same code as all the other threads/processes of the workload. Server applications that rely on multithreading leverage a single application binary that is loaded into the memory and all threads within a process share common virtual-to-physical address translations. In applications leveraging multiple processes, new processes are often launched through forking, which maintains the same virtual-to-physical address mappings of the parent process for the children processes until there is a write operation performed by parent or children processes, upon which a new page is created through a copy-on-write operation. The commonality of addresses across cores running a given homogeneous server workload is the key enabler for sharing the metadata of instruction-supply mechanisms across cores as explained in the rest of this section.

In the rest of this section, we first quantify the overheads of storage-intensive instruction-

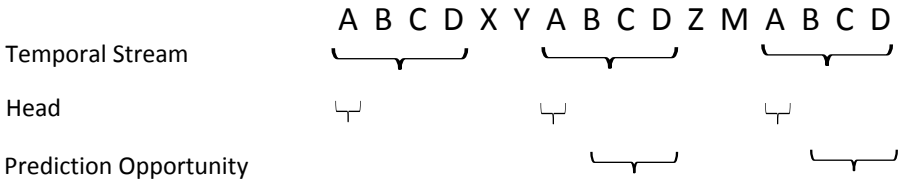

Figure 2.1: Temporal instruction stream example.

supply mechanisms for a suite of traditional and emerging homogeneous server workloads listed in Table 2.1. For each mechanism, we also demonstrate the metadata commonality across cores running a homogeneous server workload by maintaining a shared metadata storage across cores and assuming a hypothetical server system, in which the cores can access shared metadata without any latency penalties.

2.2.1 Instruction Streaming

Instruction prefetching is an established approach to alleviate instruction-fetch stalls prevalent in server workloads stemming from frequent instruction misses. Next-line prefetcher, a common design choice in today's processors, can eliminate only around 30% of instruction cache misses on average, for the workload suite listed in Table 2.1. Given the remaining performance potential from eliminating more instruction cache misses, server processors call for more sophisticated instruction prefetchers.

To overcome the next-line prefetcher's inability to predict instruction cache misses that are not to contiguous instruction blocks, instruction streaming mechanisms [27, 28] exploit the recurring control-flow graph traversals in server workloads. As program control flow recurs, the core frontend generates repeating sequences of fetch addresses. The instruction fetch addresses that appear together and in the same order are temporally correlated and together form a so-called temporal stream. For instance, as illustrated in Figure 2.1, in the instruction cache access sequence *A, B, C, D, X, Y, A, B, C, D, Z, M, A, B, C, D*, the address sequence *A, B, C, D* constitutes a temporal instruction stream. Once a temporal stream is recorded, it can be identified by its first address, the stream head (address *A* in our example) and replayed to issue

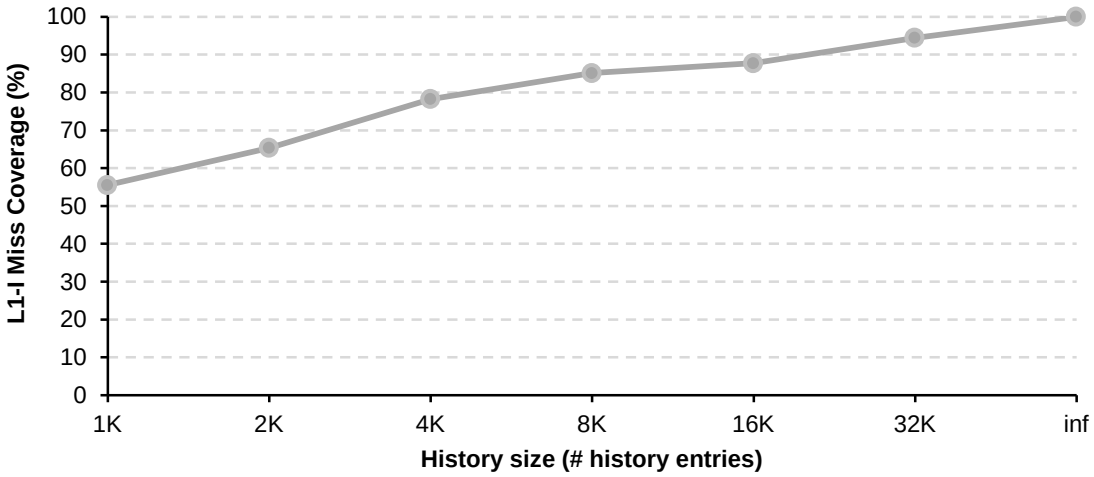

Figure 2.2: PIF's miss coverage as a function of dedicated history size.

prefetch requests in advance of the core frontend to hide the instruction cache miss latency from the core. If the actual instruction stream matches the replayed stream (addresses *B, C, D*), the instruction misses are eliminated by the streaming mechanism.

The state-of-the-art instruction streaming mechanism is Proactive Instruction Fetch (PIF) [27], which extends earlier work on temporal streaming [28]. PIF's key innovation over prior work is its reliance on access streams instead of miss streams. By recording all accesses to the instruction cache, as opposed to just those that miss, PIF eliminates the dependence on the content of the cache. While cache content can change over time, reference patterns that PIF records remain stable. To mitigate the storage overhead associated with recording all instruction cache access addresses, PIF exploits the temporal and spatial locality in instruction access stream to maintain a compact representation of the addresses of the instruction blocks that are in close proximity in the address space and accessed closely in time. Despite the stream compaction optimization, PIF incurs significant storage overhead for high instruction miss coverage (i.e., fraction of misses eliminated). Figure 2.2 illustrates how the miss coverage increases with the history size. For instance, to eliminate an average of 90% of instruction cache misses, thus approaching the performance of a perfect I-cache, PIF requires 32K entries in history, corresponding to over 210KB of history storage per core (Section 3.4 details PIF's storage cost).

Relative Area Overhead: While the performance benefits of instruction streaming are consistently high across the core microarchitecture spectrum, the relative cost varies greatly. For instance, the 210KB of storage required by the PIF prefetcher described above consumes $0.9mm^2$ of the die real-estate in 40nm process technology. Meanwhile, a Xeon Nehalem core along with its private L1 caches has an area footprint of $25mm^2$. The 4% of area overhead that PIF introduces when coupled to a Xeon core is a relative bargain next to the 23% performance gain it delivers. This makes PIF a good design choice for conventional server processors. (Section 3.4 details the system configurations used for these experiments)

On the opposite end of the microarchitectural spectrum is a lean core like the ARM Cortex-A8 [9]. The A8 is a dual-issue in-order design with an area of $1.3mm^2$, including the private L1 caches. In comparison to the A8, PIF's $0.9mm^2$ storage overhead is prohibitive given the 17% performance boost that it delivers. Given that server workloads have abundant request-level parallelism, making it easy to scale performance with core count, the area occupied by PIF is better spent on another A8 core. The extra core can double the performance over the baseline core, whereas PIF only offers a 17% performance benefit.

To succinctly capture the performance-area trade-off, we use the metric of performance-density (PD), defined as performance per unit area [53]. By simply adding cores, server processors can effectively scale performance, while maintaining a constant PD (i.e., twice the performance in twice the area). As a result, the desirable microarchitectural features are those that grow performance-density, as they offer a better-than-linear return on the area investment.

Figure 2.3 shows the relative performance-density merits of three PIF-enabled core microarchitectures over their baseline (without PIF) counterparts. Two of the cores are the Xeon and ARM Cortex-A8 discussed above; the third is an ARM Cortex-A15 [84], an out-of-order core with an area of $4.5mm^2$.

In the figure, the trend line indicates constant PD (PD = 1), which corresponds to scaling performance by adding cores. The shaded area that falls into the left-hand side of the trend

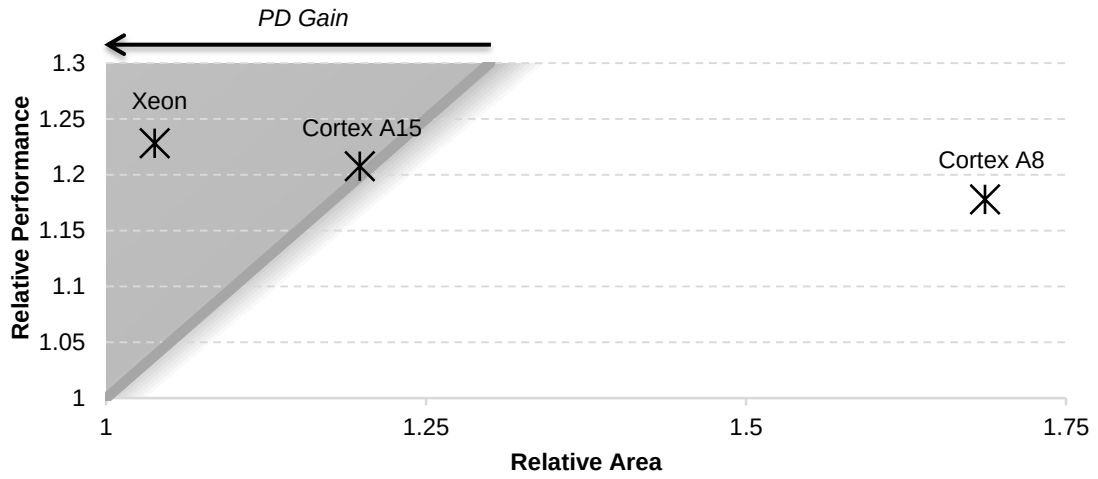

Figure 2.3: Comparison of PIF area overhead and performance improvement for various core types.

line is the region of PD gain, where the relative performance improvement is greater than the relative area overhead. The right-hand side of the trend line corresponds to PD loss, where the relative performance gain is less than the relative area. In summary, designs that fall in the shaded region improve performance-density over the baseline; those outside strictly diminish it.

The results in the figure match the intuition. For the Xeon core, PIF improves performance-density. In contrast, for the Cortex-A15 core, PIF fails to improve PD, and for the Cortex-A8 core, PIF actually diminishes PD, providing less-than-linear performance density benefit that cannot compensate for its area overhead. Thus, we conclude that processors with specialized lean cores (i.e., ARM Cortex cores) benefit from instruction streaming nearly as much as designs with fat wide-issue cores (i.e., Intel Xeon), but mandate area-efficient mechanisms to minimize the overhead.

Temporal Instruction Stream Commonality: To quantify the similarity between the instruction streams of cores running a homogeneous server workload, we pick a random core to record its instruction cache access stream. All the other cores, upon referencing the first address in a stream, replay the most recent occurrence of that stream in the recorded history. The commonality between two streams is quantified as the number of matching instruction block

addresses between the replayed stream and those issued by the core replaying the stream. For this study, we use instruction fetch traces from 16 cores and average the results across all of the cores.

Workload	% I-cache accesses
OLTP DB2	91
OLTP Oracle	92
DSS Qry2	94
DSS Qry8	92
DSS Qry17	91
DSS Qry20	93
Media Streaming	96
Web Frontend	90
Web Search	95

Table 2.2: Instruction cache accesses within common temporal streams.

Table 2.2 shows the commonality of instruction streams between cores executing a given homogeneous server workload. More than 90% (up to 96%) of instruction cache accesses (comprised of both application and operating system instructions) from all sixteen cores belong to temporal streams that are recorded by a single core picked at random. This result indicates that program control flow commonality between cores yields temporal instruction stream commonality, suggesting that multiple cores running a homogeneous server workload can benefit from a single shared instruction history for instruction streaming.

2.2.2 Branch Predictors

Branch prediction is a vital component for high single-threaded performance. Branch instructions constitute a special class of instructions as they can divert the control flow of a program from sequential flow to another location in the program. In the absence of a branch predictor, upon an incoming branch instruction, the core would stall until the execution of the branch instruction to resolve the branch (i.e., determine if it is taken or not) and calculate the target instruction address, if the branch is taken, to redirect the control flow to the target address. As a result, each branch instruction would stall the instruction fetch unit of the core until the branch gets resolved and the next instruction address is calculated by the processor, creating

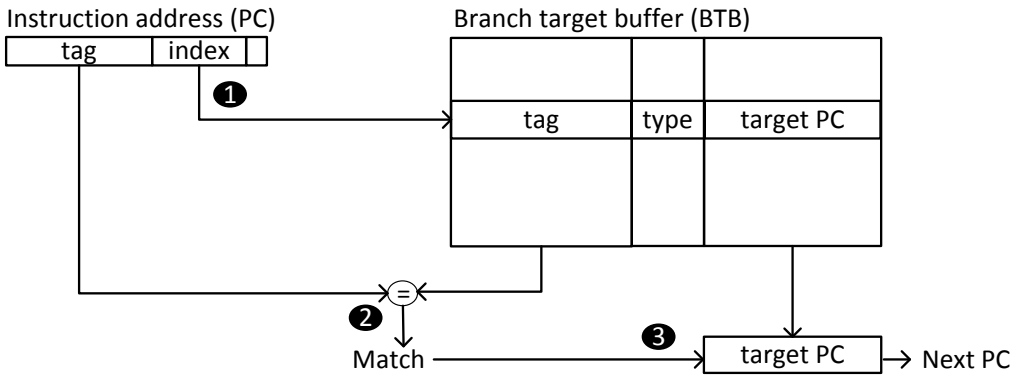

Figure 2.4: Target address prediction of a branch instruction in BTB.

bubbles in the pipeline. These pipeline bubbles would essentially lead to lower performance.

To mitigate the pipeline bubbles due to branch instructions, today's processors rely on aggressive branch predictors. A branch predictor consists of two orthogonal components: 1) branch direction predictor and 2) branch target predictor. The branch direction predictor predicts whether a branch will be taken or not. If a branch is predicted not taken, the instruction fetch stream continues sequentially with the next instruction in the program. However, if the branch is predicted taken, its target instruction address, from which instruction the fetch stream will continue, needs to be predicted as well.

Branch Target Prediction

The chief mechanism employed to predict the target instruction addresses of taken branches is the Branch Target Buffer (BTB) [79]. Each BTB entry stores the most recent target instruction address for a particular branch instruction and uses the stored target instruction address as the prediction next time that branch instruction is encountered. In doing so, the BTB exploits the fact that, for the majority of branches (i.e., relative branches) the target instruction address is stable throughout the program execution. [1] BTB, as shown in Figure 2.4, is typically a set-associative table, where each entry is tagged with a branch instruction's program counter

[1] Return instructions and indirect branches may have several different target addresses. Because the BTB is not very effective for these types branches as it only stores one target address per branch instruction, today's processors employ separate prediction mechanism to predict targets of return instructions and indirect branches.

(PC). Each BTB entry stores the most recent target instruction address (target PC) encountered for a particular branch instruction and optionally a few status bits to indicate the type of the branch (i.e., conditional, unconditional, return).

The BTB needs to be both accurate and timely meaning that it needs to provide the correct branch target address within a cycle to constantly feed the processor pipeline with correct instructions. The single-cycle latency constraint for the BTB can be relaxed in the presence of a branch direction predictor with an access latency higher than one cycle, as retrieving the target address of the branch instruction would be of no use before the direction of the branch is known. In this work, we target a system with a branch direction predictor latency of only single cycle. To achieve maximum hit rate, thus accuracy, the BTB needs to maintain entries of all the taken branches in the instruction working set of an application. As a result, the capacity requirements of the BTB scale with the instruction working set of an application. However, larger BTBs exhibit higher access latencies, resulting in high-latency target address predictions. Consequently, delayed target address predictions offset the performance benefits of accurate predictions as they cause bubbles in the pipeline.

Superscalar cores necessitate the branch prediction unit to provide multiple instructions every cycle depending on the fetch width of the core. To provide multiple instructions to be fetched with a single BTB lookup, the BTB can be organized to provide a basic block range with a single lookup [64, 91]. In this case, each BTB entry is tagged with the starting address of a basic block and maintains the target address of the branch ending the basic block, the fall-through address (the next instruction after the branch ending the basic block) and the type of the branch instruction ending the basic block. Every cycle, a direction prediction is made for the branch instruction terminating the current basic block (i.e., the instruction before the fall-through instruction). Based on the outcome of the direction prediction, the fall-through address or the target instruction address is used to perform a lookup in the BTB for the next basic block. In the rest of the paper, we leverage such a BTB as the baseline BTB design. Whenever a basic block (hence the branch ending the basic block) is not found in the BTB, a basic block with a predetermined number of sequential instructions is passed to the

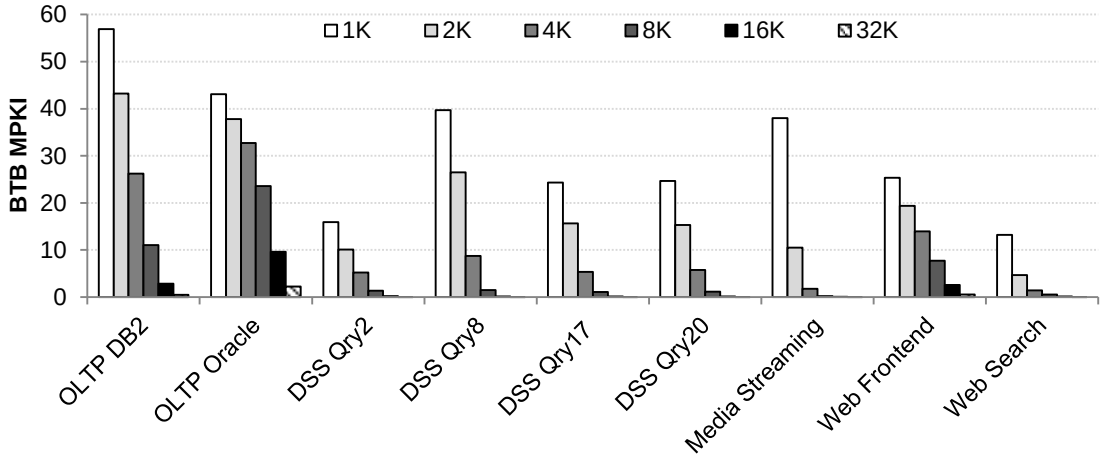

Figure 2.5: BTB MPKI as a function of BTB capacity (in K entries).

instruction fetch unit to be fetched.

Figure 2.5 shows the BTB MPKI as a function of the total number of BTB entries per core. A BTB miss corresponds to a branch instruction (or the basic block that ends with the branch instruction in the basic-block-based design) whose entry is not present in the BTB and hence is not considered as a branch at prediction time. As a result, BTB misses result in misfetches (i.e., wrong target predictions) if the corresponding branch instruction is taken. The server workloads require up to 16K BTB entries to fully capture the taken branches in the instruction working sets of the workloads, while OLTP on Oracle benefits from even 32K entries, corroborating prior work [11, 12, 35]. The storage capacity requirement of a 32K-entry BTBs is around 290KB (Section 4.3 details the storage cost).

Relative Area Overhead: A 16K-entry BTB (~150KB) in 40nm technology occupies around $0.6mm^2$ area. This area footprint corresponds to 46% of the Cortex-A8, 13% of the Cortex-A15, and only 2.5% of the Xeon area footprint. As the relative area overheads indicate, large BTBs incur a much higher area overhead for the Cortex cores as compared to the Xeon core.

BTB Metadata Commonality: Table 2.3 demonstrates the BTB miss rate when 16 cores running a given server workload leverage a shared 32K-entry BTB. The insignificant increase in BTB MPKI (0.2 in the worst case) in the shared BTB validates our expectations of the commonality

Workload	Private	Shared
OLTP DB2	0.50	0.55
OLTP Oracle	2.27	2.40
DSS Qry2	0.07	0.03
DSS Qry8	0.02	0.01
DSS Qry17	0.05	0.02
DSS Qry20	0.06	0.03
Media Streaming	0.02	0.02
Web Frontend	0.54	0.70
Web Search	0.03	0.04

Table 2.3: BTB MPKI when 16 cores share a 32K-entry BTB compared to private 32K-entry BTB.

of BTB entries between cores running a homogeneous server workload. The subtle increase in the BTB miss rate can be attributed to cold code paths exercised by a random core due to infrequent events such as OS scheduler activity.

To conclude, although the BTB storage requirements for server workloads are as big as 290KB, the commonality of control flow across cores paves the way for sharing the BTB storage, thus eliminating the redundancy and amortizing its storage overhead across many cores.

Branch Direction Prediction

The key enabler for the high accuracy provided by today's state-of-the-art branch predictors is mainly correlating a branch's outcome with the outcome of *many* previously executed branch instructions in the dynamic instruction stream [39, 41, 68]. While increasing the prediction accuracy for some branch instructions necessitates maintaining longer branch histories (i.e., hundreds of dynamic branch outcomes), most branches can be predicted accurately by leveraging relatively short histories. Maintaining a fixed-size history for all branches results in an unnecessary explosion in the required predictor size. Instead, only the branches that benefit from longer histories should maintain predictions correlated with long branch histories, while other branches should maintain predictions correlated with shorter branch histories in the predictor table. Furthermore, achieving high accuracy necessitates minimizing aliasing in the predictor tables, preventing two or more different branch patterns

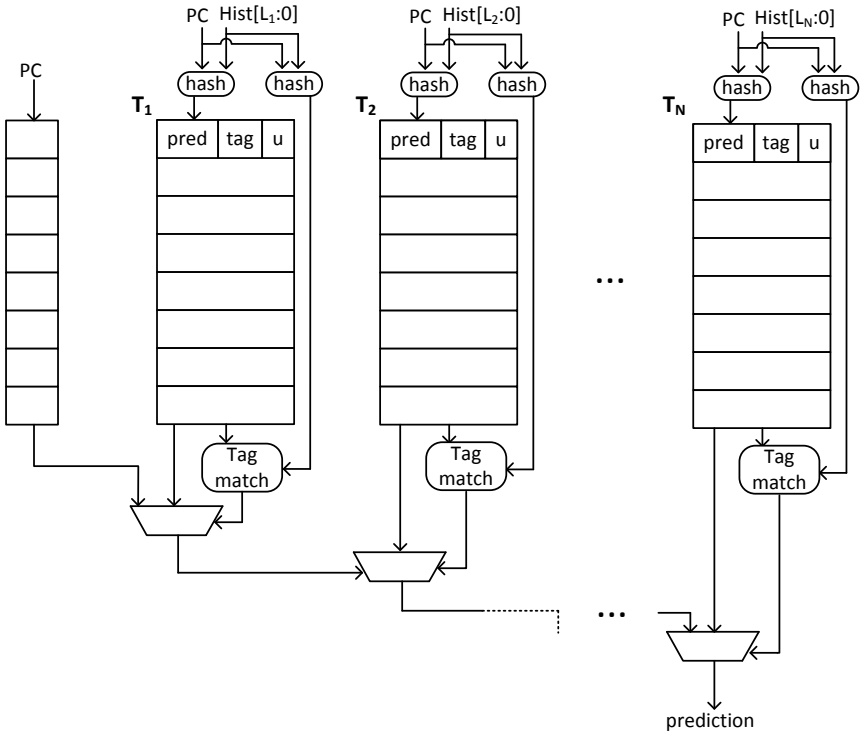

Figure 2.6: TAGE branch predictor with a base bimodal predictor and N tagged tables.

from reading or updating the same entry in the predictor.

Given these requirements for high branch prediction accuracy, the TAGE predictor [68] shown in Figure 2.6, employs a base bimodal predictor and a set of partially tagged predictor tables, each of which maintains predictions for branch outcome patterns with different lengths. The global history lengths used by the tagged tables all together form a geometric series. The base bimodal predictor is indexed by a branch PC and provides a prediction if there is no matching entry in the tagged tables for a given branch history at the time of the prediction. In addition to a partial tag, each tagged table entry contains a 3-bit saturating counter for direction prediction and a 2-bit useful counter. Each tagged table is indexed through a unique function of the current branch PC, global history and path history. Similarly, the partial tags are formed through a unique function of the branch PC and global history.

At prediction time, all tables are accessed in parallel. The predictions from two tables with the longest histories are considered for the actual prediction upon a tag match. Typically, the

prediction from the table associated with a longer history is actually used unless the entry is considered as recently inserted. If the prediction is wrong, a new entry is allocated in a table associated with a longer history than the table that provided the prediction. This way, the branches that cannot be correctly predicted by short histories are promoted to tables with longer histories.

Overall, through the combination of entry tagging and use of variable history lengths in several disparate tables, TAGE outperforms all the other branch predictors in the literature. However, the use of tags in the predictor tables and the need for capturing all the recurring branch patterns in a workload for high accuracy inflate the storage capacity requirements of the TAGE predictor.

Configuration	Tage_1	Tage_2	Tage_3	Tage_4	Tage_5	Tage_6
Scaling Factor	1/16	1/8	1/4	1/2	1	2
Aggregate Storage (KB)	4	6	9	16	30	58

Table 2.4: Total storage capacities required for various TAGE configurations.

We evaluate TAGE's accuracy by varying its per-core aggregate storage capacity for the server workloads listed in Table 2.1. We start with the ~30KB TAGE configuration [68], which achieved the highest accuracy in the first Branch Prediction Championship (CBP-1) and set the basis for the predictors that won the subsequent competitions. For the configurations with different aggregate capacities, we keep the size of the bimodal predictor constant as it is a tagless table and requires only 2.5KB storage capacity in total and scale the number of entries in the tagged tables by powers of two for each different configuration. The resulting aggregate table sizes for various configurations evaluated are listed in Table 2.4. We also evaluate the accuracy of the base bimodal predictor to better understand the contribution of tagged tables to higher accuracy.

Figure 2.7 shows the misprediction rates for the base bimodal predictor and the various TAGE configurations listed in Table 2.4. As expected, as the number of entries in tagged tables increases, TAGE achieves higher accuracy because it can accommodate more entries belonging to different branch outcome patterns and different branches. The reduction in the MPKI with

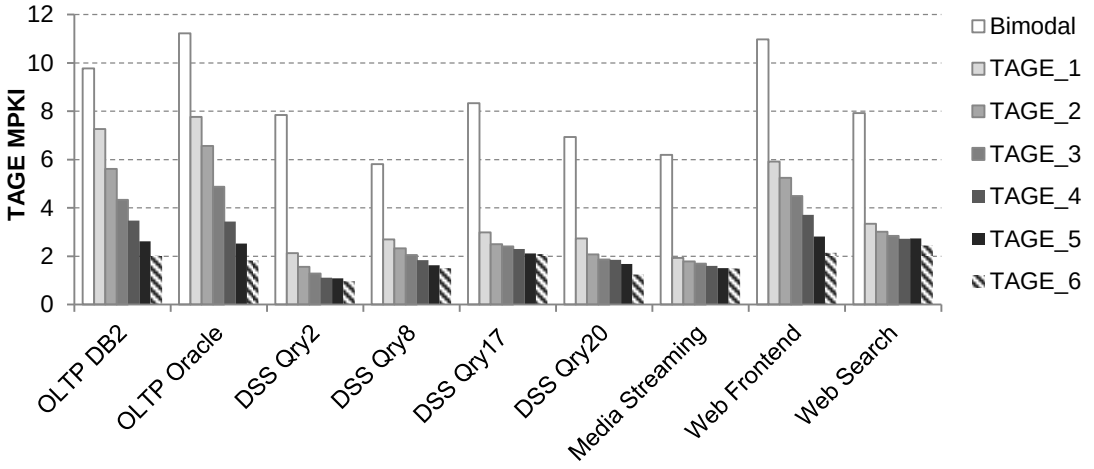

Figure 2.7: TAGE mispredictions per kilo instructions (MPKI) for various TAGE configurations.

the increasing aggregate TAGE capacity is more pronounced for the OLTP workloads and web frontend as these workloads have larger instruction working sets. In conclusion, we see that server workloads benefit from a TAGE predictor as big as ~60KB per core for higher accuracy.

Relative Area Overhead: We estimate the area of a single-ported 60KB TAGE to be around $0.25mm^2$ in 40nm technology using CACTI (This is a very rough estimate since CACTI does not accept block sizes, which correspond to entry sizes in TAGE, in bits). TAGE's area corresponds to 20% of Cortex-A8, 6% of Cortex-A15 and only 1% of a Xeon core's area. Because we do not have enough details about the branch predictors employed in these cores, their sizes and contribution to the aggregate core area, we are not able to perform a performance density analysis for TAGE. However, like the instruction-streaming mechanism and BTB, the relative area overhead induced by TAGE per core is higher for the Cortex cores as compared to the Xeon core, as expected.

TAGE Metadata Commonality: To quantify the commonality of branch history patterns across cores running a common server workload, we employ a single TAGE predictor with approximately 60KB of aggregate storage (Tage_6 in Table 2.4) shared across 16 cores. In our shared TAGE configuration, the base bimodal predictor is per-core as it incurs insignificant storage overhead (2.5KB), but all the tagged tables are shared across cores. Figure 2.8 demonstrates the miss rates for 60KB per-core TAGE and 60KB shared TAGE, averaged across all cores. Lower

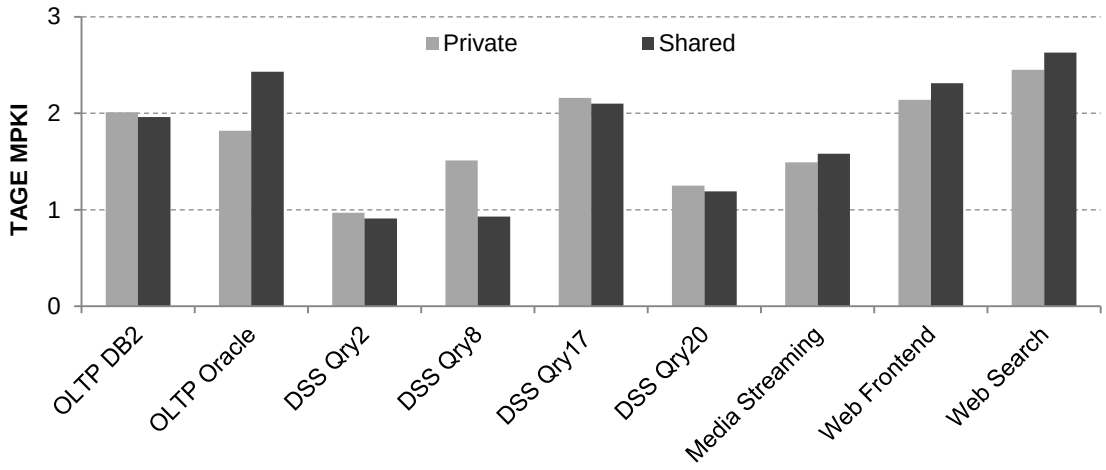

Figure 2.8: TAGE misprediction rate when a 60KB TAGE table is shared across 16 cores compared to per-core 60KB TAGE.

miss rates for the shared TAGE in the case of OLTP on DB2 and DSS queries indicate constructive sharing, meaning a prediction entry inserted or updated by one core is useful for other cores. For the rest of the workloads, the slight increase in the miss rate (0.6 in the worst case) might be due to the capacity limitations or the interference between cores when actively using the same entries.

We conclude that, although the TAGE storage requirements for server workloads are as big as 60KB per core, the commonality of control flow across cores makes the TAGE storage amenable to sharing to eliminate the redundancy and amortize its storage overhead across many cores.

2.3 Exploiting Metadata Commonality for Shared Frontend

As explained in Section 2.2, the instruction-supply mechanisms employed in each core's frontend are storage-intensive, but amenable to sharing in the context of manycore processors executing homogeneous server workloads due to the significant overlap of control flow across cores. In this thesis, our goal is to build a manycore server processor with a shared frontend and enable all the cores running a given server workload to leverage a shared single metadata storage, hence eliminate the replication of metadata redundantly across cores, without losing the single-threaded performance benefits attained by instruction-supply mechanisms

maintaining private metadata.

The primary challenge associated with sharing the metadata of the instruction-supply mechanisms is their latency sensitivity. Branch predictors are required to provide one or more predictions every cycle, while the instruction streaming lookahead must be large enough to prefetch the instructions well ahead of the core frontend's demand to hide the access latency of the lower levels of the memory hierarchy. A large metadata storage naively shared across cores would expose the core frontend to on-chip interconnect latencies and high access latencies of large predictor tables, leading to marginal or no performance benefits as the predictors would not be able to provide timely predictions.

Providing timely predictions leveraging shared frontend metadata necessitates predicting the metadata that will be required in near future and fetch that metadata into a small and private predictor table next to the core. This essentially means organizing the metadata in hierarchical tables, just like cache hierarchies in today's processors, and prefetching metadata from secondary storage into primary storage as needed. Provided the metadata can be prefetched in timely manner, leveraging the private small metadata storage to make predictions minimizes the latency the core is exposed to, while the shared metadata storage eliminates redundancy.

2.4 Summary

In this chapter, we described the common characteristics of server workloads and the current processor specialization trends to achieve high efficiency for these workloads. We then identified the metadata redundancy as another key source of inefficiency in manycore server processors and quantified the per-core storage overheads of storage-intensive instruction-supply mechanisms. We demonstrated the commonality of frontend metadata across cores running homogeneous server workloads to pave the way for a shared-frontend server processor. Finally, we described the requirements of a shared frontend that can provide similar performance benefits to private instruction-supply mechanisms, without incurring their storage overheads.

3 SHIFT: A Design for Sharing Instruction Streaming Metadata

Prefetching is a well-known mechanism to overcome the instruction-fetch stall bottleneck stemming from frequent instruction cache misses. Prior work has shown that instruction-fetch stalls account for up to 40% of execution time in server processors [33, 63, 76, 82]. Given the performance potential of fully eliminating instruction-fetch stalls, prior work has proposed increasingly aggressive instruction prefetchers through the use of more explicit instruction reference history [27, 28, 47]. In the state-of-the-art instruction prefetcher, PIF [27], the history consists of a continuous stream of instruction cache reference addresses that are not filtered by caches, thus not affected by cache content at a given point in time or the cache replacement policy. As a result, PIF yields phenomenally high accuracy and cache miss coverage provided enough capacity to capture the history of the entire instruction working set of an application. Unfortunately, such high accuracies are achieved at the cost of high per-core storage overhead to maintain the history of the complete instruction reference streams. Furthermore, in manycore server processors executing homogeneous server workloads, the significantly overlapping streams of instruction references are replicated redundantly in the private history tables of individual cores, leading to considerable waste of silicon.

We make a critical insight that commonality and recurrence in the instruction-level behavior across cores can be exploited to generate a common instruction history, which can then be shared by all of the cores running a given workload. By sharing the instruction history and its

associated storage among multiple cores, this work provides an effective approach for mitigating the severe area overhead of existing instruction streaming designs, while preserving their performance benefit. As a result, we propose a practical instruction streaming mechanism to mitigate the frontend stalls resulting from instruction cache misses in manycore server processors.

The contributions of this chapter are as follows:

- We introduce Shared History Instruction Fetch, SHIFT, a new instruction streaming design, which combines shared instruction history with lightweight per-core control logic. By sharing the history, SHIFT virtually eliminates the high history storage cost associated with earlier approaches [27].
- We show that a single core chosen at random can generate the instruction history, which can then be shared across other cores running the same workload. In a 16-core CMP, the shared history eliminates 85%, on average, of all instruction cache misses across a variety of traditional and emerging server workloads.
- SHIFT yields an absolute performance improvement of 19% on a suite of diverse server workloads, on average, capturing 90% of the performance benefit of the state-of-the-art instruction streaming mechanism [27] at 14x less storage cost in a 16-core CMP.
- By embedding the history in the memory hierarchy, SHIFT eliminates the need for dedicated storage and provides the flexibility needed to support consolidated workloads via a per-workload history.

The rest of this chapter is organized as follows. We first describe the SHIFT design with dedicated storage in Section 3.1. In Section 3.2, we describe how we can embed the SHIFT history into the LLC. In Section 3.3, we explain how SHIFT can be used by multiple workloads in the case of consolidation. Section 3.4 details the methodology used for the evaluation. Then, we study SHIFT's sensitivity to the design parameters and evaluate SHIFT against prior

proposals in Section 3.5. We discuss additional issues in Section 3.6. Finally, we conclude in Section 3.7.

3.1 SHIFT with Dedicated History Storage

SHIFT exploits the commonality of instruction fetch streams across cores running a common homogeneous server workload by enabling a single instruction fetch stream history to be shared by all the cores. We base the SHIFT history storage on the Global History Buffer [57] prefetcher to record instruction fetch streams in a similar vein to prior instruction streaming mechanisms [27, 28, 47] as described in Section 2.2.1.

We augment each core with simple logic to read instruction streams from the shared history buffer and issue prefetch requests. In this section, we present the baseline SHIFT design with dedicated storage, and then explain how to virtualize the storage (i.e., embed the storage in the LLC) in the next section. Finally, we demonstrate SHIFT with multiple history buffers to enable support for workload consolidation.

SHIFT employs two microarchitectural components shared by all the cores running a common workload to record and replay the common instruction streams: the history buffer and the index table. The history buffer records the history of instruction streams; the index table provides fast lookups for the records stored in the history buffer.

The per-core private stream address buffer reads instruction streams from the shared history buffer and coordinates prefetch requests in accordance with instruction cache misses.

Recording. SHIFT's distinguishing feature is to maintain a single shared history buffer and employ only one core, history generator core, running the target workload to generate the instruction fetch stream history.

The history generator core records retire-order instruction cache access streams to eliminate the microarchitectural noise in streams introduced by the instruction cache replacement policy and branch mispredictions [27]. To mitigate increased history storage requirements

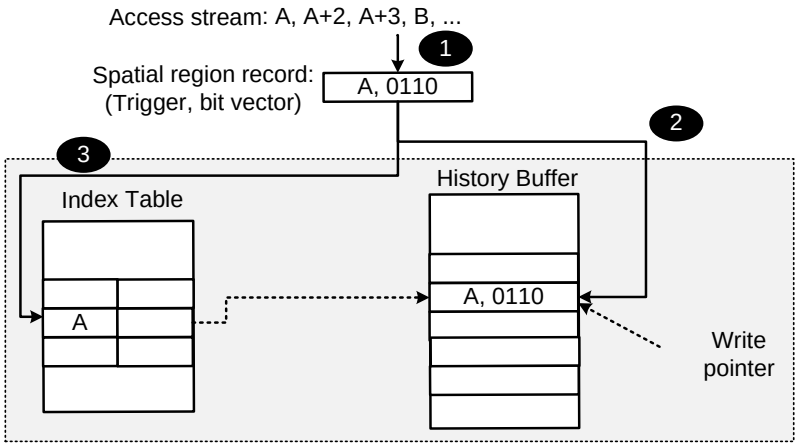

Figure 3.1: SHIFT's logical components and data flow to record temporal instruction streams.

resulting from recording instruction cache accesses rather than instruction cache misses, the history generator core collapses retired instruction addresses by forming spatial regions of instruction cache blocks.

Step 1 in Figure 3.1 depicts how a spatial region is generated by recording retire-order instruction cache accesses obtained from the history generator core's backend. In this example, a spatial region consists of five consecutive instruction cache blocks; the trigger block and four adjacent blocks. The first access to the spatial region, an instruction within block A, is the trigger access and defines the new spatial region composed of the instruction blocks between block A and A+4. Subsequent accesses to the same spatial region are recorded by setting the corresponding bits in the bit vector until an access to a block outside the current region occurs.

Upon an access to a new spatial region, the old spatial region record is sent to the shared history buffer to be recorded. The history buffer, logically organized as a circular buffer, maintains the stream of retired instructions as a queue of spatial region records. A new spatial region is recorded in the shared history buffer into the entry pointed by the write pointer, as illustrated in step 2 in Figure 3.1. The write pointer is incremented by one after every history write operation and wraps around when it reaches the end of the history buffer.

To enable fast lookup for the most recent occurrence of a trigger address, SHIFT employs an

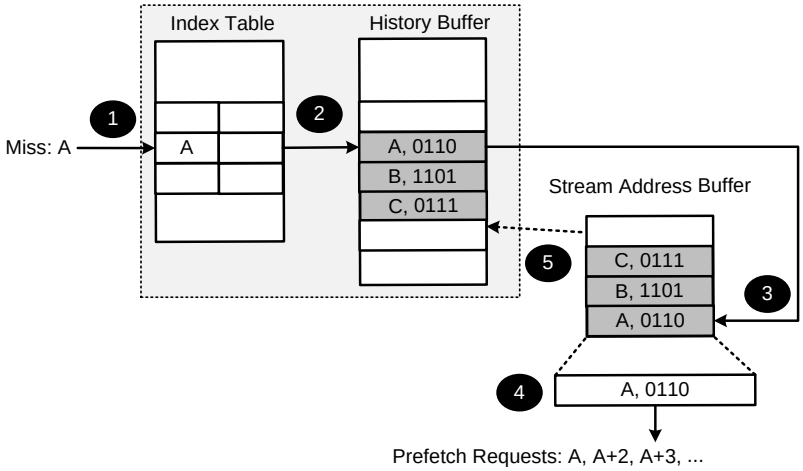

Figure 3.2: SHIFT's logical components and data flow to replay temporal instruction streams.

index table for the shared history buffer, where each entry is tagged with a trigger instruction block address and stores a pointer to that block's most recent occurrence in the history buffer. Whenever a new spatial region record is inserted into the history buffer, SHIFT modifies the index table entry for the trigger address of the new record to point to the insertion position (step 3 in Figure 3.1). SHIFT does not invalidate the index pointer of the entry that is overwritten in the history buffer in order not to generate extra traffic to the index table.

Replaying. The per-core stream address buffer maintains a queue of spatial region records and is responsible for reading a small portion of the instruction stream history from the shared history buffer in anticipation of future instruction cache misses. When an instruction block is not found in the instruction cache, the stream address buffer issues an index lookup for the instruction's block address to the index table (step 1 in Figure 3.2). If a matching entry is found, it supplies the pointer to the most recent occurrence of the address in the history buffer. Once the spatial region corresponding to the instruction block that triggered the lookup is located in the history buffer (step 2 in Figure 3.2), the stream address buffer reads out the record and a number of consecutive records following it as a lookahead optimization. The records are then placed into the stream address buffer (step 3 in Figure 3.2). If the missing instruction's block address and the address read from the history buffer do not match (indicating a stale pointer), the stream is discarded and nothing is allocated in the stream address buffer.

Next, the stream address buffer reconstructs the instruction block addresses encoded by the spatial region entries based on the trigger address and the bit vector. Then, the stream address buffer issues prefetch requests for the reconstructed instruction block addresses if they do not exist in the instruction cache (step 4 in Figure 3.2).

The stream address buffer also monitors the retired instructions. A retired instruction that falls into a spatial region maintained by the stream address buffer advances the stream by triggering additional spatial region record reads from the history buffer starting from the location pointed by the stream address buffer (points to the last spatial region entry read for that stream in the history buffer) (step 5 in Figure 3.2) and issues prefetch requests for the instruction blocks in the new spatial regions.

As an optimization, SHIFT employs multiple stream buffers (four in our design) to replay multiple streams, which may arise due to frequent traps and context switches in server workloads. The least-recently-used stream is evicted upon allocating a new stream. When the active stream address buffer reaches its capacity, the oldest spatial region record is evicted to make space for the incoming record.

For the actual design parameters we performed the corresponding sensitivity analysis and found that a spatial region size of eight, a lookahead of five and a stream address buffer capacity of twelve achieve the maximum performance.

3.2 Virtualized SHIFT

The baseline SHIFT design described in Section 3.1 relies on dedicated history storage. The principal advantage of dedicated storage is that it ensures non-interference with the cache hierarchy. However, this design choice carries several drawbacks, including (1) new storage structures for the history buffer and the index table, (2) lack of flexibility with respect to capacity allocation, and (3) considerable storage expense to support multiple histories as required for workload consolidation. To overcome these limitations, we embed the SHIFT history buffer in the LLC leveraging the virtualization framework [13].

History Virtualization. To virtualize the instruction history buffer, SHIFT first allocates a portion of the physical address space for the history buffer. History buffer entries are stored in the LLC along with regular instruction and cache blocks. For the index table entries, SHIFT extends the LLC tag array to augment the existing instruction block tags with pointers to the shared history buffer records.

SHIFT reserves a small portion of the physical address space that is hidden from the operating system. The reserved address space starts from a physical address called the History Buffer Base (HBBase) and spans a contiguous portion of the physical address space. The size of the reserved physical address space for the history buffer can change based on the instruction working set size of a workload.

The history buffer is logically a circular buffer; however, the actual storage is organized as cache blocks. To access a spatial region record in the history buffer, the value of the pointer to the spatial region record is added to HBBase to form a physical address and an LLC lookup is performed for that physical address. Each cache block that belongs to the history buffer contains multiple spatial region records, therefore, each history buffer read and write operation spans multiple spatial region records. The LLC blocks that belong to the history buffer are non-evictable, which ensures that the entire history buffer is always present in the LLC. Non-eviction support is provided at the cache controller through trivial logic that compares a block's address to the address range reserved for the history. As an alternative, a cache partitioning scheme (e.g., Vantage [66]) can easily guarantee the required cache partition for the history buffer.

The index table, which contains pointers to the spatial region records in the history buffer, is embedded in the LLC by extending the tag array with pointer bits. This eliminates the need for a dedicated index table and provides an index lookup mechanism for free by coupling index lookups with instruction block requests to LLC (details below). Although each tag is augmented with a pointer, the pointers are used only for instruction blocks. Each instruction block tag in the LLC can point to the most recent occurrence of the corresponding instruction

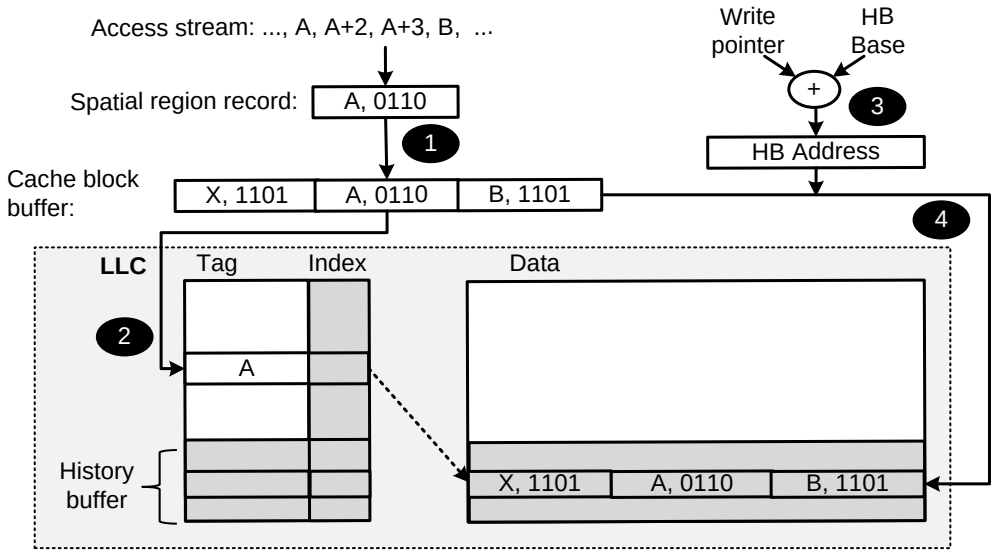

Figure 3.3: SHIFT's virtualized history and data flow to record temporal instruction streams. Virtualized history components (index pointers and shared history buffer) are shaded in the LLC.

block address in the history buffer. The width of the pointer is a function of the history buffer size and is independent of the LLC capacity. In our design, each pointer is 15 bits allowing for an up to 32K-entry history buffer.

In Figure 3.3 and Figure 3.4 , the shaded areas indicate the changes in the LLC due to history virtualization. The LLC tag array is augmented with pointers and a portion of the LLC blocks are reserved for the history buffer. The LLC blocks reserved for the history buffer are distributed across different sets and banks; however, we show the history buffer as a contiguous space in the LLC to simplify the figure.

Recording. Figure 3.3 illustrates how the SHIFT logic next to the history generator core records the retire-order instruction stream in the shared and virtualized history buffer. First, the history generator core forms spatial region records as described in Section 3.1. Because the history buffer is accessed at cache-block granularity in virtualized SHIFT, the history generator core accumulates the spatial region records in a cache-block buffer (CBB), instead of sending each spatial region record to the LLC one by one (step 1). However, upon each spatial region record insertion into the CBB, the history generator core issues an index update request to the LLC

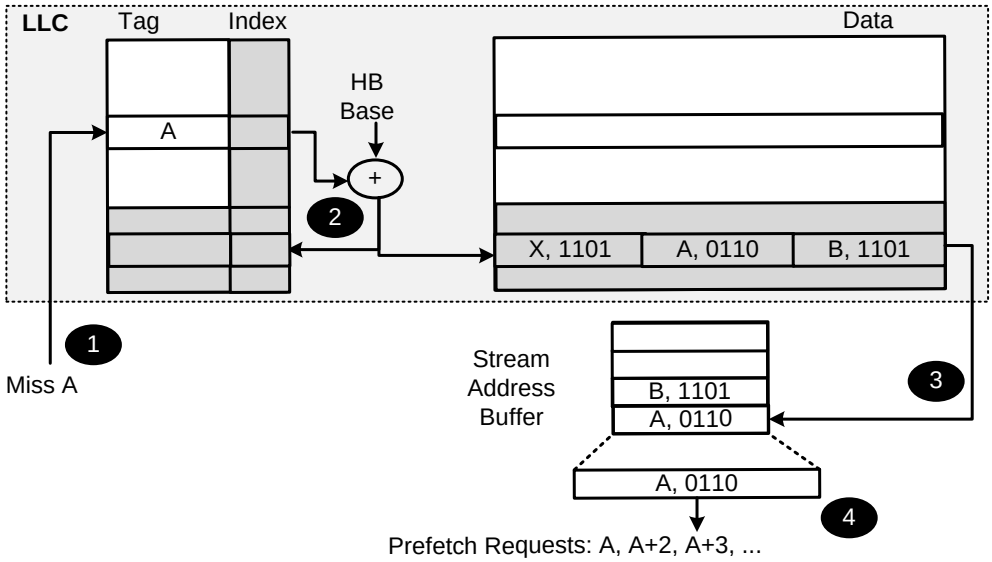

Figure 3.4: SHIFT's virtualized history and data flow to replay temporal instruction streams. Virtualized history components (index pointers and shared history buffer) are shaded in the LLC.

for the spatial region's trigger address by sending the current value of the write pointer (step 2). The LLC performs a tag lookup for the trigger instruction block address and if the block is found in the LLC, its pointer is set to the write pointer value sent by the history generator core. After sending the index update request, the history generator core increments the write pointer.

Once the CBB becomes full, its content needs to be inserted into the virtualized history buffer. To accomplish this, the SHIFT logic next to the history generator core computes the write address by adding the value of the write pointer to HBBase (step 3), and then flushes the CBB into the LLC to the computed write address (step 4).

Replaying. While the history generator core is continuously writing its instruction access history into the LLC-resident history buffer, the rest of the cores executing the workload read the history buffer to anticipate their instruction demands. Figure 3.4 illustrates how each core replays the shared instruction stream.

SHIFT starts replaying a new stream when there is a miss in the instruction cache. For every

demand request for an instruction block, the LLC sends the instruction block and the index pointer stored next to the tag of the instruction block to the requesting core (step 1). The SHIFT logic next to the core constructs the physical address for the history buffer by adding the index pointer value to HBBase and sends a request for the corresponding history buffer block (step 2). Finally, the LLC sends the history buffer block to the stream address buffer of the core (step 3).

Upon arrival of a history buffer block, the stream address buffer allocates a new stream and places the spatial region records in the history buffer block into the new stream. The stream address buffer constructs instruction block addresses and issues prefetch requests for them (step 4) as described in Section 3.1. If a retired instruction matches with an address in the stream address buffer, the stream is advanced by issuing a history read request to the LLC following the index pointer maintained as part of the stream in the stream address buffer (i.e., by incrementing the index pointer by the number of history buffer entries in a block and constructing the physical address as in step 2) .

Hardware cost. Each spatial region record, which spans eight consecutive instruction blocks, maintains the trigger instruction block address (34 bits) and 7 bits in the bit vector (assuming a 40- bit physical address space and 64-byte cache blocks). A 64-byte cache block can accommodate 12 such spatial region records. A SHIFT design with 32K history buffer entries (i.e., spatial region records) necessitates 2,731 cache lines for an aggregate LLC footprint of 171KB.

With history entries stored inside the existing cache lines and given the trivial per-core prefetch control logic, the only source of meaningful area overhead in SHIFT is due to the index table appended to the LLC tag array. The index table augments each LLC tag with a 15-bit pointer into the 32K-entry virtualized history buffer. In an 8MB LLC, these extra bits in the tag array constitute the 240KB storage overhead (accounting for the unused pointers for associated with regular data blocks in the LLC).

3.3 SHIFT Design for Workload Consolidation

Multiple server workloads running concurrently on a manycore CMP also benefit from SHIFT, as SHIFT relies on virtualization allowing for a flexible history buffer storage mechanism. SHIFT requires two minor adjustments in the context of workload consolidation. First, a history buffer per workload should be instantiated in the LLC. SHIFT's 171KB (2% of an 8MB LLC) history buffer size is dwarfed by the LLC capacities of contemporary server processors and the performance degradation due to the LLC capacity reserved for SHIFT is negligible. So, we instantiate one history buffer per workload. Second, the operating system or the hypervisor needs to assign one history generator core per workload and set the history buffer base address (HBBase) to the HBBase of the corresponding history buffer for all cores in the system. After these two adjustments, the record and replay of instruction streams work as described in Section 3.2.

Even with extreme heterogeneity (i.e., a unique workload per core), SHIFT provides a storage advantage over PIF as the history buffers are embedded in the LLC data array and the size of the index table embedded in the LLC tag array does not change. Because the size of the index pointers only depends on the size of the corresponding history buffer, it does not change with the number of active per-workload history buffers.

3.4 Methodology

We evaluate SHIFT and compare it to the state-of-the-art instruction streaming mechanism with per-core private instruction history, PIF [27], using trace-based and cycle-accurate simulations of a 16-core CMP, running server workloads. For our evaluation, we use Flexus [88], a Virtutech Simics-based, full-system multiprocessor simulator, which models the SPARC v9 instruction set architecture. We simulate CMPs running the Solaris operating system and executing the server workload suite listed in Table 2.1.

We use trace-based experiments for our opportunity study and initial predictor results by using

Processing Nodes	UltraSPARC III ISA, sixteen 2GHz cores *Intel Xeon* ($25mm^2$): 4-wide dispatch/retirement 128-entry ROB, 32-entry LSQ *ARM Cortex-A15* ($4.5mm^2$): 3-wide dispatch/retirement, 60-entry ROB, 16-entry LSQ *ARM Cortex-A8* ($1.3mm^2$): 2-wide dispatch/retirement
I-Fetch Unit	32KB, 2-way, 64B blocks, 2-cycle load-to-use L1-I cache Hyrid branch predictor: (16K-entry gShare, Bimodal, Meta selector)
L1 I&D Caches	32KB, 2-way, 64B blocks 2-cycle load-to-use latency, 32 MSHRs
L2 NUCA Cache	Unified, 64B blocks, 512KB per core, unified, 16-way, 64 MSHRs *NUCA*: 1 bank per tile, 5-cycle hit latency *UCA*: 1 bank per 2 cores, 6-cycle hit latency
Interconnect	*Mesh*: 4x4 2D, 3 cycles per hop *Crossbar*: 5 cycles
Main memory	45ns access latency

Table 3.1: Parameters of the server architecture evaluated.

traces with 32 billion instructions (two billion per core) in steady state. For the DSS workloads, we collect traces for the entire query execution. Our traces include both the application and the operating system instructions.

For performance evaluation, we use the SimFlex multiprocessor sampling methodology [88], which extends the SMARTS sampling framework [89]. Our samples are collected over 10-30 seconds of workload execution (for the DSS workloads, they are collected over the entire execution). For each measurement point, we start the cycle-accurate simulation from checkpoints with warmed architectural state and run 100K cycles of cycle-accurate simulation to warm up the queues and the interconnect state, then collect measurements from the subsequent 50K cycles. We use the ratio of the number of application instructions to the total number of cycles (including the cycles spent executing operating system code) to measure performance; this metric has been shown to accurately reflect overall system throughput [88]. Performance measurements are computed with an average error of less than 5% at the 95% confidence level.

We model a CMP with cores modeled after an ARM Cortex-A15 [84] interconnected with a

mesh network. We also evaluate a CMP with a crossbar interconnect. For the performance density study, we also consider a fat-OoO core (representative of contemporary Xeon-class cores) and a lean in-order core (similar to an ARM Cortex-A8 [9]), which all operate at 2GHz to simplify the comparison. The design and architectural parameter assumptions are listed in Table 3.1. Cache parameters, including the SRAMs for PIF's history buffer and index table, are estimated using CACTI [55].

State-of-the-art prefetcher configuration. We compare SHIFT's effectiveness and history storage requirements with the state-of-the-art instruction streaming mechanism, PIF [27]. Like other instruction and data streaming mechanisms [28, 87], PIF employs a per-core history buffer and index table. PIF records and replays spatial region records with eight instruction blocks. Each spatial region record maintains the trigger instruction block address (34 bits) and 7 bits in the bit vector. Hence, each record in the history buffer contains 41 bits. PIF requires 32K spatial region records in the history buffer targeting 90% instruction miss coverage [27], also validated with our experiments. As a result, the history buffer is 164KB for each core in total.

Each entry in PIF's index table contains an instruction block address (34 bits) and a pointer to the history buffer (15 bits) adding up to 49 bits per entry. According to our sensitivity analysis, the index table requires 8K entries for the target 90% instruction miss coverage. The actual storage required for the 8K-entry index table is 49KB per core.

We also evaluate a PIF design with a total storage cost equal to that of SHIFT. Since SHIFT stores the history buffer entries inside existing LLC cache blocks, its only source of storage overhead is the 240KB index table embedded in the LLC tag array. An equal-cost PIF design affords 2K spatial region records in the history buffer and 512 entries in the index table per core. We refer to this PIF design as PIF-2K and the original PIF design as PIF-32K to differentiate between the two design points in the rest of the paper.

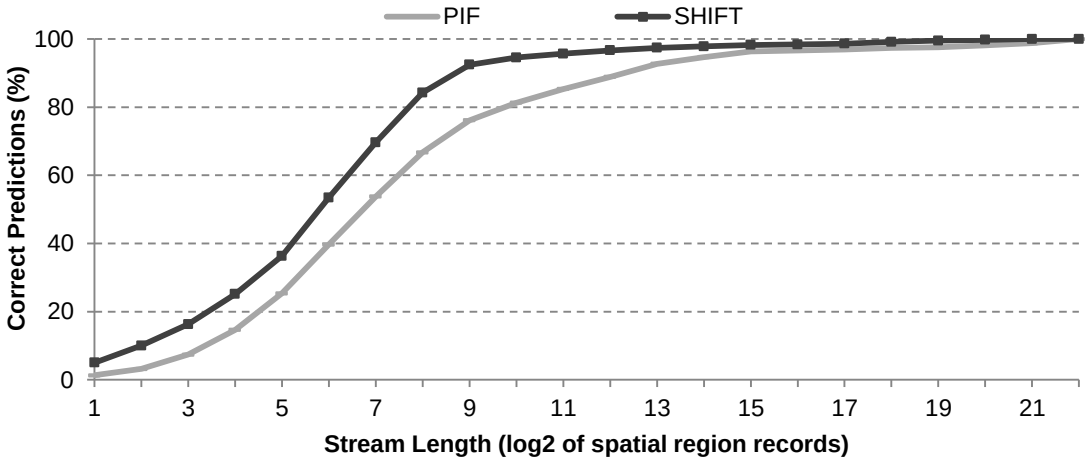

Figure 3.5: Comparison of temporal stream lengths in PIF and SHIFT histories and their contribution to correct predictions.

3.5 Evaluation

3.5.1 Temporal Stream Lengths

The distinguishing feature of instruction streaming mechanisms is their ability to replay long repetitive streams that offer high miss coverage and timely prefetches. Although the first few blocks following the head of a stream might be delayed as the history buffer entries are fetched from the LLC, long temporal streams amortize this delay by providing high lookahead and prefetching the instruction blocks along the way far in advance of demand. As a result, the longer the repetitive temporal streams are, the higher miss coverage they provide.

We analyze the temporal stream lengths in SHIFT and PIF [27] and their contribution to correct predictions in Figure 3.5. In Figure 3.5, a stream length of 5 corresponds to 32 spatial region entries, each encoding 8 instruction blocks, corresponding to 256 contiguous instruction blocks in total (if all the blocks in regions are accessed). Although the stream lengths are highly variable, the majority of the correct predictions are provided by medium and long streams both in SHIFT and PIF. Compared to PIF, streams contributing to useful predictions are slightly shorter in SHIFT. This can be attributed to the subtle divergences in the control flow due to data dependent branches in the history generator core's instruction stream and indicates

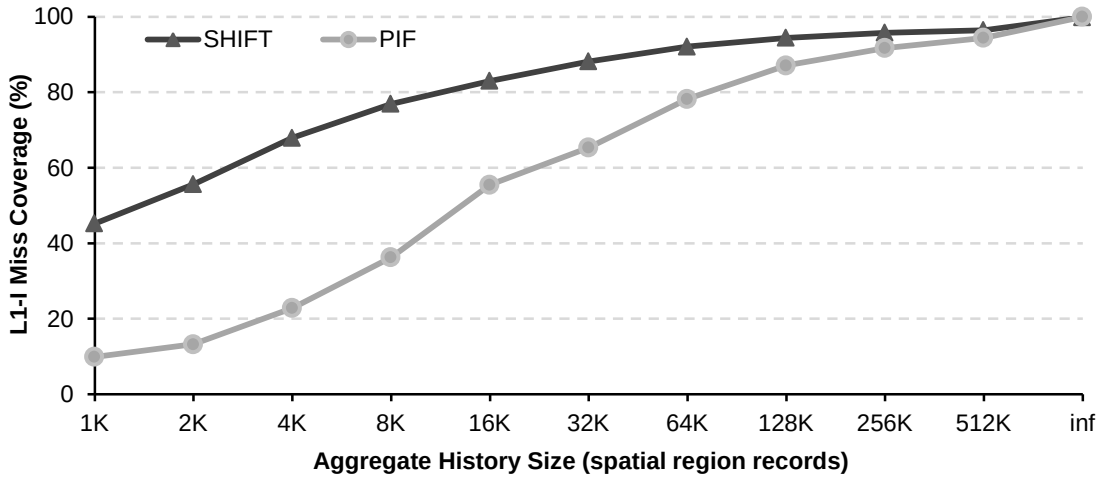

Figure 3.6: Percentage of instruction misses predicted.

slightly lower miss coverage for SHIFT as shorter repetitive streams result in more frequent stream lookups, thus more stream heads that cannot be predicted.

3.5.2 Instruction Miss Coverage

To show SHIFT's effectiveness, we first compare the fraction of instruction cache misses predicted by SHIFT to PIF [27]. For the purposes of this study, we only track the predictions that would be made through replaying recurring instruction streams in stream address buffers and do not prefetch or perturb the instruction cache state.

Figure 3.6 shows the fraction of instruction cache misses correctly predicted for all of the cores in the system averaged across all workloads, as the number of spatial region records in the history buffer increases. The history size shown is the aggregate for PIF in the 16-core system evaluated, whereas for SHIFT, it is the overall size of the single shared history buffer.

Because the history buffer can maintain more recurring temporal instruction streams as its size increases, the prediction capabilities of both designs increase monotonically with the allocated history buffer size. Because the aggregate history buffer capacity is distributed across cores, PIF's coverage always lags behind SHIFT. For relatively small aggregate history buffer sizes, PIF's small per-core history buffer can only maintain a small fraction of the instruction

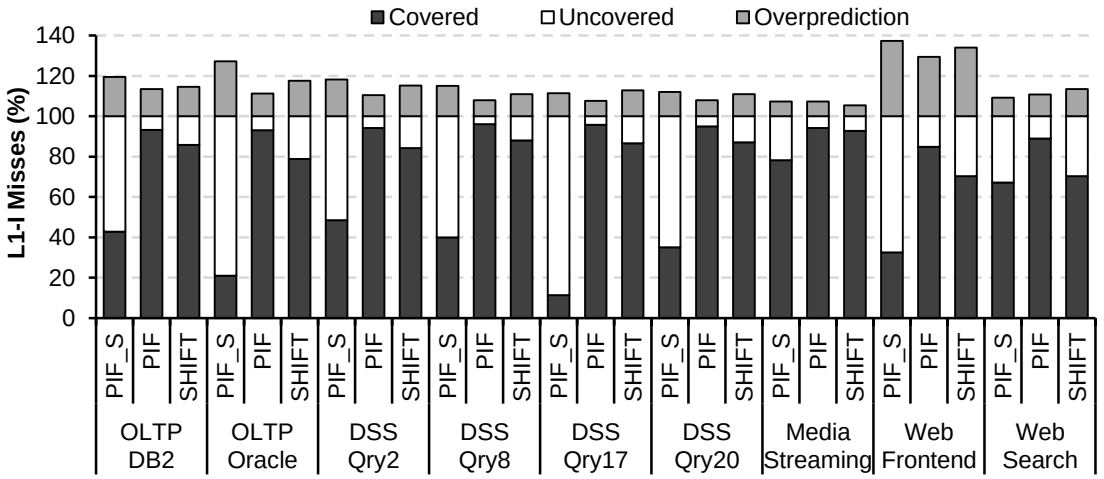

Figure 3.7: Percentage of instruction misses covered and overpredicted.

working set size. As the history buffer size increases, PIF's per-core history buffer captures a larger fraction of the instruction working set. For all aggregate history buffer sizes, SHIFT can maintain a higher fraction of the instruction working set compared to PIF by employing a single history buffer, rather than distributing it across cores. As all the cores can replay SHIFT's shared instruction stream history, SHIFT's miss coverage is always greater than PIF for equal aggregate storage capacities. Because the history sizes beyond 32K return diminishing performance benefits, for the actual SHIFT design, we pick a history buffer size of 32K records.

We compare SHIFT's actual miss coverage with PIF for each workload, this time accounting for the mispredictions as well, as mispredictions might evict useful but not-yet-referenced blocks in the cache. For this comparison, we use the two PIF design points described in Section 3.4.

Figure 3.7 shows the instruction cache misses eliminated (covered) and the mispredicted instruction blocks (overpredicted) normalized to the instruction cache misses in the baseline design without any prefetching. On average, SHIFT eliminates 85% of the instruction cache misses with 16% overprediction, while PIF-32K eliminates 92% of the instruction cache misses with 13% overprediction, corroborating prior results [27]. However, PIF-2K, which has the same aggregate storage overhead as SHIFT, can eliminate only 53% of instruction cache misses on average, with a 20% overprediction ratio. The discrepancy in miss coverage between PIF-2K and SHIFT is caused by the limited instruction stream history stored by each core's smaller

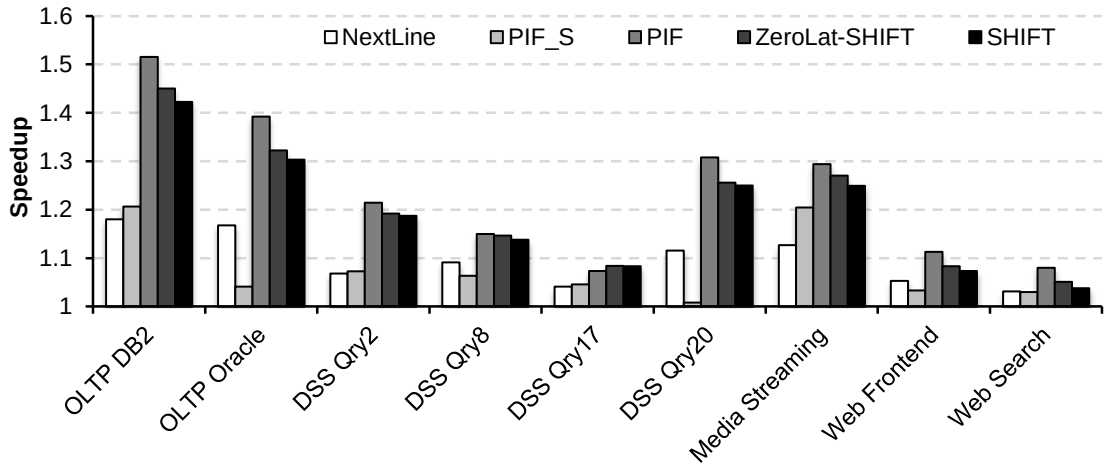

Figure 3.8: Performance comparison with a mesh on-chip interconnect.

history buffer in PIF, which falls short of capturing the instruction working set.

In conclusion, by sharing the instruction history generated by a single core, all cores running a common server workload attain similar benefits to per-core instruction streaming, but with a much lower storage overhead.

3.5.3 Performance Comparison

We compare SHIFT's performance against PIF-32K, PIF-2K and the next-line prefetcher normalized to the performance of the baseline system with OoO cores interconnected with a mesh network, where no instruction prefetching mechanism is employed in Figure 3.8.

The difference in PIF-32K and SHIFT's speedups stems from two main differences. First, SHIFT has slightly lower miss coverage compared to PIF-32K, as shown in Figure 3.7. Second, SHIFT's history buffer is embedded in the LLC resulting in accesses to the LLC for reading the history buffer, which delays the replay of streams until the history buffer block arrives at the core. Moreover, history buffer reads and writes to LLC incur extra LLC traffic as compared to PIF-32K. To compare the performance benefits resulting solely from SHIFT's prediction capabilities, we also plot the performance results achieved by SHIFT assuming a dedicated history buffer with zero access latency (ZeroLat-SHIFT).

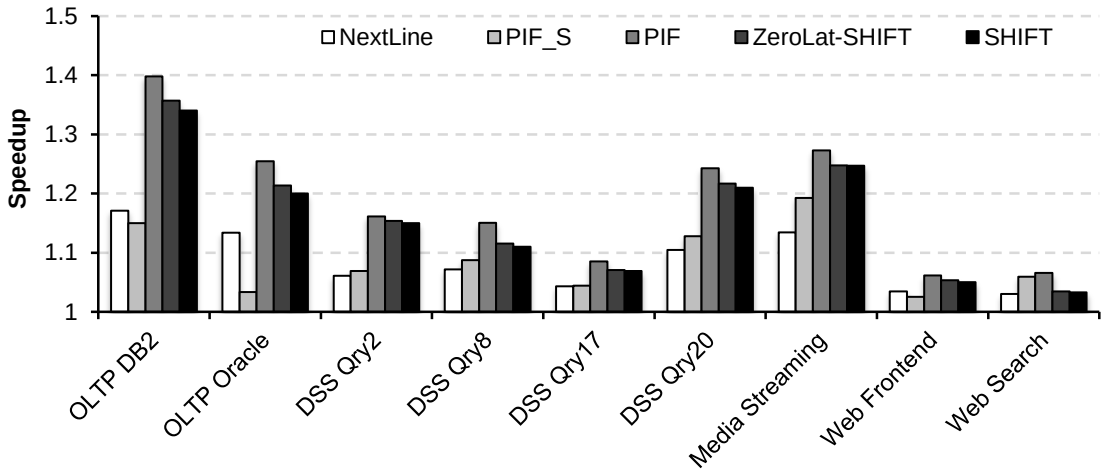

Figure 3.9: Performance comparison with a crossbar on-chip interconnect.

The relative performance improvements of zero-latency SHIFT and PIF-32K match the miss coverages shown in Figure 3.7. On average, zero-latency SHIFT provides 20% performance improvement, while PIF-32K improves performance by 21%. A realistic SHIFT design results in 1.5% speedup loss compared to zero-latency SHIFT, due to the latency of accessing the history buffer in the LLC and the traffic created by transferring history buffer data over the network. Overall, despite its low history storage cost, SHIFT retains over 90% of the performance benefit (98% of the overall absolute performance) that PIF-32K provides.

In comparison to PIF-2K, SHIFT achieves higher speedups for all the workloads as a result of its higher miss coverage as Figure 3.7 shows. Due to its greater effective history buffer capacity, SHIFT outperforms PIF-2K for all the workloads (by 9% on average). For the workloads with bigger instruction working sets (e.g., OLTP on Oracle), SHIFT outperforms PIF-2K by up to 26%.

Finally, we compare SHIFT's performance to the next-line prefetcher. Although the next-line prefetcher does not incur any storage overheads, it only provides 9% performance improvement due to its low miss coverage (35%) stemming from its incapability of predicting misses to discontinuous instruction blocks.

We also evaluate SHIFT and the other prefetchers in the context of a CMP using a crossbar

on-chip interconnect, which reduces the LLC access latency compared to a mesh interconnect (i.e., in a 4x4 2D mesh with 3-cycle latency per hop the average latency is 10 cycles, whereas crossbar provides a 5-cycle access latency to the LLC). For applications with higher L1-I miss rates, we expect the instruction-fetch stalls to be reduced in the baseline system employing a crossbar interconnect instead of a mesh interconnect, thus less performance benefit from instruction streaming in the case of the crossbar interconnect. We also expect the performance penalty to decrease for SHIFT as compared to ZeroLat-SHIFT as the metadata accesses become faster with crossbar as compared to mesh.

Figure 3.9 shows the speedups provided by various prefetching techniques. When compared to the CMP design with mesh on-chip network as shown in Figure 3.8, the speedups provided by instruction prefetching are relatively lower, but still significant in the case of a crossbar on-chip interconnect. For example, while SHIFT provides 42% and 30% performance improvement for OLTP on DB2 and Oracle respectively in the case of the mesh interconnect, the performance improvement is 34% and 20% respectively in the case of the crossbar interconnect. Overall, SHIFT delivers 19% performance improvement on average for mesh and 15% for crossbar.

3.5.4 LLC Overheads

SHIFT introduces two types of LLC overhead. First, the history buffer occupies a portion of the LLC, effectively reducing its capacity. Our results indicate that the performance impact of reduced capacity is negligible. With SHIFT occupying just 171KB of the LLC capacity, the measured performance loss with an 8MB LLC is under 1%.

The second source of overhead is due to the extra LLC traffic, generated by (1) read and write requests to the history buffer; (2) useless LLC reads as a result of mispredicted instruction blocks, which are discarded before used by the core; and (3) index updates issued by the history generator core. Figure 3.10 illustrates the extra LLC traffic generated by SHIFT normalized to the LLC traffic (due to both instruction and data requests) in the baseline system without any prefetching. History buffer reads and writes increase the LLC traffic by 6%, while discards

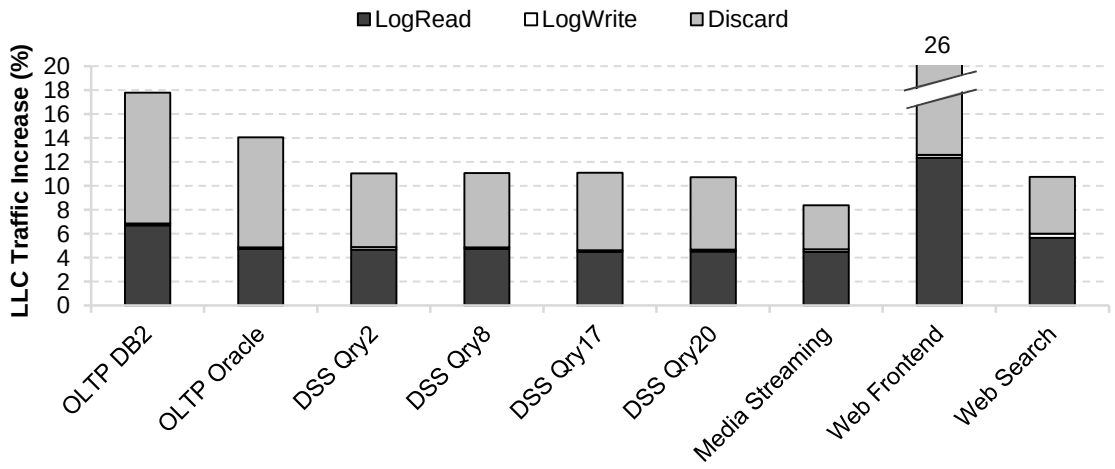

Figure 3.10: LLC traffic overhead.

account for the 7% of the baseline LLC traffic on average. The index updates (not shown in the graph) are only 2.5% of the baseline LLC traffic; however, they only increase the traffic in the LLC tag array.

In general, we note that LLC bandwidth is ample in our system, as server workloads have low ILP and MLP, plus the tiled design provides for a one-to-one ratio of cores to banks, affording very high aggregate LLC bandwidth. With average LLC bandwidth utilization well under 10%, the additional LLC traffic even for the worst case workload (web frontend) is easily absorbed and has no bearing on performance in our studies.

3.5.5 Workload Consolidation

Figure 3.11 shows the performance improvement attained by SHIFT in comparison to the next-line prefetcher, PIF-2K and PIF-32K, in the presence of multiple workloads running on a server CMP. In this experiment, we use the 16-core server processor with a mesh on-chip interconnect described in Section 3.4. We consolidate two traditional (OLTP on Oracle and web frontend) and two emerging (media streaming and web search) server workloads. Each workload runs on four cores and has its own software stack (i.e., separate OS images). For SHIFT, each workload has a single shared history buffer with 32K records embedded in the LLC.

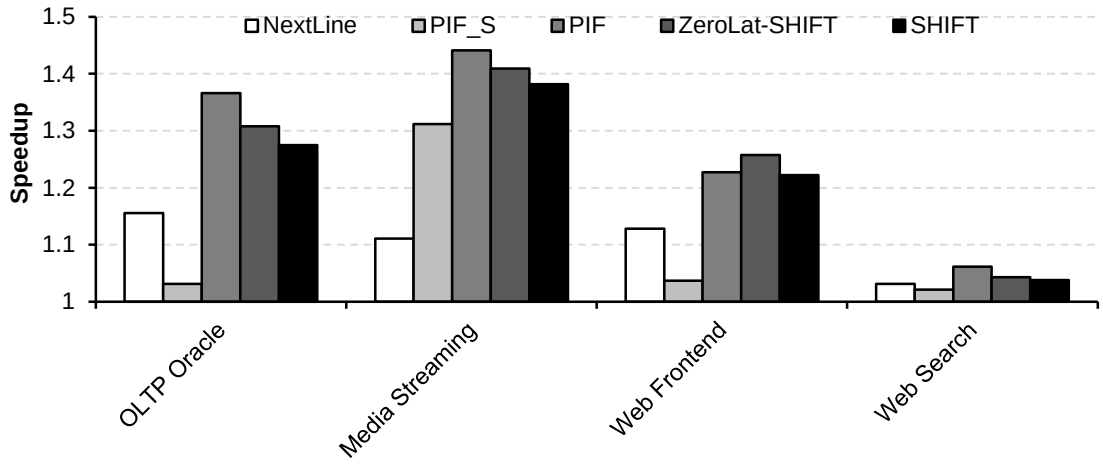

Figure 3.11: Speedup for workload consolidation.

We see that the speedup trends for the various design options follow the same trend as the standalone workloads, as shown in Section 3.5.3. SHIFT delivers 95% of PIF-32K's absolute performance and outperforms PIF-2K by 12% and the next-line prefetcher by 11% on average.

Zero-latency SHIFT delivers 25% performance improvement over the baseline, while SHIFT achieves 22% speedup on average. The 3% difference mainly results from the extra LLC traffic generated by virtualized SHIFT. The LLC traffic due to log reads remains the same in the case of workload consolidation as all the cores read from their corresponding shared history embedded in the LLC. However, the index updates and log writes increase with the number of workloads, as there is one history generator core per workload. While log writes increase the fetch traffic by 1.1%, index updates, which only increase the traffic in the LLC tag array, correspond to 15% of the baseline fetch accesses.

Overall, we conclude that in the presence of multiple workloads, SHIFT's benefits are unperturbed and remain comparable to the single-workload case.

3.5.6 Performance Density Implications

To quantify the performance benefits of the different prefetchers as a function of their area cost, we compare SHIFT's performance density (PD) with PIF-32K and PIF-2K. We consider

the three core designs described in Section 3.4 namely, Xeon, Cortex-A15, and Cortex-A8.

SHIFT improves performance-density over PIF-32K as a result of eliminating the per-core instruction history, while retaining similar performance benefits. Compared to PIF-32K, SHIFT improves the overall performance by 16 to 20%, depending on the core type, at a negligible area cost per core (0.96mm^2 in total, as opposed to PIF-32K's 14.4mm^2 cost in aggregate in a 16-core CMP). While PIF-32K offers higher absolute performance, the relative benefit is diminished due to the high area cost. As a result, SHIFT improves PD over PIF-32K for all three core microarchitectures. As expected, the biggest PD improvement is registered for lean cores (16% and 59% for Cortex-A15 and Cortex-A8 respectively); however, even the fat-core design (Xeon) enjoys a 2% improvement in PD due to SHIFT's low area cost.

In comparison to PIF-2K, SHIFT achieves higher miss coverage due to the better capacity management of the aggregate history buffer storage. Although PIF-2K occupies the same aggregate storage area as SHIFT, SHIFT almost doubles the performance improvement for the three core types, as a result of its higher miss coverage. Consequently, SHIFT improves performance density over PIF-2K by around 9% on average for all the core types.

SHIFT improves the absolute performance-density for both lean-core designs and the fat-core design over a no-prefetch system, while providing 98% of performance of the state-of-the-art instruction streaming mechanism, demonstrating the feasibility of area-efficient high performance instruction streaming for servers.

3.5.7 Power Implications

SHIFT introduces power overhead to the baseline system due to two factors: (1) history buffer reads and writes to/from the LLC and (2) index reads and writes to/from the LLC. To quantify the overall power overhead induced by SHIFT, we use CACTI [55] to estimate the LLC power (both for index pointers in the tag array and history buffers in the data array) and custom NoC power models to estimate the link, router switch fabric and buffer power in the NoC [52]. We find the additional power overhead due to history buffer and index activities in the LLC to be

less than 150mW in total for a 16-core CMP. This corresponds to less than 2% power increase per core. We thus conclude that the power consumption due to SHIFT is negligible.

3.6 Discussions

3.6.1 Choice of History Generator Core

We show SHIFT's miss coverage and performance improvement by employing one history generator core picked at random in our studies. In our experience, in a sixteen-core system, there is no sensitivity to the choice of the history generator core.

Although the cores executing a homogeneous server workload exhibit common temporal instruction streams, there are also spontaneous events that might take place both in the core generating the shared instruction history and the cores reading from the shared instruction history, such as the OS scheduler, TLB miss handlers, garbage collector, and hardware interrupts. In our experience, such events are rare and only briefly hurt the instruction cache miss coverage due to the pollution and fragmentation of temporal streams. In case of a long-lasting deviation in the program control flow of the history generator core, a sampling mechanism that monitors the instruction miss coverage and changes the history generator core accordingly can overcome the disturbance in the shared instruction history.

3.6.2 Virtualized PIF

Although virtualization could be readily used with streaming mechanisms using per-core history, such designs would induce high capacity and bandwidth pressure in the LLC. For example, virtualizing PIF's per-core history buffers would require 2.7MB of LLC capacity and this requirement grows linearly with the number of cores. Furthermore, as each core records its own history, the bandwidth and power consumption in the LLC also increase linearly with the number of cores. By sharing the instruction stream history, SHIFT not only saves area but also minimizes the pressure on the LLC compared to virtualized per-core instruction

streaming mechanisms.

3.7 Concluding Remarks

Instruction-fetch stalls are a well-known cause of performance loss in server processors due to the large instruction working sets of server workloads. Sophisticated instruction-prefetch mechanisms developed by researchers specifically for this workload class have been shown to be highly effective at mitigating the instruction-stall bottleneck by recording, and subsequently replaying, entire instruction sequences. However, for high miss coverage, existing instruction streaming mechanisms require prohibitive storage for the instruction history due to the large instruction working sets and complex control flow. The storage overhead is further exacerbated by the increasing core counts on chip.

This work confronted the problem of high storage overhead in instruction streaming. We observed that the instruction history among all of the cores executing a server workload exhibits significant commonality and showed that it is amenable to sharing. Building on this insight, we introduced SHIFT – a shared history instruction streaming mechanism. SHIFT records the instruction access history of a single core and shares it among all of the cores running the same workload. In a 16-core CMP, SHIFT delivers over 90% of the performance benefit of PIF, a state-of-the-art instruction streaming mechanism, while largely eliminating the prohibitive per-core storage overhead associated with PIF.

4 Confluence: Unifying Instruction-Supply Metadata

In this chapter, we identify and eliminate the redundancy by focusing on two performance-critical and storage-intensive structures: the instruction prefetcher and the branch target buffer (BTB). We observe that state-of-the-art instruction cache prefetchers achieve extremely high miss coverage through *temporal instruction streaming* [27, 28, 47]. As explained in Chapter 3, the key idea of temporal streaming is to record the history of L1-I accesses at the block granularity and subsequently replay the history in order to prefetch the blocks into the instruction cache. The instruction streaming mechanisms eliminate the vast majority of L1-I misses, because they maintain the entire control-flow history as a continuous stream of instruction block addresses. By introducing SHIFT, we provide a practical and highly effective instruction streaming mechanism.

On the BTB side, the massive instruction working sets and complex control flow of server applications require tracking many thousands of branch targets, necessitating over 200KBs of BTB storage capacity for perfect coverage as shown in Section 2.2.2. Since BTBs of that size are impractical due to their dedicated storage overheads and latency penalties, researchers proposed virtualizing the BTB state into the LLC and prefetching it into a small conventional BTB, thus decoupling the large BTB footprint from the core [11, 12, 23].

We observe that in both cases – instruction prefetching and BTB prefetching – the prefetcher

metadata contains a record of the application's control flow history. While the instruction prefetcher maintains its history at the instruction block granularity, the BTB prefetcher maintains its history at instruction granularity. Due to the different granularities at which history is maintained, existing schemes require dedicated histories and prefetchers for both the instruction cache and the BTB.

The key contribution of this chapter is in identifying the redundancy in the control flow metadata for both types of prefetchers and eliminating it by unifying the two histories. To that end, we introduce Confluence – a frontend design with a single prefetcher (with unified metadata) feeding both the L1-I and the BTB. An important challenge Confluence addresses is in managing the disparity in the granularity of control flow required by each of the prefetchers. Whereas an I-cache prefetcher needs to track block-grain addresses, a BTB must reflect fine-grain information of the individual branches.

Confluence overcomes this problem by exploiting a critical insight that a BTB only tracks branch targets, which do not depend on whether or not the branch is taken or even executed. Based on this insight, Confluence maintains the unified control flow history at the block granularity and for each instruction block brought into the L1-I, it eagerly inserts the targets of all PC-relative branches contained in the block into the BTB. Because the control flow exhibits spatial locality, the eager insertion policy provides high intra-block coverage without requiring fine-grain knowledge of the control flow. Finally, to overcome the exorbitant bandwidth required to insert 3-4 branches found in a typical cache block into the BTB, Confluence employs a block-based BTB organization, which is also beneficial for reducing the tag overhead.

The contributions of this work are as follows:

- We show that a single block-grain temporal stream is sufficient for prefetching into both L1-I and the BTB, as the instruction blocks encapsulate the instruction-grain information necessary for the BTB. Based on this observation, we introduce Confluence – a unified instruction-supply architecture that maintains one set of metadata used by a single prefetcher for feeding both the L1-I and the BTB.

- We propose AirBTB, a lightweight block-based BTB design for Confluence that takes advantage of a block-grain temporal stream and spatial locality within blocks to maintain only a small set of BTB targets.
- We show that Confluence equipped with AirBTB can eliminate 93% of the misses of a conventional BTB design with the same storage budget (around 10KB), and 85% of all L1-I misses, providing 85% of the speedup possible with a perfect L1-I and BTB.
- Compared to the state-of-the-art BTB prefetcher, AirBTB eliminates 30% more misses within the same BTB and prefetcher history storage budget, while providing 8% higher performance.

The rest of this chapter is organized as follows. In Section 4.1, we explain the performance limitations and associated area overheads of existing instruction-supply mechanisms and sources of redundancy in instruction-supply metadata. We describe the Confluence design and how it synchronizes BTB content with L1-I using AirBTB in Section 4.2. Section 4.3 details our methodology. We evaluate Confluence and compare it against prior work in Section 4.4. Finally, we conclude in Section 4.5.

4.1 Performance Limitations and Area Overheads of Instruction-Supply Mechanisms

In this section, we briefly describe the state-of-the-art mechanisms to alleviate frequent misses in the BTB and L1-I. We quantify their performance benefits and associated storage overheads, and demonstrate that there is a need for an effective and low-cost unified mechanism for storing and managing instruction-supply metadata.

4.1.1 Conventional Instruction-Supply Path

Extracting the highest performance from a core necessitates supplying the core with a useful stream of instructions to execute continuously. To do so, modern processors employ branch

predictors accommodating conditional branch history and branch target buffers to predict the correct-path instructions to execute.

High-accuracy branch prediction necessitates capturing the prediction metadata of the entire instruction working set of a given application in the predictor tables. Unfortunately, server applications with large instruction working sets exacerbate the storage capacity requirements of these predictors with low access-latency requirements as shown in Section 2.2.2

Recent work has examined hierarchical BTBs that combine a small-capacity low-latency first level with a large-capacity but slower second level. The state-of-the-art proposals combine a two-level BTB with a dedicated transfer engine, which we refer to as a BTB prefetcher, that move multiple correlated entries from the second level into the first upon a miss in the first-level BTB. One approach, called PhantomBTB, uses temporal correlation, packing several entries that missed consecutively in the first level into blocks that are stored in the LLC using predictor virtualization [12]. Another approach, called Bulk Preload and implemented in the IBM zEC12, moves a set of spatially-correlated regions (4KB) between a dedicated 24K-entry second-level BTB structure and the first level [11].

For both two-level designs, second-level storage requirements are more than 200KB per core. Moreover, accesses to the second level are triggered by misses in the first level, exposing the core to the latency of the second-level structure. For PhantomBTB, this latency is a function of NOC and LLC access delays, likely running into tens of cycles for a manycore CMP. In the case of bulk preload, this latency is in excess of 15 cycles [11].

While predicting the correct-path instructions to execute is essential for high performance, serving those instructions from the L1-I cache is also performance-critical in order not to expose the core to the long access latencies of lower levels of the cache hierarchy. Doing so necessitates predicting the instructions that are likely to be fetched and proactively fetching the corresponding instruction blocks from the lower levels of the cache hierarchy into L1-I (or prefetch buffers). To that end, fetch-directed prefetching (FDP) [65] decouples the branch predictor from the L1-I and lets the branch predictor run ahead to explore the future control

flow. The instruction blocks that are not present in the L1-I along the predicted path are prefetched into the L1-I.

Although huge BTBs are effective at accommodating the target addresses for all taken branches in the instruction working set, when leveraged for FDP, they fall short of realizing the performance potential of a frontend with a perfect L1-I (i.e., L1-I that always hits) [28]. FDP's limitations are two-fold. First, because the branch predictor generates just one or two predictions per cycle, its lookahead is limited and is often insufficient to hide the long-latency accesses to the lower levels of the cache hierarchy, which includes the round-trip time to the LLC and the LLC access itself. Second, because the branch predictor speculatively runs ahead of the fetch unit to provide sufficient prefetch lookahead, its miss rate geometrically compounds, increasingly predicting the wrong-path instructions. As a result, even with a perfect BTB, FDP significantly suffers from fetch stalls.

4.1.2 Covering L1-I Misses with Stream-Based Prefetching

To overcome FDP's lookahead and accuracy limitations, the state-of-the-art instruction prefetchers [27, 28, 44, 47] exploit temporal correlation between instruction cache references. The control flow in server applications tends to be highly recurring at the request level due to serving the same types of requests perpetually. Because of the recurring control flow, the core frontend generates repeating sequences of instruction addresses, so-called temporal instruction streams. For example, in the address sequence *A,B,C,D,A,B,C,E*, the subsequence *A,B,C* is a temporal stream. The state-of-the-art instruction prefetchers exploit temporal correlation by recording and replaying temporal instruction streams consisting of instruction block addresses. This way, every prediction made by the prefetcher triggers the fetch of a whole instruction block into L1-I. Because of the high recurrence in the control flow, temporal instruction streams span several hundreds of instruction blocks [27]. As a result, stream-based instruction prefetchers can eliminate over 90% of the L1-I misses in server applications, providing near-perfect L1-I hit rates [27, 44].

However, the aggregate storage requirements of stream-based prefetchers scale with the application working set size and core count, commonly exceeding 200KB per core. To mitigate the storage overhead, the most recent work [44], SHIFT, introduced in Chapter 3, proposes embedding the prefetcher metadata into the LLC and sharing it across the cores running the same server application, thus eliminating inter-core metadata redundancy.

4.1.3 Putting It All Together

We quantify the performance benefits and relative area overheads of all the instruction-supply mechanisms described above for a 16-core CMP running server workloads (details are listed in Section 2.2) in Figure 4.1. Both the performance and area numbers are normalized to that of a core with a 1K-entry BTB without any prefetching. For all the BTB design points (except for the baseline and SHIFT), we leverage FDP for instruction prefetching.

We evaluate an aggressive two-level BTB design composed of a 1K-entry BTB in the first level (1-cycle access latency) and a 16K-entry BTB in the second level (4-cycle access latency). Such a BTB design necessitates around 150KB storage per core,[1] corresponding to 12% of the core area footprint. For a 16-core CMP, this per-core overhead totals more than 2MB of storage.

We also evaluate PhantomBTB, a state-of-the-art hierarchical BTB design with prefetching. PhantomBTB is comprised of a 1K-entry private BTB backed by a second-level BTB virtualized in the LLC. The second level features 4K temporal groups, each spanning a 64B cache line, for a total LLC footprint of 256 KB. While not proposed in the original design, we take inspiration from SHIFT and share the second BTB level across cores executing the same workload in order to reduce the storage requirements. Without sharing, the storage overhead for PhantomBTB would increase by 16X (i.e., to 4MB) for a 16-core CMP. We observe that sharing the virtualized second-level BTB does not cause any reduction in the fraction of misses eliminated by PhantomBTB.

As Figure 4.1 shows, the 1K-entry BTB with *FDP* improves performance by just 5% over the

[1] Our core is modeled after an ARM Cortex A15.

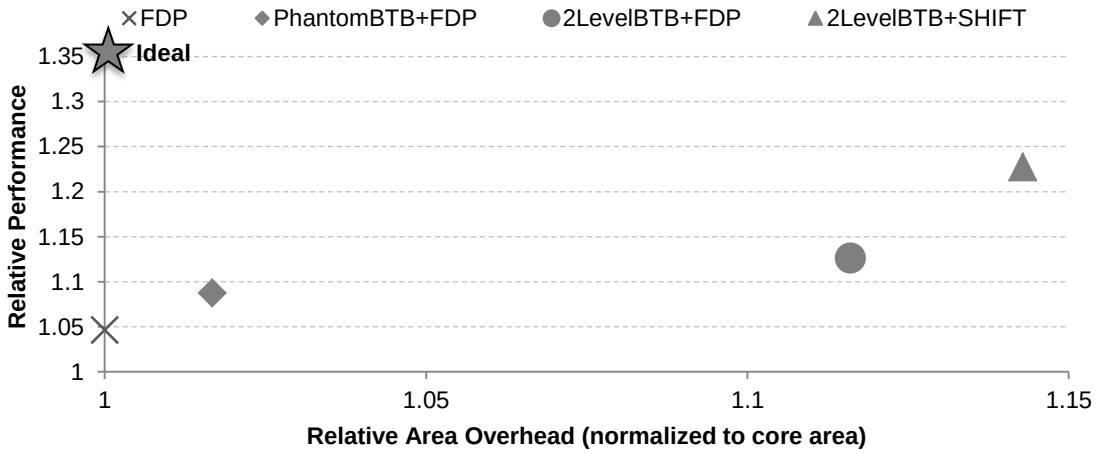

Figure 4.1: Relative performance and area overhead of various instruction-supply mechanisms.

baseline as it incurs frequent BTB misses, thus failing to identify control flow redirects upon taken branches. *PhantomBTB+FDP* provides a 9% performance improvement over the baseline despite a large second-level BTB. The underwhelming performance of this configuration is attributed to its low BTB miss coverage, which stems from the way PhantomBTB correlates branches (detailed analysis in Section 4.4.5) and delays in accessing the second level of BTB storage in the LLC.

Compared to *PhantomBTB+FDP*, *2LevelBTB+FDP* delivers better performance as the BTB metadata accesses from the second-level BTB are faster compared to the LLC access latency incurred by PhantomBTB. Among the evaluated designs, the highest performance is reached by *2LevelBTB+SHIFT*, which combines a dedicated L1-I prefetcher (SHIFT) with a two-level BTB. This configuration improves performance by 22% over the baseline, demonstrating the importance of a dedicated L1-I prefetcher and underscoring the limitations of FDP. However, *2LevelBTB+SHIFT* increases core area by 1.14X due to the high storage footprint of separate BTB and L1-I prefetcher metadata.

Finally, we observe that an *Ideal* configuration comprised of a perfect L1-I and a perfect single-cycle BTB achieves a 35% performance improvement over the baseline. *2LevelBTB+SHIFT* delivers only 60% of the *Ideal* performance improvement. Since both the L1-I and the BTB in the *2LevelBTB+SHIFT* design provide excellent coverage, the performance shortfall relative to

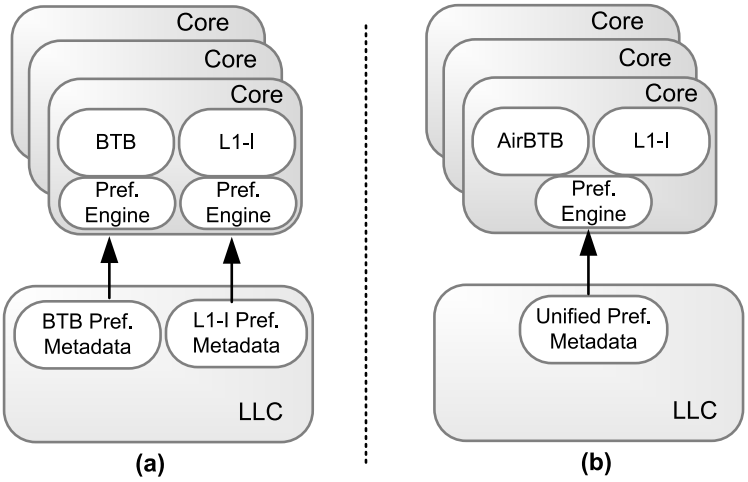

Figure 4.2: High-level organization of cores around (a) disparate BTB and L1-I prefetcher metadata (b) Confluence with unified and shared prefetcher metadata.

Ideal is caused by the delays in accessing the second level of the BTB upon a miss in the first level. Because of the high miss rate of the first level BTB, these delays are frequent and result in multi-cycle fetch bubbles.

To summarize, existing frontend designs are far from achieving the desired combination of high performance and low storage cost. Performance is limited by the delays caused in accessing the second BTB level. The high storage overheads arise from maintaining separate BTB and instruction prefetcher metadata. Because both sets of metadata capture the control flow history, they cause redundancy within a core. Moreover, because the server cores run the same application, the metadata across cores overlap significantly, causing inter-core redundancy. Eliminating the intra- and inter-core redundancy necessitates the metadata to be unified within a core and shared across cores to maximize the performance benefits harvested from a given area investment.

4.2 Confluence Design

Confluence unifies the prefetching metadata to feed BTB and L1-I synchronously as shown in Figure 4.2. Confluence relies on an existing instruction streaming mechanism, which provides high miss coverage for L1-I. However, exploiting an instruction streaming mechanism that

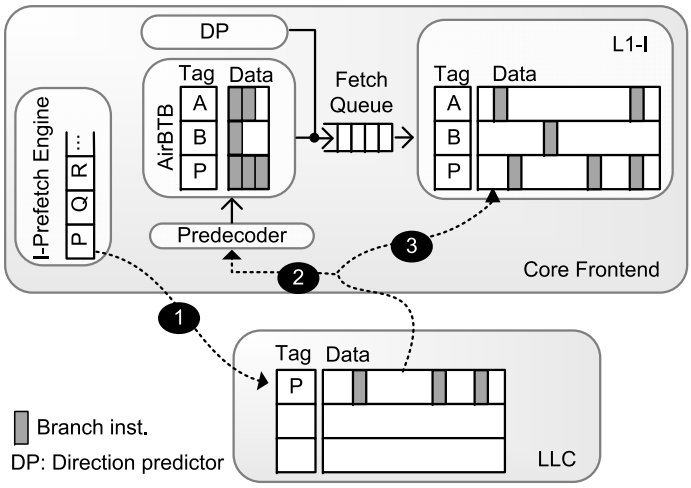

Figure 4.3: Core frontend organization and instruction flow.

tracks control flow at the block granularity for filling the BTB requires rethinking the BTB organization. To that end, we introduce AirBTB, a lightweight BTB design whose content mirrors that of the L1-I, thus enabling a single control flow history to be used for streaming into both structures.

Although instruction streaming mechanisms, which maintain history at block granularity, effectively capture the control flow history at coarse granularity, they lack the fine-grain branch information required to predict future BTB misses. To overcome the granularity mismatch between cache blocks and individual branches, Confluence exploits spatial locality within instruction blocks (i.e., the likelihood of multiple branches instructions being executed and taken in a block) by eagerly inserting all of the BTB entries of a block into AirBTB upon the arrival of a block at the L1-I.

As shown in Figure 4.3, Confluence synchronizes the insertions and the evictions into AirBTB with the L1-I, thus guaranteeing that the set of blocks present in both structures is identical. As the blocks are proactively fetched from lower levels of the cache hierarchy by the prefetch engine (step 1), Confluence generates the BTB metadata by predecoding the branch type and target displacement field encoded in the branch instructions in a block and inserts the metadata into AirBTB (step 2) and the instruction block itself into the L1-I (step 3).

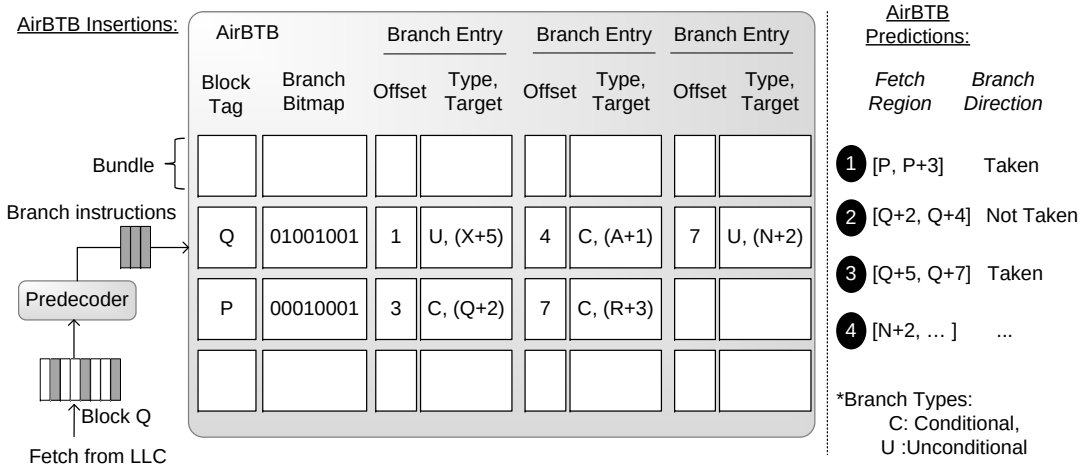

Figure 4.4: AirBTB organization.

In the rest of this section, we first describe the AirBTB organization, the insertion and replacement operations in AirBTB and how AirBTB operates within the branch prediction unit. Then, we briefly describe how SHIFT orchestrates both the AirBTB and instruction content. Finally, we explain why block-grain temporal streams are a better predictor of BTB misses compared to the history of individual branches.

4.2.1 AirBTB Organization

AirBTB is organized as a set-associative cache. Because AirBTB's content is in sync with the L1-I, as shown in Figure 4.3, AirBTB maintains a *bundle* for each block in L1-I. Each bundle comprises a fixed number of *branch entries* that belong to the branch instructions in a block.

In a conventional BTB design, each entry for a branch instruction (or basic block entry) is individually tagged, and hence necessitates maintaining a tag for each individual entry. Because the branches in a bundle in AirBTB belong to the same instruction block, the branch addresses share the same high-order bits, which constitute the address of the block. To exploit the commonality of high-order bits of the branch instruction addresses in a bundle, AirBTB maintains a single tag for a bundle, which is the instruction block address that contains the branches. We refer to this organization as *block-based organization*. The block-based

organization amortizes the tag cost across the branches in the same block. Moreover, the block-based organization avoids conflict misses between the branch entries that belong to two different blocks resident in the L1-I.

Figure 4.4 depicts the AirBTB organization, where each bundle is tagged with the block address and contains entries of three branches, which fall into the same cache block. The *branch bitmap* in each bundle is a bit vector that identifies the branch instructions in an instruction block. The branch bitmap maintains the knowledge of basic block boundaries within a block, allowing for providing the instruction fetch unit (L1-I), with multiple instructions to fetch in a single lookup. Each branch entry in a bundle contains the offset of the branch instruction within the cache block, the branch type (i.e., conditional, unconditional, indirect, return) and the branch target address (if the branch is a relative branch, which is mostly the case).

Because each bundle maintains a fixed number of branch entries, L1-I blocks with more branch instructions can overflow their bundles. Such overflows happen very rarely if bundles are sized correctly to accommodate all the branches in a cache block in the common case. To handle overflows, AirBTB is backed with a fully-associative *overflow buffer* consisting of a fixed number of entries. Each entry in the overflow buffer is tagged with full branch instruction address and maintains the branch type and target address. The branch bitmap in a bundle also keeps track of the the branch entries in a block that overflowed to the overflow buffer.

4.2.2 AirBTB Insertions and Replacements

To provide the synchronization with the L1-I, Confluence inserts the branch entries of a block into AirBTB upon the insertion of the block into the L1-I. By relying on spatial locality, Confluence inserts all the branch entries of a block eagerly into AirBTB. This way, Confluence overprovisions for the worst case where the branch prediction unit might need every branch entry within a block, even though the control flow might diverge to a different block before all the entries in the current block are used by the branch prediction unit.

For each block fetched into the L1-I, Confluence necessitates identifying the branch instruc-

tions in a block, extracting the type and relative displacement field encoded in each branch instruction. Confluence relies on predecoding to generate the BTB metadata of the branches in the block before the block is inserted into the L1-I. The predecoder requires a few cycles to perform the branch scan within a cache block before the block is inserted into the cache [14, 70]. However, this latency is not on the critical path if the block is fetched into the L1-I earlier than it is needed with the guidance of the instruction prefetcher.

As shown in Figure 4.4 on the left-hand side, for each instruction block fetched into the L1-I, Confluence creates a new bundle and inserts the branch entries into the bundle, while setting the bits of the corresponding branches in the branch bitmap, until the bundle becomes full. If the block overflows its bundle, the entries that cannot be accommodated by the bundle are inserted into the overflow buffer, while their corresponding bits are also set in the bitmap.

Upon the insertion of a new bundle due to a newly fetched instruction block, the bundle evicted from AirBTB belongs to the instruction block evicted from the L1-I. This way, AirBTB maintains only the entries of the branch instructions resident in the L1-I.

4.2.3 AirBTB Operation

Every lookup in AirBTB, in cooperation with the branch direction predictor, provides a *fetch region*, the addresses of the instructions starting and ending a basic block, to be fetched from the L1-I. In this section, we explain how AirBTB performs predictions in collaboration with the direction predictor in detail.

Figure 4.4(the right-hand side) lists the predictions made step by step. Let's say the instruction stream starts with address *P*. AirBTB first performs a lookup for block *P* and, upon a match, identifies the first subsequent branch instruction that comes after instruction *P* by scanning the branch bitmap. In our example, the first branch instruction after *P* is the instruction at address *P+3*. The fetch region, *P* to *P+3*, is sent to the instruction fetch unit and the target address for the branch instruction *P+3* is read out. Next, a direction prediction is made for the conditional branch at address *P+3* by the direction predictor and a lookup is performed for

P+3's target address *Q+2* in AirBTB. Because the conditional branch is predicted taken, the next fetch region provided by the target address' bundle, *Q+2* to *Q+4*, is sent to the fetch unit. Then, because the conditional branch *Q+4* is predicted not taken, the next fetch region is *Q+5* to *Q+7*.

If a branch is a return or indirect branch, the target prediction is made by the return address stack or indirect target cache respectively. If a branch indicated by the branch bitmap is not found in one of branch entries in the bundle, AirBTB performs a lookup for that branch instruction in the overflow buffer. The rest of the prediction operation is exactly the same for branch entries found in the overflow buffer.

If AirBTB cannot find a block or a branch entry indicated by a branch bitmap (because the entry was evicted from the overflow buffer), it speculatively provides a fetch region consisting of a predefined number of instructions following the last predicted target address, until it is redirected to the correct fetch stream by the core.

4.2.4 Prefetcher Microarchitecture

Providing the pipeline with a continuous stream of useful instructions to execute necessitates the branch predictor to be highly accurate. For BTB, accuracy corresponds to being able to identify all the branches and provide their targets as the branch prediction unit explores the future control flow. If the branch prediction unit does not identify instructions as branches because they are not present in the BTB, it speculatively provides the fetch unit with sequential fixed-size fetch regions, which become misfetches if there are actually taken branches in those sequential fetch regions. To avoid such misfetches, AirBTB requires a mechanism to predict the future control flow, so that it can eagerly insert the branch entries that will be needed soon.

The key enabler of AirBTB with a high hit ratio is an effective and accurate instruction prefetcher as AirBTB leverages the instruction prefetcher to populate its limited storage with branch entries that are likely to be referenced soon. Confluence leverages SHIFT, which amortizes its history storage cost across many cores running the same workload as described in

Chapter 3.

SHIFT consists of two components to maintain the history of instruction streams; the history buffer and the index table. The history buffer maintains the history of the L1-I access stream generated by one core at block granularity in a circular buffer and the index table provides the location of the most recent occurrence of an instruction block address in the circular buffer for fast lookups. The content of these two components are generated by only one core and used by all cores running a common server workload in a server CMP. To enable sharing and eliminate the need for a dedicated history table, the history is maintain in the LLC leveraging the virtualization framework [13].

A miss in the L1-I initiates a lookup in the index table to find the most recent occurrence of that block address in the history buffer. Upon a hit in the index table, the prefetch engine fetches prediction metadata from the history buffer starting from the location that is pointed by the index table entry. The prefetch engine uses this metadata to predict future L1-I misses, thus prefetches the instruction blocks whose addresses are in the metadata. As predictions turn out to be correct (i.e., the predicted instruction blocks are demanded by the core), more block addresses are read from the metadata and used for further predictions.

4.3 Methodology

4.3.1 Baseline System Configuration

We simulate a sixteen-core CMP running server workloads using Flexus [88], a Virtutech Simics-based full-system multiprocessor simulator. We use the Solaris operating system and run the server workloads listed in Table 2.1.

We run trace-based simulations for profiling, predictor accuracy and miss coverage studies by using traces with 16B instructions (1B instructions per core) in the steady state of the server workloads. For the DSS queries, we use the traces of the full query executions. Our traces consist of both application and operating system instructions.

Processing Nodes	UltraSPARC III ISA, Cortex A15-like (4.5 mm^2 in 40nm technology) 3GHz, 3-way OoO, 128-entry ROB, 32-entry LSQ
Branch Prediction Unit	Hyrid branch predictor: (16K-entry gShare, Bimodal, Meta selector) 1K-entry indirect target cache, 64-entry return address stack 1 branch per cycle
L1 I&D Caches	32KB, 4-way, 64B blocks 2-cycle load-to-use latency, 8 MSHRs
L2 NUCA Cache	Unified, 64B blocks, 512KB per core, unified, 16-way, 64 MSHRs *NUCA*: 1 bank per tile, 5-cycle hit latency *UCA*: 1 bank per 2 cores, 6-cycle hit latency
Interconnect	*Mesh*: 4x4 2D, 3 cycles per hop *Crossbar*: 5 cycles
Main memory	45ns access latency

Table 4.1: Parameters of the server architecture evaluated.

For performance comparison, we leverage the SimFlex multiprocessor sampling methodology [88] extending the SMARTS sampling framework [89]. The samples are collected over 10-30 seconds of workload execution (from the beginning to the completion of each query for DSS queries). The cycle-accurate timing simulation for each measurement point starts from a checkpoint with warmed architectural state (branch predictors, caches, memory, and prefetcher history), and then, runs 100K cycles in the detailed cycle-accurate simulation mode to warm up the queues on on-chip interconnect. The reported measurements are collected from the subsequent 50K cycles of simulation for each measurement point. Our performance metric is the ratio of number of application instructions retired to the total number of cycles (including the cycles spent executing the operation system instructions) as this metric has been shown to accurately represent the overall system throughput [88]. We compute the performance measurements with an average error of less than 5% at the 95% confidence level.

We model a tiled server processor architecture whose architectural parameters are listed in Table 4.1. We also model a CMP with a crossbar on-chip interconnect. Today's commercial processor cores typically comprise 3-5 fetch stages followed by several decode stages [14, 48, 70, 84]. Similarly, we model a core with three fetch stages and fifteen stages in total. The branch prediction unit is decoupled from the fetch unit with a fetch queue of six basic blocks [64]. The branch prediction unit outputs a fetch region every cycle and enqueues the fetch region into

the fetch queue to be consumed by the L1-I. Upon a miss in the BTB, a predefined number of instructions (eight, in our case) subsequent to the last fetch address predicted by the BTB are enqueued in the fetch queue as the next fetch region.

The branch instructions are identified right after the fetch stage, in the first decode stage. If a branch instruction was not identified by the branch prediction unit because of the missing BTB entry (i.e., branch misfetch), the instruction fetch stream can be redirected in the first decode stage. If a branch is conditional and has a relative target address, the direction predictor is looked up. If the conditional branch instruction is predicted taken, the fetch stream is redirected to the target address calculated in the first decode stage. If the branch is unconditional and has a relative target, the fetch stream is redirected to the target address calculated in the first decode stage. Finally, if the branch instruction is an indirect branch or a return instruction, the fetch stream is redirected to the target address predicted by the indirect target address or the return address stack.

4.3.2 Instruction-Supply Mechanisms

We compare Confluence (AirBTB coupled with SHIFT) against fetch-directed prefetching leveraging three different BTB designs, namely the Idealized BTB with 1-cycle access latency and no misses, realistic conventional two-level BTB, and PhantomBTB as a two-level BTB design allowing for sharing the second-level BTB across cores. We also couple SHIFT with these three different BTB designs and compare with AirBTB to decouple the effects of instruction prefetching and BTB design. The area overheads of these different design points are determined based on Cacti 6.5 [55] in 40nm technology.

Instruction Prefetchers

Shared History Instruction Fetch (SHIFT): SHIFT, described in detail in Section 4.2.4, tuned for our workloads for maximum L1-I miss coverage, requires a 32K-entry history buffer (160KB) virtualized in the LLC and around 240KB of index storage embedded in the tag array of the

LLC. The resulting per-core area overhead stemming from extra LLC storage and extension of the tag array is estimated to be 0.8 and $1.2mm^2$ respectively. This corresponds to $0.12mm^2$ area overhead per core.

Fetch-Directed Prefetching (FDP): The branch prediction unit is decoupled from the L1-I with a queue that can accommodate six basic blocks (determined experimentally to maximize performance) and the branch prediction unit outputs a basic block every cycle, as described above. For each fetch region enqueued in the fetch queue, prefetch requests are issued for the instruction blocks that fall into the fetch region, if they are not already in the L1-I. Because FDP relies on the existing branch predictor metadata, it does not incur any additional storage overhead.

BTB Designs

AirBTB: The final AirBTB design maintains 512 bundles in total (same as the number of blocks in the L1-I) and 3 branch entries per bundle. Because the instruction size is 4B, each 64B cache block has 16 instructions in total. So, each bundle keeps a 16-bit branch bitmap. Each branch entry has a 4-bit offset, 2-bit branch type, and 30-bit target field. The overflow buffer has 32 entries. Overall, the final AirBTB design requires 10.5KB of storage, incurring $0.08mm^2$ area overhead per core (AirBTB's sensitivity to design parameters is evaluated in Section 4.4.3). AirBTB's footprint is comparable in sizes to a 1K-entry conventional BTB with a victim buffer and to the first-level of PhantomBTB. In total, Confluence, AirBTB($0.08mm^2$) backed by SHIFT ($0.12mm^2$), incurs $0.2mm^2$ area overhead per core.

Conventional BTB: To provide multiple instructions to be fetched with a single BTB lookup to feed superscalar cores, prior work proposed organizing the BTB to provide a fetch range (i.e., basic-block range) [64, 91]. Each BTB entry is tagged with the starting address of a basic block (excluding the low-order bits used for indexing) and maintains the target address of the branch ending the basic block (30-bit PC-relative displacement; the longest displacement field in the UltraSPARC III ISA), the type of the branch instruction ending the basic block (2

bits), and a number of bits to encode the fall-through address (the next instruction after the branch ending the basic block). We found that the fall-through distance can be encoded with four bits for 99% of the basic blocks. We assume a 52-bit virtual address space and evaluate a four-way set associative BTB organization. We did not see any additional benefits in hit rate from increasing the associativity. In our comparisons of conventional BTB with AirBTB, we augment the conventional BTB with a 64-entry victim buffer (around 0.7KB). So, the 1K-entry conventional BTB used as the baseline requires 10.4 KB of storage in total ($0.08mm^2$) and has 1-cycle access latency. For the conventional two-level BTB configuration, we use a 16K-entry conventional BTB as the second level, which is 148KB and occupies $0.6mm^2$ area per core and has 4-cycle access latency.

PhantomBTB: In the PhantomBTB design tuned for our benchmark suite to maximize the percentage of misses eliminated and performance, we use a 1K-entry conventional first-level BTB with a 64-entry prefetch buffer (10.4 KB in total and $0.08mm^2$). PhantomBTB's first-level table storage cost is the same as AirBTB. For the virtualized prefetcher history, we pack six BTB entries (the maximum possible) in an LLC block and dedicate 4K LLC blocks (256KB assuming 64B blocks, with an estimated aggregate area overhead of $1.2mm^2$). We did not see any significant change in the percentage of misses eliminated with a bigger history. The region size used to tag each temporal group (i.e., LLC block) is 32 instructions.

Although the original PhantomBTB design maintains a private prefetcher history per core, we evaluate a shared prefetcher history as we run homogeneous server workloads where each core runs the same application code, thus is amenable to BTB sharing. This enables a fair comparison between PhantomBTB and Confluence, as Confluence relies on shared history for instruction prefetching. It is important to note that sharing the prefetcher history has negligible (i.e., less than 2%) effect on the percentage of misses eliminated by PhantomBTB as compared to the private prefetcher history that was originally proposed. As a result the aggregate storage overhead per core is $0.15mm^2$.

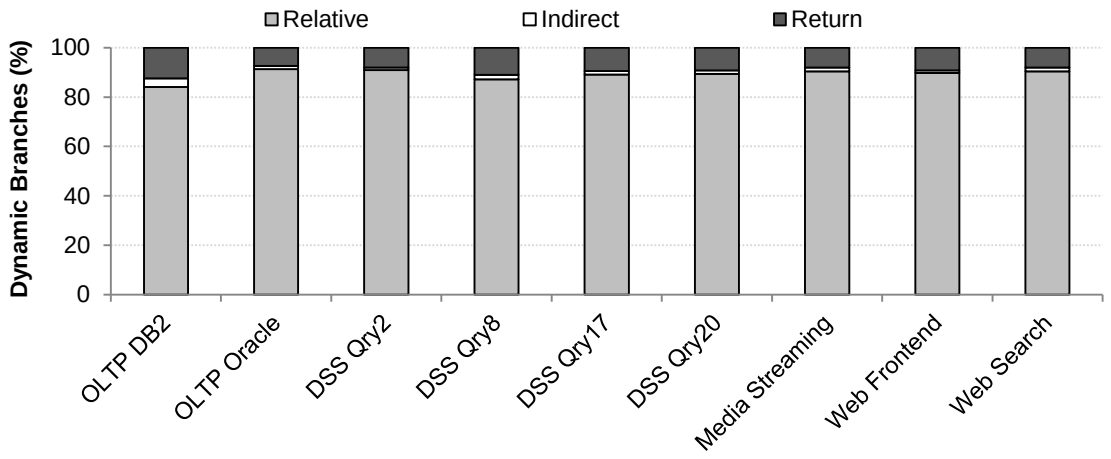

Figure 4.5: Distribution of dynamic branches according to their target address types.

4.4 Evaluation

Confluence unifies the disparately maintained instruction-supply metadata for BTB and L1-I by relying on an instruction streaming mechanism, SHIFT, whose effectiveness is already demonstrated for L1-I. For that reason, in this section, we focus on the benefits achieved with our light-weight BTB design, AirBTB, operating in sync with the L1-I backed by the effective instruction streaming mechanism. However, we quantify the performance benefits of Confluence stemming from synergy between AirBTB and the instruction streaming mechanism in Sections 4.4.6- 4.4.8.

4.4.1 Distribution of Branch Types

While AirBTB is responsible for identifying the branches along with their types, it is also responsible for providing the target addresses of branches with static targets. Because AirBTB relies on predecoding to generate the target addresses of branches when they are fetched into the instruction cache from the lower levels of the cache hierarchy, the more PC-relative branches (i.e., branch instructions whose target addresses are generated adding the branch PC with the displacement field embedded in the instruction) there are, the more branch targets AirBTB can predict for taken branches. For other branches that are not PC-relative (i.e.,

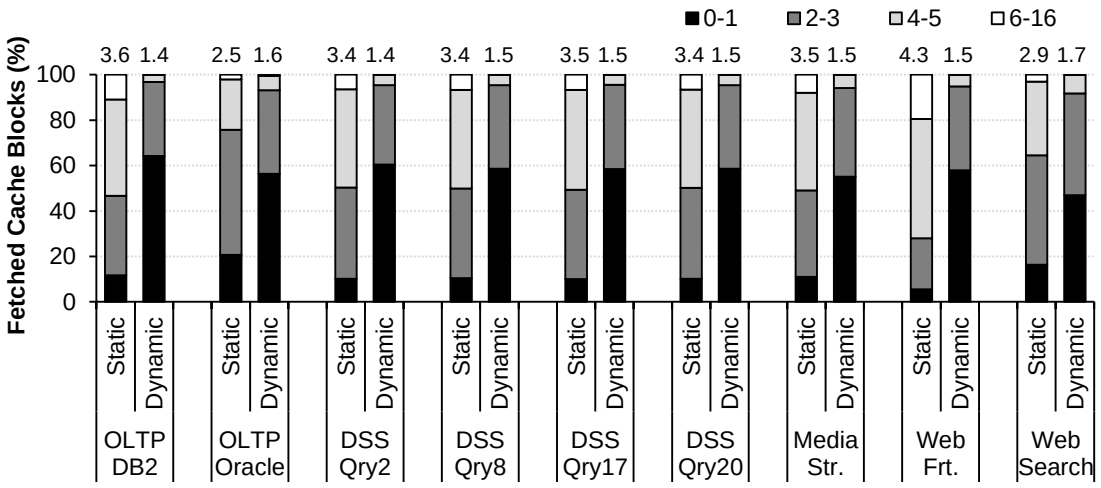

Figure 4.6: Density of static and dynamic branches in the demand-fetched instruction cache blocks (The numbers on top of the bars represent the average values).

indirect branches and returns), AirBTB cooperates with the indirect branch target cache or the return address stack.

Figure 4.5 shows the distribution of branches executed based on their classes. The PC-relative branches constitute 90% of the retired branch instructions on average allowing AirBTB to provide most of the target addresses for taken branches without requiring a large indirect branch target cache or return address stack.

4.4.2 Branch Density in Cache Blocks

AirBTB maintains bundles of branch entries that belong to the same instruction block to exploit spatial locality both for high miss coverage and storage savings. To achieve a high hit ratio, AirBTB needs to accommodate all of the branch entries in a cache block in its corresponding bundle in the common case and maintain an overflow buffer for the instruction blocks that exceed the capacity of a bundle.

Figure 4.6 presents the total number of branch instructions in a demand-fetched instruction cache block (i.e., *static*) as well as the total number of branches actually executed (and taken) during the residency of the cache block in the L1-I (i.e., *dynamic*).

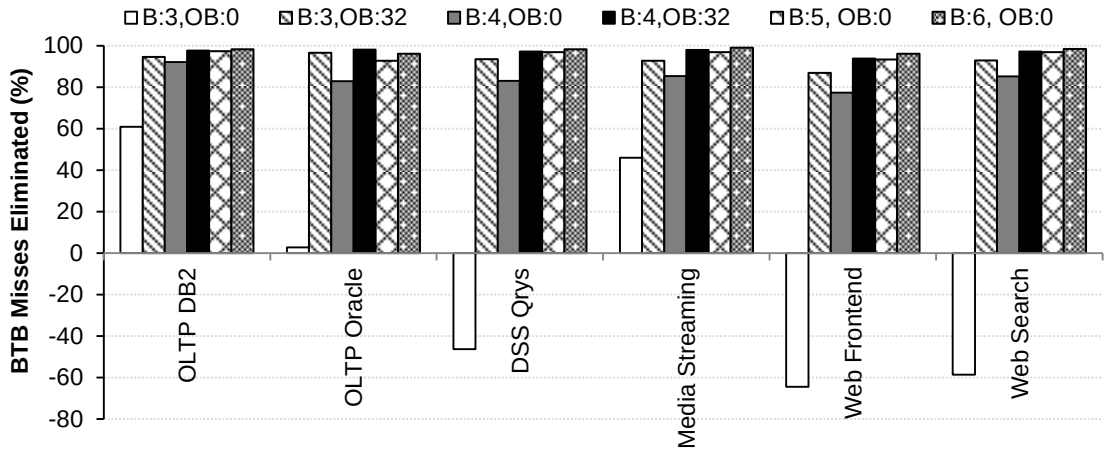

Figure 4.7: Miss coverage for various AirBTB configurations over a 1K-entry conventional BTB (B = branch entries in a bundle, OB = branch entries in the overflow buffer).

Demand-fetched instruction blocks typically contain 3-4 branch instructions on average out of which only 1-2 are actually executed and taken, because the control flow is redirected to another instruction block before reaching the rest of the branches in the current instruction block. While a bundle size of 3-4 entries would be sufficient for most instruction blocks to accommodate their branch entries, there is also a need for a sufficiently sized overflow buffer to accommodate the branches of instruction blocks that exceed the capacity of their corresponding bundle (i.e., overflows).

4.4.3 AirBTB Miss Coverage

In light of the branch behavior characterization findings presented in Section 4.4.2, we seek to find the optimal AirBTB configuration to achieve the highest BTB miss coverage with minimum storage overhead. To do so, we examine different AirBTB configurations by varying the bundle size (i.e., the number of branch entries in a bundle) and the size of the overflow buffer. The total number of bundles is fixed as AirBTB maintains only the instruction blocks resident in the instruction cache at a given time. Figure 4.7 shows AirBTB's miss coverage over the 1K-entry conventional BTB with a 64-entry victim cache. We show only the average values for the four DSS queries due to space constraints.

As the number of branch entries in a bundle increases, the overall miss coverage increases for all workloads as each bundle can accommodate more entries. Figure 4.6 shows that 50% of instruction blocks contain up to three branches on average. As a result, an AirBTB configuration with three branch entries per bundle is able to capture the branch footprint of half of the instruction blocks at a given point in time. Unfortunately, such an AirBTB configuration (B:3, OB:0) has a higher miss rate than the baseline 1K-entry BTB for some of the workloads (i.e., increases the miss rate by 18% on average), because it cannot maintain the overflowing branches.

To capture the overflowing branches, there is a need to increase the bundle size or accommodate the overflowing branches in the overflow buffer. When AirBTB with three branch entries per bundle is backed with an overflow buffer (B:3, OB:32), it becomes effective at eliminating the misses as compared to a 1K-entry conventional BTB. For the AirBTB configuration with three branch entries per bundle, we found the optimal overflow buffer size to be 32 entries. We did not see any improvement in coverage beyond 32 overflow buffer entries. Such a configuration (B:3, OB:32) provides 93% miss coverage on average.

Similarly, the AirBTB configuration with four branch entries in a bundle without an overflow buffer (B:4, OB:0) achieves 77% of miss coverage. The overflow buffer with 32-entries (B:4, OB:32) further improves the coverage to 95% on average. However, compared to the AirBTB configuration (B:3, OB:32), both of these design points (B:4, OB:0 and B4, OB:32) require more storage (around 2KB), while increasing the miss coverage by only 2%. For this reason, we use the AirBTB configuration with three branch entries per bundle and 32-entry overflow buffer (B:3, OB:32) as the final AirBTB design and for the rest of the evaluation. This configuration requires the same overall storage as the conventional 1K-entry BTB with a 64-entry victim cache, which we use as the baseline comparison point in the rest of this section.

Finally, among the AirBTB design points that do not employ an overflow buffer, (B:5, OB:0) and (B:6, OB:0) achieve 3% and 4% higher miss coverage on average compared to the final AirBTB configuration (B:3, OB:32) with the cost of extra dedicated storage due to the use of bigger

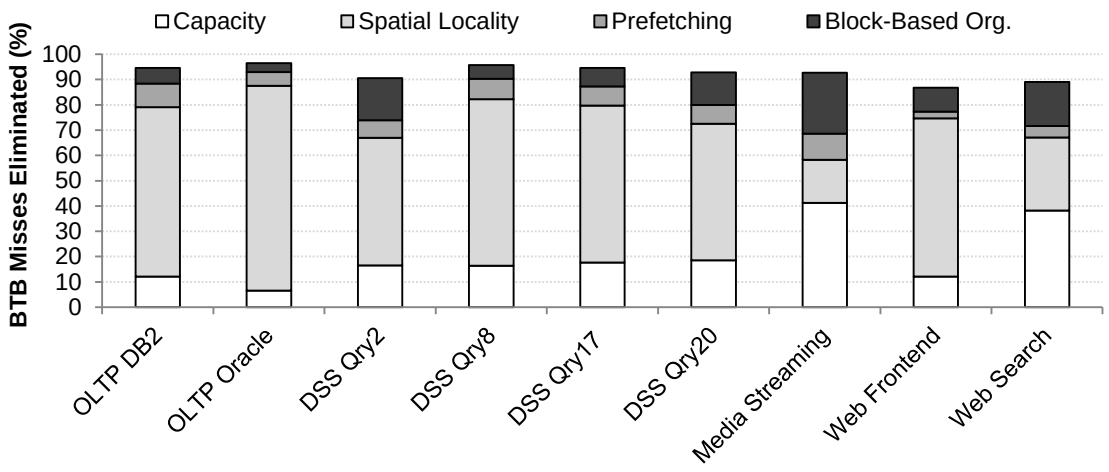

Figure 4.8: Breakdown of AirBTB miss coverage benefits over 1K-entry conventional BTB.

bundle sizes. In these two design points, although the majority of the blocks underutilize the bundles by not occupying all the branch entries, the complexity of the overflow buffer is not incurred.

4.4.4 Dissecting the AirBTB Benefits

To eliminate most of the misses within a given BTB storage budget, we modified the baseline BTB design in several ways. Figure 4.8 shows how much each design decision helps to improve the miss coverage over a conventional BTB design with 1K entries by employing the mechanisms proposed for AirBTB step by step. First, AirBTB can afford more entries within in a given storage budget as compared to the 1K-entry conventional BTB, because it amortizes the cost of tags across the entries in the same instruction cache block, eliminating 18% of the misses (*Capacity*). Second, because AirBTB eagerly identifies the branch instructions in a block upon a BTB miss in an instruction block and eagerly installs their entries before the branches are actually executed, it can eliminate 57% more misses on average (*Spatial Locality*). Third, by relying on the instruction streaming mechanism, AirBTB can eliminate 7% more misses by eliminating a BTB miss even if the first instruction touched in a missing block is a branch (*Prefetching*). Finally, the block-based organization employed by AirBTB guarantees that the blocks in the BTB are in sync with the L1-I, so that the BTB entries of two L1-I-resident blocks

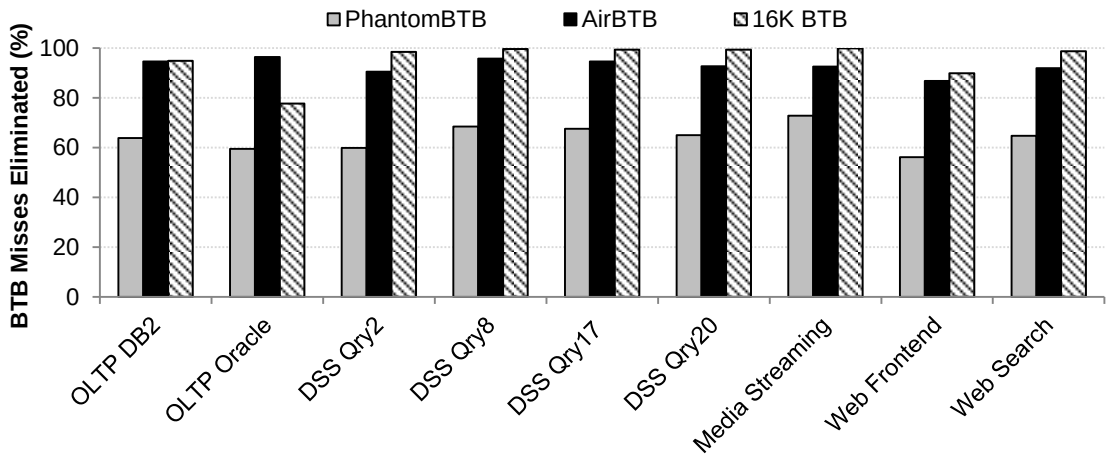

Figure 4.9: PhantomBTB, AirBTB and 16K-entry conventional BTB miss coverages over a 1K-entry conventional BTB.

do not conflict, which provides 11% additional miss coverage (*Block-Based Organization*).

It is important to note that, the coverage benefits of instruction streaming is only 7% in the trace-based simulation. However, the presence of an instruction streaming mechanism is absolutely necessary to hide the access latency of instruction blocks fetched from the lower levels of the hierarchy, which are required to generate the AirBTB content that will be required soon. Not hiding the long-latency accesses to lower levels of cache hierarchy would stall the branch prediction unit, and hence the entire pipeline.

4.4.5 Competitive BTB Miss Coverage Comparison

Figure 4.9 shows the fraction of BTB misses eliminated by AirBTB, PhantomBTB and a 16K-entry conventional BTB over the 1K-entry conventional BTB. PhantomBTB eliminates only 64% of the misses on average, compared to AirBTB's 93% miss coverage. The discrepancy in the coverage is attributed to two major differences between the two designs.

First, because AirBTB amortizes the cost of the tags across the branch entries within the same instruction cache block, it can maintain more BTB entries as compared to PhantomBTB's first-level BTB, which is a conventional BTB organization, within the same storage budget.

More importantly, PhantomBTB forms temporal groups of BTB entries that consecutively miss in the first-level BTB by packing a number of BTB entries in an LLC block and prefetching those entries into the first-level BTB upon a miss in the first-level BTB. In PhantomBTB, the BTB entries that fall into a temporal group depend heavily on the branch outcomes in the local control flow. Small divergences in the control flow significantly affect the content of the temporal groups and reduce the likelihood of same sets of branches always missing in the BTB together. Moreover, because PhantomBTB maintains fixed-sized temporal groups of BTB entries, as opposed to arbitrary-length temporal streams as in SHIFT, its prefetch lookahead is limited to only a few BTB entries upon each L1-BTB miss.

In contrast, the stream-based prefetcher leveraged by Confluence is a better predictor of future control flow as it relies on a coarse-grain temporal streams of instruction block addresses that often cover as many as a few hundred instruction blocks per stream. Hence, the stream prefetcher's highly accurate control flow prediction at the macro level coupled with AirBTB's eager insertion and block-based organization (which uncovers spatial locality) provides a higher miss coverage than PhantomBTB. We conclude by noting that the coverage reported for PhantomBTB is the highest coverage we could attain and does not benefit from further increases in the size of its history storage.

Overall, AirBTB closely approaches the miss coverage of the 16K-entry conventional BTB, which provides 95% miss coverage on average, without incurring its high per-core storage overhead.

4.4.6 Competitive Performance Comparison

We compare Confluence's (AirBTB backed by SHIFT) performance against PhantomBTB and 16K-entry conventional BTB (with only one-cycle access latency) normalized to the performance of the 1K-entry conventional BTB in Figure 4.10. It is important to note that all the other designs employ SHIFT only for instruction streaming, while Confluence takes advantage of SHIFT to stream the BTB metadata into AirBTB.

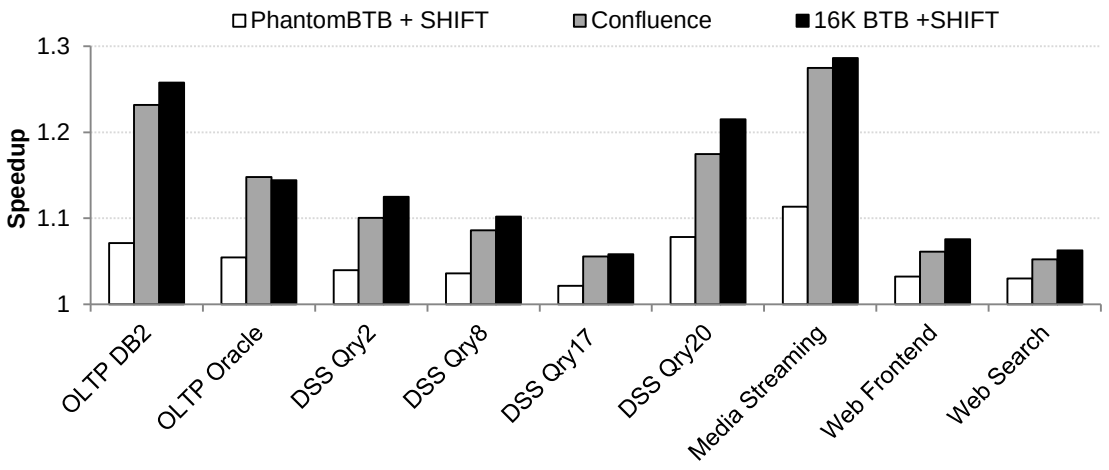

Figure 4.10: PhantomBTB, Confluence and 16K-entry conventional BTB speedup comparison over 1K-entry conventional BTB with SHIFT for a 16-core CMP with a mesh on-chip interconnect.

PhantomBTB provides only 6% performance improvement on average, while Confluence provides 14% performance improvement closely approaching the performance improvement attained by the 16K-entry conventional BTB, which is 15.5%. The difference in speedups of PhantomBTB and Confluence stems from two main differences.

First, PhantomBTB provides around 30% lower miss coverage than AirBTB, thus Confluence, over the baseline 1K-entry conventional BTB as shown in Figure 4.9.

Second, PhantomBTB's timeliness dramatically lags behind that of Confluence's, because PhantomBTB fetches only an LLC block worth of BTB entries (i.e., six BTB entries in a temporal group) upon a miss in a particular code region. To fetch the next temporal group, PhantomBTB waits until there is a miss in another code region. Waiting for a miss in a code region to prefetch only a limited number of entries in a temporal group prevents PhantomBTB from hiding the latency of the LLC from the branch prediction unit.

Confluence, on the other hand, relies on temporal streams with arbitrary lengths (on the order of hundreds of cache blocks) provided by SHIFT enjoying a high lookahead. The high lookahead provided by temporal streams helps Confluence to install the BTB entries that will be accessed in near future before they are actually needed by the branch prediction unit. As a

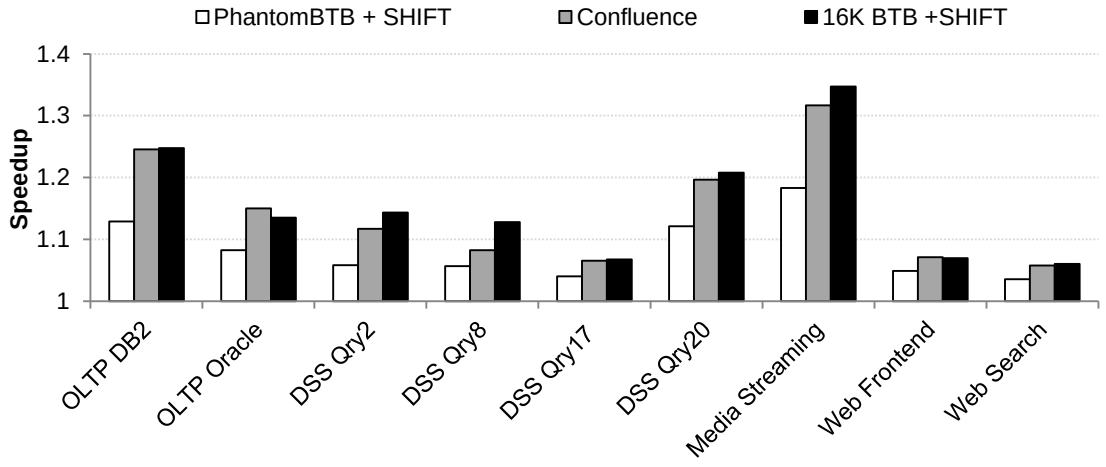

Figure 4.11: PhantomBTB, Confluence and 16K-entry conventional BTB speedup comparison over 1K-entry conventional BTB with SHIFT for a 16-core CMP with a crossbar on-chip interconnect.

result, Confluence achieves 99% of the performance provided by the 16K-entry conventional BTB with one-cycle access latency. Bridging the remaining 1% performance gap would be possible with an instruction prefetcher with a higher coverage than SHIFT (i.e., SHIFT can eliminate 85% of L1-I misses on average).

Overall, Confluence achieves 85% of the performance improvement provided by a perfect BTB and L1-I.

We also evaluated Confluence's performance improvements for a CMP with a crossbar on-chip interconnect as shown in Figure 4.11. We expect the crossbar interconnect to mitigate instruction and data stalls compared to a mesh interconnect, exposing the performance degradation due to BTB misses as a bigger portion of the execution time, thus resulting in higher performance improvements for Confluence. However, because the majority of the instruction-fetch stalls are eliminated by SHIFT, and a significant portion of the latency of accessing the LLC for data can be hidden by the OoO cores, Confluence provides only slightly higher performance benefits in the case of the crossbar interconnect (on average 16% as compared to 14% in the mesh configuration). PhantomBTB also benefits from the reduction in the average LLC access latency provided by the crossbar interconnect as average latency of accessing the metadata in its LLC decreases. In the case of a crossbar interconnect, PhantomBTB provides 8% perfor-

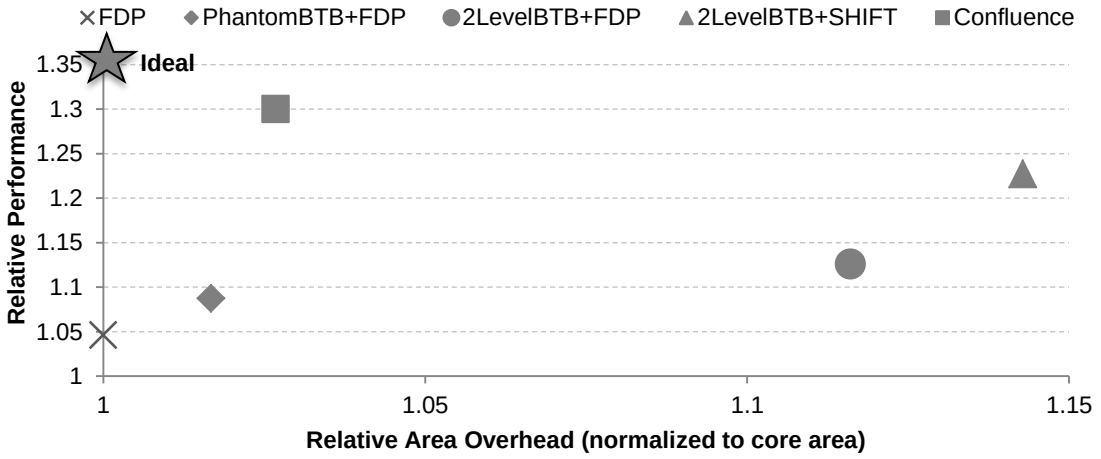

Figure 4.12: Confluence performance benefits and area savings compared to other design points.

mance improvement as opposed to 6% in the mesh interconnect case, but still lags behind Confluence in terms of performance improvement.

4.4.7 Performance and Area Comparison

We finally compare the performance benefits and associated area overheads of Confluence with various frontend designs discussed in Section 4.1 in Figure 4.12. All the performance and area numbers are normalized to a core with 1K-entry BTB to be consistent with Section 4.1.3. As Figure 4.12 demonstrates, Confluence is the closest design point to the Ideal by delivering 85% of the performance improvement delivered by the Ideal configuration (i.e., perfect L1-I and BTB) with only 3% storage area overhead per core (including private BTB's and SHIFT's per-core storage overhead). Confluence delivers higher performance as compared to all other design points detailed in Section 4.1.3 thanks to its timely and accurate insertions of instructions into the L1-I and branch entries into the BTB.

It is instructive to compare the 2LevelBTB+SHIFT and Confluence designs. Both feature a SHIFT instruction prefetcher and a high-accuracy BTB. Because of the redundant metadata in the 2LevelBTB+SHIFT design, Confluence achieves a considerably lower storage footprint (10KB per core for Confluence vs. 150KB per core for the 2LevelBTB). Performance-wise,

Confluence is 8% better, despite the fact that the 2LevelBTB+SHIFT delivers a slightly higher hit rate (detailed analysis in Section 4.4.4). The reason for Confluence's superior performance is timeliness, as the BTB is filled proactively ahead of the fetch stream. In contrast, decoupled BTB designs (including both 2LevelBTB and PhantomBTB) trigger BTB fills only when a miss is discovered in the first-level BTB, thus exposing the core to the access latency of the second level.

4.4.8 Performance Benefits without Prefetching

Confluence provides high-performance instruction supply by (i) synchronizing the BTB and L1-I contents and (ii) streaming into both BTB and L1-I. Figure 4.8 quantifies the benefits of content synchronization. In this section, we quantify the absolute need for an effective instruction streaming mechanism to enable synchronization to achieve its full performance potential. To do that, we employ AirBTB without SHIFT, only by maintaining the AirBTB content in sync with the L1-I content. Without SHIFT, AirBTB provides only 2% performance improvement on average. The discrepancy stems from AirBTB not being able to run ahead to provide the pipeline with a useful instruction fetch stream, because it needs to wait for the arrival of the instruction blocks at the core, getting highly exposed to the LLC latency. This underscores the importance of backing AirBTB with an effective instruction streaming mechanism, which is what Confluence does.

4.5 Concluding Remarks

Large instruction working sets of server applications are beyond the reach of practical BTB and L1-I sizes due to their strictly low access-latency requirements. Frequent misses in BTB and L1-I result in frequent misfetches and instruction fetch stalls dramatically hurting the performance of server applications. In response, prior research proposed discrete prefetchers for BTB and L1-I, whose metadata essentially capture the same control flow exhibited by an application.

This work proposed Confluence, a new front-end design, which synchronizes the BTB and L1-I content to leverage a single prefetcher and unified prefetcher metadata to prefetch for both BTB and L1-I by relying on a highly accurate instruction prefetcher. By doing so, Confluence eliminates 93% of BTB misses and 85% of L1-I misses and provides 85% of the speedup possible with a perfect L1-I and BTB.

5 Related Work

5.1 Sharing Core Resources Across Hardware Contexts

Hardware multithreading has been proposed as a way of mitigating underutilization of core resources by executing multiple independent threads concurrently on the same core. Hardware multithreading intrinsically enables resource sharing across multiple hardware contexts including both the frontend and computation resources of a core [83]. In hardware multithreading, hardware contexts typically compete for cache, TLB and branch predictor capacity as these resources have limited capacities due to their low access-latency requirements, unless identical software threads are executing in lock-step at a given time (lock-step execution would not incur contention for frontend resources). Furthermore, the degree of sharing is limited by the multithreading degree of a core.

Seeking a balance between the CMP designs with low-complexity cores and SMT designs with high-complexity cores, Conjoined-core CMP [49] and Cash [22] enable sharing of resources such as the L1 caches, branch predictors and functional units across adjacent cores. Similar to hardware multithreading, the degree of sharing in such architectures is limited to a few cores, as larger clusters of cores sharing resources would not benefit from sharing due to the increasing physical distance, thus the increasing latency of accessing the shared resources. Furthermore, the contention for the shared resources necessitates using larger

shared resources.

As opposed to limited degree of sharing that can be provided by hardware multithreading and sharing resources across adjacent cores, our shared frontend framework allows for sharing of resources across all the hardware contexts in a CMP independently of the core microarchitecture and the physical distance between the cores. By providing each core with dedicated storage to store the metadata of the active instruction working set and filtering most accesses to shared metadata through the dedicated storage, our shared frontend framework minimizes the contention for shared metadata. Because the shared metadata storage typically captures the entire working set of an application, introducing additional hardware contexts does not affect the storage capacity requirement of the shared metadata as long as all hardware contexts execute the same given workload.

A line of work has proposed mechanisms to enable cooperation between cores running the same workload to improve TLB hit rates by exploiting the commonality of TLB miss patterns across cores [10]. Our work also allows for inter-core cooperation for prefetching, however, with the goal of eliminating the storage overheads of existing prediction mechanisms.

5.2 Instruction Prefetching

5.2.1 Next-Line Prefetchers

Instruction-fetch stalls have long been recognized as a dominant performance bottleneck in servers [2, 16, 26, 38, 45, 51, 63, 85]. Simple next-line instruction prefetchers have been ubiquitously employed in commercial processors to eliminate misses to subsequent blocks [3, 72]. Despite being ineffective for misses to non-sequential blocks, next-line prefetchers are critical to server performance [63]. Because only one prefetch request is issued upon a miss in the cache, next-line prefetchers highly suffer from insufficient lookahead.

More recent work has extended next-line prefetching to prefetch sequences of contiguous instruction blocks with arbitrary lengths [62, 67]. Despite their zero storage overhead, next-line

prefetchers fail to eliminate instruction cache misses due to discontinuities in the program control flow caused by function calls, taken branches and interrupts.

SHIFT, on the other hand, records all the cache accesses including misses to sequential blocks as well as to non-sequential blocks.

5.2.2 Discontinuity Prefetcher

To predict misses to non-sequential blocks, the discontinuity prefetcher [76] maintains the history of transitions between two of non-sequential cache blocks that consecutively miss in the cache. Each entry in the history maps one instruction block to another instruction block.

The discontinuity prefetcher provides only marginal benefits because of several fundamental limitations. First, it cannot capture all the transitions for blocks with multiple branch instructions as different branches are likely to have different targets. Second, the lookahead of the discontinuity prefetcher is limited to one target instruction block for each source block.

In contrast, in SHIFT, a block might be associated with many transitions in the history depending on the context. Unlike the discontinuity prefetcher, the instruction streaming mechanism, SHIFT, does not have a lookahead limitation while replaying temporal streams with arbitrary lengths.

5.2.3 Branch-Predictor-Directed Prefetchers

A class of instruction prefetchers rely on the existing branch predictor running ahead of the fetch unit to predict instruction cache misses along the way capturing both sequential and non-sequential blocks [20, 65, 86]. The branch-history guided prefetcher [77], execution-history-guided prefetcher [92], multiple-stream predictor [67], next-trace predictors [36], call-graph prefetching [5] and Efetch [18] maintain disparate history to essentially maintain the branch outcomes creating discontinuities to make predictions. The branch-predictor-guided prefetchers are limited by the accuracy and lookahead of the branch predictor.

In contrast to branch-predictor-guided prefetchers, due to the history maintained at instruction-block granularity, SHIFT is not affected by the minor divergences in the control flow within individual blocks and exploration of the local control flow spanning cache-resident instruction blocks.

Wrong-path prefetcher [60] prefetches the instructions along the not-taken path of conditional branches at the cost of generating extra traffic in case the wrong-path instructions are indeed never executed. Wrong-path prefetcher is typically effective only for backward loop exits or frequently executed data-dependent branches, but not for other types of discontinuities such as function calls.

5.2.4 Speculative-Thread Prefetchers

A line of research explores ways of leveraging the idle hardware contexts in CMPs or SMTs to speed up single-threaded applications [1, 78, 93]. To do so, these techniques spawn threads executing a speculative and shortened version of the program or upcoming critical regions of the program to generate the future state and fetch instructions and data that will be required by the main program, which runs slower than the helper thread. In a similar way, run-ahead execution [56] allows a program to run far ahead in the program path upon long-latency operations blocking the pipeline.

Similar to these techniques, SHIFT provides the instruction fetch unit with a speculative stream of likely-to-be-fetched instructions, which are recorded previously during the execution of a program. Instead of utilizing an idle core or hardware thread, SHIFT records the streams that are then used for speculatively fetching instructions.

5.2.5 Instruction Streaming

The first instruction streaming mechanism, TIFS [28], records and replays streams of discontinuous instruction cache misses, enhancing the lookahead of discontinuity prefetching. PIF [27], records the complete retire-order instruction cache access history, capturing both

discontinuities and next-line misses, without being affected by the microarchitectural noise introduced by the cache-replacement policy and wrong-path instructions like TIFS. As a result, PIF achieves superior miss coverage and timeliness compared to all other instruction prefetchers.

The SHIFT design adopts its key history record and replay mechanisms from previously proposed per-core data and instruction prefetchers [27, 28, 73, 87]. To facilitate sharing the instruction history, SHIFT embeds the history buffer in the LLC as proposed in predictor virtualization [13].

SHIFT maintains the retire-order instruction cache access history like PIF. Unlike prior instruction streaming mechanisms, SHIFT maintains a single shared history, allowing all cores running a common workload to use the shared history to predict future instruction misses.

RDIP [47] correlates instruction cache miss sequences with the call stack content. RDIP associates the history of miss sequences with a signature, which summarizes the return address stack content. In doing so, RDIP reduces the history storage requirements by not recording the entire instruction streams as in streaming mechanisms. However, RDIP's miss coverage is still limited by the amount of per-core storage. SHIFT, on the other hand, amortizes the cost of the entire instruction stream history across multiple cores, obviating the need for per-core history storage reduction.

5.2.6 Computation Spreading

A number of orthogonal studies mitigate instruction cache misses by exploiting code commonality across multiple threads [7, 19]. These approaches distribute the code footprint across private instruction caches of cores to leverage the aggregate on-chip instruction cache capacity. A thread migrates from one core to another core, whose private instruction cache accommodates the instructions that the thread is likely to execute in the next phase of its execution. Although these approaches are beneficial to reduce the instruction miss rate each thread is exposed to, the latency and other side effects (e.g., data cache misses) incurred when

a thread is migrated from one core to the other might offset the benefits of reduction in cache miss rate.

In a similar manner, SHIFT relies on the code path commonality, but it does not depend on the aggregate instruction cache capacity, which might be insufficient to accommodate large instruction footprints. Moreover, SHIFT supports multiple workloads running concurrently, while these techniques might lose their effectiveness due to the contention for instruction cache capacity in the presence of multiple workloads.

5.2.7 Batching & Time Multiplexing Similar Computations

Another way to exploit the code commonality across multiple threads is to group similar requests and time-multiplex their execution on a single core, so that the threads in a group can reuse the instructions, which are already brought into the instruction cache by the lead thread [8, 33, 81]. Unfortunately, these approaches are likely to hurt response latency of individual threads, as each thread is queued for execution and has to wait for the other threads in the group to execute.

5.2.8 Compiler-Based Techniques

A line of work aims to improve instruction cache behavior by mitigating the discontinuities in the dynamic instruction stream. Profile-guided code positioning [59] groups the basic blocks executed one after the other and splits the portions of procedures that are never executed based on run-time profiling of applications. These two techniques improve code density and increase the number of accesses to sequential instruction blocks. Some techniques leverage the compiler, linker and runtime information to organize the code layout to mitigate the conflicts between blocks in the instruction cache [34, 42, 80]. Procedure inlining also improves the code sequentiality at the cost of increasing the instruction working set size [61]. Because SHIFT does not depend on profiling, in contrast to these techniques, it immediately learns the execution sequences at run time and remains effective even in the case of changes in the

dataset and program behavior.

Instruction-prefetch instructions in an ISA allow for compiler-inserted instruction prefetches leveraging compiler's knowledge of instructions that are likely to miss in the cache [54]. Such techniques are limited in terms of lookahead as the compiler does not have much visibility across procedure boundaries. Compiler hints can also be leveraged to throttle the hardware next-line prefetcher to mitigate the pollution caused by erroneous next-line prefetches [54, 90].

5.3 Branch Target Buffer

Branch target buffer is the key component that allows the branch prediction unit to run ahead of the core and provide the core with a continuous instruction stream to be executed along with the branch direction predictor to avoid the bubbles in the pipeline that are caused by taken branch instructions.

5.3.1 Alternative BTB Organizations

In the early BTB designs, a BTB entry contains the branch instruction address and the branch target address to redirect control flow upon a predicted taken branch [50, 58, 79].

To increase the fetch bandwidth for wide superscalar cores and be able to fetch an entire basic block every cycle, Yeh and Patt proposed tagging BTB entries with the address of the instruction that starts a basic block and maintaining the fall-through address and the target instruction address of the branch terminating the basic block in each BTB entry [91]. Every cycle a direction prediction is made for the branch instruction terminating the current basic block (i.e., the instruction before the fall-through instruction). Based on the outcome of the direction prediction, the fall-through address or the target instruction address is used to perform a lookup in the BTB for the next basic block.

Reinman et al. proposed decoupling the branch predictor from the instruction cache with a fetch-target queue to mitigate the pipeline bubbles caused by the predictor tables with multi-

cycle access latencies [64]. The fetch target queue allows the branch predictor to run ahead and generate basic blocks to be consumed by the slower instruction cache by utilizing the small and fast first-level predictor table in the common case and the big and slower second-level table in the uncommon case. The buffering between the branch prediction unit and the fetch unit hides the latency of the slower table from the pipeline. Moreover, the fetch-target queue allows for instruction prefetching by exploring the basic-block addresses in the fetch-target queue that are not yet consumed by the instruction cache [65].

5.3.2 Next Cache Line and Set Prediction

Next cache line and set prediction (NLS) [15] has been proposed as an alternative to BTB and is employed in the Alpha EV8 processor [69]. NLS maintains pointers to the targets of branch instructions in the instruction cache instead of maintaining a target address per branch as the BTB does. When a branch instruction is read from the instruction cache, the instruction pointed by the branch instruction's predictor entry is used as the target instruction. NLS is likely to be effective for workloads with cache-resident instruction working sets, as the predictor maintains the predictions of branch and target instructions within the cache. For workloads with low instruction cache locality, thus high instruction miss rate, NLS frequently relearns branch-target instruction pairs every time an instruction block is evicted from and then fetched again into the cache. As a result, it is expected to achieve poor accuracy for workloads with large instruction working sets.

To mitigate the performance penalty associated with poor NLS accuracy, the Alpha EV8 processor [69] employs predecoding to scan the branches in the instruction cache blocks that are fetched into L1-I, precompute the target addresses of the branches, and modify the branch instructions to store the lower bits of their target addresses before they are inserted into the L1-I. This way, the target address of a taken branch is formed with a simple concatenation of the branch PC and the low order bits of the target address, right after the instruction is fetched from the L1-I. Processors featuring cores with hardware multithreading [70, 71] also employ the same mechanism to eliminate the BTB storage overhead and also to detect the branch

instructions and their targets (if they are predicted taken) as early as possible in the pipeline. In such a design, constructing the target addresses of branches in an instruction block read from the L1-I takes 3-4 cycles [71], significantly hurting single-threaded performance. This latency is tolerable in Alpha EV8, where branch direction prediction takes several cycles. Similarly, in the presence of multiple hardware contexts, the latency penalty associated with branch target prediction can be overlapped by fetching instructions from other hardware contexts. However, in cores employing branch direction predictors with only one-cycle latency and in the absence of multiple hardware threads to cover the latency, this scheme significantly hurts single-threaded performance by requiring several cycles after fetch to identify branches within a cache block and compute their targets. To mitigate the resulting fetch bubbles in the cases where there is only a single thread to execute, some processors employ small BTBs [71]; however, such designs still expose the core to the high latency of target computation whenever the small BTB misses.

5.3.3 Compression Techniques

Because the branch predictor is on the critical path, a large BTB with several cycles of access latency greatly penalizes the rate at which the instruction stream is delivered to the core. One way of reducing the capacity requirements of the BTB is to maintain fewer bits in the tag of a BTB entry instead of uniquely identifying a basic block with its full tag [25], making BTB entries susceptible to aliasing. Another way is to maintain only the offsets of the fall-through and target addresses from the basic-block address instead of their full addresses, since the distance between the basic-block address and the fall-through or target address is expected to be small [46, 64]. Although these compression techniques help to reduce the BTB capacity requirements to some extent, they cannot mitigate the number of individual entries that need to be maintained in the BTB to capture the entire instruction working set of an application, which is the fundamental problem for server workloads.

5.3.4 Hierarchical BTBs

To mitigate the access latency of large predictor tables, hierarchical branch predictors provide low access latencies with a smaller but less accurate first-level predictor in the common case and leverage a larger but slower second-level predictor to increase accuracy [14, 40, 69]. The second-level table overrides the prediction of the first-level table in case of disagreement at a later stage.

Although hierarchical BTBs are likely to achieve higher hit rates than that of small BTBs with low access-latency constraints, they still expose the core frontend to the access latency of the second-level BTB in the case of misses in the first-level BTB, preventing them from fully eliminating performance degradation due to misfetches as shown in Section 4.1.3.

5.3.5 Prefetching BTB Metadata

While hierarchical BTBs provide a trade-off between accuracy and delay, they still incur high latencies to access lower levels of the hierarchy. To hide the latency of accesses to lower-level predictor tables, several studies have proposed prefetching the predictor metadata from the lower-level predictor table into the first-level predictor table. To do so, PhantomBTB exploits the temporal correlation between BTB misses as misses to a group of entries are likely to recur together in the future due to the repetitive control flow in applications [12]. Emma et al. also propose spilling groups of temporally correlated BTB entries to the lower levels of the cache hierarchy and tagging each group with the instruction block address of the first instruction in the group [23]. This way, upon a miss in the instruction cache, the corresponding BTB entry group can be loaded from the secondary table into the primary table. As explained and quantified in Section 4.4.5, temporal correlation between individual BTB entries is highly limited due to frequent divergences, resulting in limited predictability of BTB misses.

In a similar vein, bulk preload [11] fetches a group of BTB entries that belong to a spatial code region of a predefined size upon a miss in that region. Although bulk preloading of BTB entries exploits the spatial correlation between BTB entries in a large code region, it falls short of

capturing the temporal correlation between BTB entries in different regions.

5.3.6 Synchronizing Predictor and Cache Content

One of the key ideas employed in Confluence, syncing the BTB content with the instruction cache, is similar to fetching the data prefetcher metadata of memory pages from off-chip to on-chip upon a TLB miss to a particular page as done in recent work [37] .

6 Concluding Remarks

6.1 Thesis Summary

The steadily growing demand for server applications call for more computing power, while the end of Dennard scaling has ceased continuous performance improvements provided by every new technology node. As a result, the growing need for more computation resources has pushed the server efficiency into the forefront of the server design requirements. Researchers and industry have started addressing the inefficiencies in commodity server processors by identifying the mismatch between the server application needs and the existing general-purpose server processor architectures. As a first step toward improving server efficiency, processor vendors have started integrating tens of low-power cores on a server chip, which are well-matched to the low ILP and high request-level parallelism prevalent in server applications, as opposed to a few aggressive cores tuned for applications with high ILP.

In this thesis, we identified the frontend inefficiencies associated with manycore server processors running server applications with large instruction working sets and proposed techniques to mitigate them. We first quantified the capacity requirements of the storage-intensive instruction-supply mechanisms, which maintain application metadata to predict future control flow, and demonstrated the commonality of metadata across cores running a given homogeneous server application. We concluded that the metadata commonality across cores

leads to unnecessary silicon provisioning due to similar metadata maintained redundantly for each core and paves the way for a specialized processor with a shared frontend to eliminate this redundancy.

As a first step toward a specialized server processor with a shared frontend, we proposed SHIFT, an instruction prefetcher with shared metadata generated by only one core selected at random and accessed by all other cores running the same homogeneous server application. The shared metadata embedded in the last-level cache in the SHIFT design allows for maintaining a disparate history instance for each application running on a CMP when multiple applications are consolidated. As a result, SHIFT eliminates the per-core history overhead, while amortizing the overhead of the shared history across many cores. We showed that despite its low storage overhead, SHIFT provides performance benefits similar to private history maintained per core even in the presence of workload consolidation.

We then identified the redundancy in metadata maintained by individual instruction-supply mechanisms, namely the instruction prefetcher and the branch target buffer. We observed that the metadata maintained for instruction prefetching is at instruction block granularity encapsulating the metadata maintained by the BTB and BTB prefetching metadata at instruction granularity. Based on this observation, we proposed Confluence, which maintains a lightweight BTB whose content mirrors the content of the instruction cache and is populated by SHIFT, as SHIFT prefetches instruction blocks into the cache. This way, Confluence eliminates the need for a disparate BTB prefetcher and the associated metadata, and limits the per-core BTB storage overhead only to the L1-resident primary instruction working set of an application. By doing so, Confluence delivers performance benefits close to that of a perfect BTB.

6.2 Future Directions

This thesis introduces two new mechanisms to enable sharing the metadata associated with two storage-intensive instruction-supply mechanisms, namely the instruction prefetcher and

branch target buffer. Because the shared instruction prefetcher provides a minimal-overhead way of predicting the active instruction working set in near future, which corresponds to the set of blocks resident in the instruction cache, it opens new opportunities to share other frontend metadata. Doing so would require organizing various metadata tables hierarchically (a small dedicated table per core and a large shared table), associating certain metadata with the SHIFT history entries based on the PCs, and prefetching metadata from the secondary storage into the primary storage with SHIFT's assistance, just like instruction blocks are prefetched from LLC into L1-I.

In the rest of this section, we summarize the opportunities and challenges associated with integrating various predictors into the shared frontend framework.

Branch Direction Predictor

As we quantified in Section 2.2.2, branch direction accuracy increases with the dedicated storage size, but the metadata storage is amenable to sharing due to the significant overlap of branch patterns across server cores.

Branch predictor tables are typically indexed by PCs or by hashing a branch PC, global branch history and/or path history. Predictors that use PC for indexing (e.g., bimodal, perceptron predictor [41]) are the primary candidates for sharing, as each entry can be easily associated with SHIFT history entries based on PCs. The branch predictor entries that belong a certain instruction block can be grouped together in the secondary storage, perhaps leveraging the virtualization framework, and then can be prefetched together into the dedicated primary storage along with the corresponding instruction block, as SHIFT predicts the instruction blocks that are likely to be accessed in near future.

The state-of-the-art branch predictors, however, typically rely on hash functions to index into the predictor tables [39, 68] and they also tag predictor entries to minimize aliasing [68]. Because the predictor entries do not maintain information about to the branch instructions or the encapsulating instruction blocks they belong to, integrating such predictors into the

shared frontend framework necessitates a disparate index table. The index table should be accessed with the address of the instruction block provided by SHIFT as prediction. Each entry in the index table should contain a list of pointers to the branch predictor table entries associated with a certain block, so that those entries can be prefetched all together into the dedicated primary storage from the shared storage. Such an index table would require extra storage, which would be amortized across many cores.

Nevertheless, the main challenge associated with prefetching metadata entries is the highly varying number of predictor entries per branch instruction, thus per instruction block. For example, in TAGE, while most branch instructions are easy to predict and are correctly predicted by the bimodal predictor without requiring any tagged entries, hard-to-predict branches maintain many tagged entries (i.e., hundreds of entries) due to the numerous global history patterns associated with them. This leads to two potential problems. First, the per-block pointer lists in the disparate index table should be over-provisioned for hard-to-predict branches, which would result in wasting space for the blocks with only a few entries. Second, fetching all those entries from the secondary storage into the primary storage and writing those entries into the secondary storage would consume significant on-chip network bandwidth offsetting the benefits of the increase in branch prediction accuracy.

Although several challenges associated with sharing branch direction metadata exist, enabling such a mechanism might enable the use of more sophisticated and storage-intensive branch predictors as their storage overheads would be amortized by many cores.

Translation Lookaside Buffer

Translation lookaside buffer (TLB) is used to cache virtual-to-physical address translations and has strictly low access-latency requirements as it is typically on the critical path when accessing the primary caches. Reducing the TLB size might be beneficial to reduce its access latency and power consumption. SHIFT's history can be used for I-TLB prefetching by predicting the pages that are likely to be accessed in near future, provided it can achieve sufficient lookahead

as performing a page-table walk might take hundreds of cycles. While SHIFT intrinsically incorporates predictions for future page accesses in the I-TLB, leveraging the SHIFT history for I-TLB prefetching automatically enables sharing I-TLB prefetching metadata.

Data Prefetchers

The state-of-the-art data prefetchers are also characterized by their storage-intensiveness as their metadata storage requirements scale with instruction and data working sets of applications [73, 74]. For example, the state-of-the-art data prefetcher that exploits spatial correlation, spatial memory streaming predictor (SMS) [74, 75], maintains history patterns per PC and requires up to 64KB of per-core storage to maximize its miss coverage for L1-D.

To mitigate the storage overhead and enable sharing of the predictor storage without losing its performance benefits, SMS entries whose PC tags fall into the same instruction block can be grouped together in the secondary storage and then can be prefetched into the dedicated primary storage through SHIFT's assistance. This way, only the SMS metadata of the instruction blocks that are likely to be accessed in near future would be maintained in the primary storage, significantly reducing the per-core storage overhead. We believe that similar techniques can be applied to data prefetchers indexed by PCs or instruction block addresses to enable sharing and reduction in dedicated storage capacity.

Bibliography

[1] Tor Aamodt, Paul Chow, Per Hammarlund, Hong Wang, and John Shen. Hardware support for prescient instruction prefetch. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2004.

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *The VLDB Journal*, pages 266–277, September 1999.

[3] D. W. Anderson, F. J. Sparacio, and Robert M. Tomasulo. The IBM system/360 model 91: Machine philosophy and instruction handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.

[4] Murali Annavaram, Trung. Diep, and John Shen. Branch behavior of a commercial oltp workload on Intel IA32 processors. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, 2002.

[5] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Call graph prefetching for database applications. *ACM Transactions on Computer Systems*, 21(4), December 2003.

[6] Applied Micro. https://www.apm.com, 2015.

[7] Islam Atta, Pinar Tozun, Anastasia Ailamaki, and Andreas Moshovos. Slicc: Self-assembly of instruction cache collectives for oltp workloads. In *Proceedings of the International Symposium on Microarchitecture*, 2012.

[8] Islam Atta, Pinar Tozun, Xin Tong, Anastasia Ailamaki, and Andreas Moshovos. Strex: Boosting instruction cache reuse in OLTP workloads through stratified transaction execution. In *Proceedings of the International Symposium on Computer Architecture*, 2013.

[9] Max Baron. The F1: TI's 65nm Cortex-A8. *Microprocessor Report*, 20(7):1–9, July 2006.

[10] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core cooperative TLB for chip multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[11] James Bonanno, Adam Collura, Daniel Lipetz, Ulrich Mayer, Brian Prasky, and Anthony Saporito. Two level bulk preload branch prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2013.

[12] Ioana Burcea and Andreas Moshovos. Phantom-BTB: A virtualized branch target buffer design. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[13] Ioana Burcea, Stephen Somogyi, Andreas Moshovos, and Babak Falsafi. Predictor virtualization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[14] Michael Butler, Leslie Barnes, Debjit D. Sarma, and Bob Gelinas. Bulldozer: An approach to multithreaded compute performance. *Micro, IEEE*, 31(2):6–15, March 2011.

[15] Brad Calder and Dirk Grunwald. Next cache line and set prediction. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

[16] Q. Cao, P. Trancoso, J.-L. Larriba-Pey, J. Torrellas, R. Knighten, and Y. Won. Detailed characterization of a Quad Pentium Pro server running TPC-D. In *International Conference on Computer Design*, 1999.

[17] Cavium ThunderX ARM Processors. www.cavium.com, 2015.

[18] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. Efetch: Optimizing instruction fetch for event-driven web applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation*, 2014.

[19] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[20] I-Cheng K. Chen, Chih-Chieh Lee, and Trevor N. Mudge. Instruction prefetching using branch prediction information. In *Proceedings of the International Conference on Computer Design*, October 1997.

[21] John D. Davis, James Laudon, and Kunle Olukotun. Maximizing CMP throughput with mediocre cores. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[22] Romain Dolbeau and Andre Seznec. Cash: Revisiting hardware sharing in single-chip parallel processor. *Journal of Instruction-Level Parallelism (JILP)*, 6, 2004.

[23] Philip G. Emma, Allan M. Hartstein, Brian R. Prasky, Thomas R. Puzak, Moinnuddin K.A. Qureshi, and Vijayalakshmi Srinivasan. Context look ahead storage structures, February 26 2008. IBM, US Patent 7,337,271.

[24] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the International Symposium on Computer Architecture*, 2011.

[25] Barry Fagin and Kathryn Russell. Partial resolution in branch target buffers. In *Proceedings of the Symposium on Microarchitecture*, 1995.

[26] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware.

In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[27] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive instruction fetch. In *Proceedings of the International Symposium on Microarchitecture*, 2011.

[28] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal instruction fetch streaming. In *Proceedings of the International Symposium on Microarchitecture*, 2008.

[29] Boris Grot, Damien Hardy, Pejman Lotfi-Kamran, Babak Falsafi, Chrysostomos Nicopoulos, and Yiannakis Sazeides. Optimizing data-center tco with scale-out processors. *Micro, IEEE*, 32(5):52–63, Sept 2012.

[30] Tom R. Halfhill. Ezchip's TILE-MX grows 100 ARMs. *Microprocessor Report*, 29(3):6–8, March 2015.

[31] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Proceedings of the International Symposium on Computer Architecture*, 2009.

[32] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *Micro, IEEE*, 31(4), July 2011.

[33] Stavros Harizopoulos and Anastassia Ailamaki. Steps towards cache-resident transaction processing. In *Proceedings of the International Conference on Very Large Databases*, 2004.

[34] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1997.

[35] R. B. Hilgendorf, G. J. Heim, and W. Rosenstiel. Evaluation of branch-prediction methods on traces from commercial applications. *IBM J. Res. Dev.*, 43(4):579–593, July 1999.

[36] Quinn Jacobson, Eric Rotenberg, and James E. Smith. Path-based next trace prediction. In *Proceedings of the International Symposium on Microarchitecture*, 1997.

[37] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the International Symposium on Microarchitecture*, 2013.

[38] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C. Steely, and Joel Emer. High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches. In *Proceedings of International Symposium on High Performance Computer Architecture*, 2015.

[39] Daniel A. Jimenez. Piecewise linear branch prediction. In *Proceedings of the International Symposium on Computer Architecture*, 2005.

[40] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the International Symposium on Microarchitecture*, 2000.

[41] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2001.

[42] John Kalamatianos and David R. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 1998.

[43] Svilen Kanev, Juan P. Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the International Symposium on Computer Architecture*, 2015.

[44] Cansu Kaynak, Boris Grot, and Babak Falsafi. Shift: Shared history instruction fetch for lean-core server processors. In *Proceedings of the International Symposium on Microarchitecture*, 2013.

[45] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP workloads. In *Proceedings of the International Symposium on Computer Architecture*, 1998.

[46] Ryotaro Kobayashi, Yuji Yamada, Hideki Ando, and Toshio Shimada. A cost-effective branch target buffer with a two-level table organization. In *Proceedings of the International Symposium of Low-Power and High-Speed Chips*, 1999.

[47] Aasheesh Kolli, Ali Saidi, and Thomas F. Wenisch. Rdip: Return-address-stack directed instruction prefetching. In *Proceedings of the International Symposium on Microarchitecture*, 2013.

[48] Kevin Krewell and Linley Gwennap. Silvermont energizes Atom. *Microprocessor Report*, 27(5):12–17, May 2013.

[49] Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen. Conjoined-core chip multiprocessing. In *Proceedings of the International Symposium on Microarchitecture*, 2004.

[50] Johny K.F. Lee and Alan J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, Jan 1984.

[51] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the International Symposium on Computer Architecture*, 1998.

[52] Pejman Lotfi-Kamran, Boris Grot, and Babak Falsafi. Noc-out: Microarchitecting a scale-out processor. In *Proceedings of the International Symposium on Microarchitecture*, 2012.

[53] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. Scale-out processors. In *Proceedings of the International Symposium on Computer Architecture*, 2012.

[54] Chi-Keung Luk and Todd C. Mowry. Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the International Symposium on Microarchitecture*, 1998.

[55] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the International Symposium on Microarchitecture*, 2007.

[56] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: an effective alternative to large instruction windows. *IEEE Micro*, 23(6):20–25, Nov.-Dec. 2003.

[57] Kyle J. Nesbit and James E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2004.

[58] Chris H. Perleberg and Alan J. Smith. Branch target buffer design and optimization. *Computers, IEEE Transactions on*, 42(4):396–412, Apr 1993.

[59] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1990.

[60] Jim Pierce and Trevor Mudge. Wrong-path instruction prefetching. In *Proceedings of the International Symposium on Microarchitecture*, 1996.

[61] Alex Ramirez, Luiz Andre Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. Code layout optimizations for transaction processing workloads. In *Proceedings of the International Symposium on Computer Architecture*, 2001.

[62] Alex Ramírez, Josep-L. Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero. Software trace cache. In *Proceedings of the International Conference on Supercomputing*, 1999.

[63] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[64] Glenn Reinman, Todd Austin, and Brad Calder. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the International Symposium on Computer Architecture*, 1999.

[65] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *Proceedings of the International Symposium on Microarchitecture*, 1999.

[66] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *Proceedings of the International Symposium on Computer Architecture*, 2011.

[67] Oliverio J. Santana, Alex Ramirez, and Mateo Valero. Enlarging instruction streams. *IEEE Transactions on Computers*, 56(10):1342–1357, 2007.

[68] Andre Seznec. The L-TAGE branch predictor. *Journal of Instruction Level Parallelism*, May 2007.

[69] Andre Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the alpha EV8 conditional branch predictor. In *Proceedings of the International Symposium on Computer Architecture*, 2002.

[70] Manish Shah, Robert Golla, Greg Grohoski, Paul Jordan, Jama Barreh, Jeff Brooks, Mark Greenberg, Gideon Levinsky, Mark Luttrell, Christopher Olson, Zeid Samoail, Matt Smittle, and Tom Ziaja. Sparc t4: A dynamically threaded server-on-a-chip. *Micro, IEEE*, 32(2):8–19, March 2012.

[71] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter,

and P. Williams. Ibm power7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1:1–1:29, May 2011.

[72] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.

[73] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *Proceedings of the International Symposium on Computer Architecture*, 2009.

[74] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *Proceedings of the International Symposium on Computer Architecture*, 2006.

[75] Stephen Somogyi, Thomas F. Wenisch, Michael Ferdman, and Babak Falsafi. Spatial memory streaming. *Journal of Instruction-Level Parallelism (JILP)*, 13, 2011.

[76] Lawrence Spracklen, Yuan Chou, and Santosh G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2005.

[77] Viji Srinivasan, Edward S. Davidson, Gary S. Tyson, Mark J. Charney, and Thomas R. Puzak. Branch history guided instruction prefetching. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2001.

[78] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[79] Edward H. Sussenguth. Instruction sequence control, January 26 1971. US Patent 3,559,183.

[80] Josep Torrellas, Chun Xia, and Russell Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 1995.

[81] Pinar Tözün, Islam Atta, Anastasia Ailamaki, and Andreas Moshovos. Addict: Advanced instruction chasing for transactions. *Proc. VLDB Endow.*, 7(14), October 2014.

[82] Pinar Tozun, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. From A to E: Analyzing TPC's OLTP benchmarks: The obsolete, the ubiquitous, the unexplored. In *Proceedings of the International Conference on Extending Database Technology*, 2013.

[83] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1996.

[84] Jim Turley. Cortex-A15 Eagle flies the coop. *Microprocessor Report*, 24(11):1–11, November 2010.

[85] Richard Uhlig, David Nagle, Trevor Mudge, Stuart Sechrest, and Joel Emer. Instruction fetching: coping with code bloat. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

[86] Alexander V. Veidenbaum. Instruction cache prefetching using multilevel branch prediction. In *Proceedings of the International Symposium on High-Performance Computing*, 1997.

[87] Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. In *Proceedings of the International Symposium on Computer Architecture*, 2005.

[88] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, July-Aug. 2006.

[89] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the International Symposium on Computer Architecture*, 2003.

[90] Chun Xia and Josep Torrellas. Instruction prefetching of systems codes with layout optimized for reduced cache misses. In *Proceedings of the International Symposium on Computer Architecture*, 1996.

[91] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *Proceedings of the International Symposium on Microarchitecture*, 1992.

[92] Chengqiang Zhang and Sally A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *Proceedings of the International Conference on Supercomputing*, 2000.

[93] Craig B. Zilles and Gurindar S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the International Symposium on Computer Architecture*, 2001.

Cansu Kaynak
cansu.kaynak@epfl.ch
parsa.epfl.ch/~kaynak

École Polytechnique Fédérale de Lausanne
Computer & Communication Sciences
INJ 238, Station 14
CH-1015 Lausanne, Switzerland
+41 21 69 31 379

Research Interests

Computer Architecture: Server system design, high-performance memory systems, instruction-fetch unit, core frontend, prefetching.

Education

Ph.D. in Computer Science, 2009-2015
École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
Thesis Title: Shared Frontend for Manycore Server Processors

B.Sc. in Computer Engineering, 2005-2009
TOBB University of Economics and Technology, Ankara, Turkey
GPA: 3.98/4.00 (Valedictorian)

Honors

- Google Anita Borg Scholarship, 2014-2015
- IBM Ph.D. Fellowship, 2014-2015
- IEEE Micro Top Picks from Computer Architecture Conferences of 2013 for "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware"
- Best Paper Award at the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) for "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware"
- Fellowship (full tuition & stipend), École Polytechnique Fédérale de Lausanne (EPFL), 2009-2010
- Valedictorian, TOBB University of Economics and Technology Class of 2009
- Award for Superior Achievement in Cooperative Education, TOBB University of Economics and Technology, 2009
- Scholarship (full tuition & stipend), TOBB University of Economics and Technology, 2005-2009

Publications (peer-reviewed)

- Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache
  D. Jevdjic, G. Loh, C. Kaynak, B. Falsafi, In Proc. of the 47th Annual IEEE Symposium on Microarchitecture (MICRO), 2014.
- A Case for Specialized Processors for Scale-Out Workloads
  M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, B. Falsafi, In IEEE Micro Top Picks, 2014.
- SHIFT: Shared History Instruction Fetch for Lean-Core Server Processors
  C. Kaynak, B. Grot, B. Falsafi, In Proc. of the 46th Annual IEEE Symposium on Microarchitecture (MICRO), 2013.
- From A to E: Analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored.
  P. Tozun, I. Pandis, C.Kaynak, D. Jevdjic, A. Ailamaki, In Proc. of the 16th International Conference on Extending Database Technology (EDBT), 2013.
- Quantifying the Mismatch Between Emerging Scale-out Applications and Modern Processors
  M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, B. Falsafi, In ACM Transactions on Computer Systems (TOCS), Vol.30, 2012.

- Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware
  M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, B. Falsafi, In Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.
- Proactive Instruction Fetch
  M. Ferdman, C. Kaynak, B. Falsafi, In Proc. of the 44th Annual IEEE Symposium on Microarchitecture (MICRO), 2011.
- Reducing the Energy Dissipation of the Issue Queue by Exploiting Narrow Immediate Operands
  I. C. Kaynak, Y.O. Kocberber, O. Ergin, In Journal of Circuits, Systems and Computers (JCSC), Vol. 19, 2010.

Experience

- École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
  Research Assistant at the Parallel Systems Architecture Lab, 2010-present
- Siemens Enterprise Communications, Ankara, Turkey
  Software Engineering Intern, April - August 2009
- Siemens Program and System Engineering, Ankara, Turkey
  Software Engineering Intern, January - April 2007

Teaching

- Teaching Assistant, Graduate Course, Topics in Approximate Computing Systems, Spring 2015.
- Head Teaching Assistant, Graduate Course, Topics in Datacenter Design, Spring 2014.
- Teaching Assistant, Undergraduate Course, Computer-Aided Engineering, Fall 2012.
- Head Teaching Assistant, Graduate Course, Advanced Multiprocessor Architecture, Fall 2010 & 2011.

Professional Activities

Co-developer

- FLEXUS, an open-source, scalable, full-system, cycle-accurate multi-processor and multi-core simulation framework.
- Shore-MT, an open-source, scalable open-source storage manager (implemented the TPC-E transaction processing benchmark).

Tutorial Organizer & Presenter

- CloudSuite 2.0 on Flexus Tutorial at 40th International Symposium on Computer Architecture (ISCA), 2013.
- CloudSuite on Flexus Tutorial at 39th International Symposium on Computer Architecture (ISCA), 2012.

External Reviewer

- HPCA'15, ICCD'14, TACO'14, HPCA'14, CAL'13, MICRO'12, IISWC'12, ICCD'11, ICS'11

Student Member

- IEEE, ACM SIGARCH