UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA
Dpto de Informática de Sistemas y Computadores

Design of Efficient TLB-based Data Classification Mechanisms in Chip Multiprocessors

A dissertation submitted in fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Science)

Author:
ALBERT ESTEVE GARCIA
Advisors:
Prof. ANTONIO ROBLES MARTÍNEZ
Prof. MARIA ENGRACIA GÓMEZ REQUENA
Prof. ALBERTO ROS BARDISA

Valencia, July 2017

Acknowledgment

Cuatro años de esfuerzo dan para mucho. Echándo la vista atrás, muchas son las personas que me han apoyado, muchos los que me han escuchado. Gracias a todos.

A Ramón y Lola. Sin ellos, sin su apoyo y su amor, no habría llegado hasta aquí. Sin su inspiración no habría siquiera empezado el camino. Ellos plantaron la semilla de la curiosidad, tan necesaria para superar la frustración y las dificultades a las que todo investigador tiene que enfrentar en su camino.

A Víctor. Hermano y amigo. El espejo en el que me miro. Me has dado fuerza en cada etapa de mi vida. Gracias por todo. Habrá que pensar una buena ruta en bici para celebrarlo.

A Carla. Por su paciencia, por su ayuda, por respaldarme cuando lo he necesitado. Siempre me has animado a seguir adelante y has permanecido a mi lado en cada etapa del doctorado. Más que eso, eres uno de los pilares principales de mi vida.

A Salomé, Eloy y Sergio. Por quererme y aceptarme como uno más de la familia. Realmente yo lo siento del mismo modo. Gracias.

A mis amigos. Samuel, Eduardo, Manolo. Ya sea jugando a algún juego, organizándo viajes juntos, o saliendo de fiesta o a tomar algo, siempre habéis estado ahí, siendo partícipes de mis logros y mis frustraciones. Tanto como yo de las vuestras. Y espero que siga siendo así por muchos, muchos años.

Allá por 2007 empezaba mi andadura por la universidad, hace ya diez años. Ésta ha sido mi segunda casa, y le debo mucho. Y tengo muchos que agradecer también a toda la gente que he conocido en este etapa. Incluyendo ésta última fase en el Grupo de Arquitecturas Paralelas.

Por supuesto y ante todo a mis directores, Alberto, Antonio y Maria Engracia. Me habéis guiado e inspirado, desafiado y ayudado en cada etapa de mi doctorado. Vuestra paciencia y esfuerzo han sido claves para la consecución de este trabajo. He aprendido mucho durante estos años. No sólo en materia de arquitectura de computadores, sino también en investigación científica; a amarla, a observar los pequeños detalles, a analizar los datos y entender qué esta ocurriendo para así poder seguir avanzando. Ellos han cambiado mi vida, profesional y personal, para siempre.

A Núria, Salva y Eduardo. No nos vemos tanto como antes, nuestras vidas, sus obligaciones, nos alejan. Pero aún nos quedarán esas tardes de cervezas para ponernos al día. Siempre.

A mis compañeros de laboratorio. José Vicente, José María, Migue, Vicent, Joan, Fran, Javi, Roberto, Carlos, Santi, y un largo etcétera. A los que se fueron, Knut, Mario, Crispín. Tantos y tantos. A todos os debo algo. Lo que ha unido el BoardGameArena que no lo separen nuestras exitosas carreras en el extranjero, como un tren de mercancías desbocado y sin frenos. Y por supuesto a Ricardo. El verdadero pilar del laboratorio, en lo técnico y en lo personal. Tu increible paciencia y dedicación te hacen pieza central del trabajo de todos nosotros en el grupo.

A Stefanos Kaxiras y a toda la gente que conocí en Uppsala. Trevor, David, Alexandra, Andra, etc. As tough as it was, it has been one of the most enriching experiences of my life. I cannot thank you enough. Observing first hand how another research group works really changed my

perspective towards investigation. Specially Kaxiras, your wisdom was inspirational. Thank you all. Already missing the snow.

A.E.

Contents

Preface

The current document has been elaborated with the aim to obtain a PhD degree in Computer Science. This work has been done under the advising and guidance of professors Antonio Robles, Alberto Ros, and Maria Engracia Gómez.

This document can be divided in three parts. The first part introduced the state-of-art in classification approaches and private-based data optimizations.

The second part present all the research work and its evaluation. This research activity has been developed within the Parallel Architectures Group (GAP) in the Department of Computer Engineering (DISCA) at the Universitat Politècnica de València.

In the last part, the contributions and conclusions are summarized.

List of Acronyms

TLB Translation Lookaside Buffer.

CS-TLB Complete Subblock TLB.

PS-TLB Partial Subblock TLB.

CMP Chip Multiprocessor.

NUCA Non-Uniform Cache Architecture.

LLC Last-Level Cache.

SC Sequential Consistency.

ROB Reorder Buffers.

TSO Total Store Ordering.

RAWR Read-After-Write Races.

NSRT Not Shared Region Table.

CRH Cached Region Hash.

TC Temporal Coherence.

DRF Data Race Free.

HRF Heterogeneous Race Free.

nDRF Non Data Race Free.

xDRF Extended Data Race Free.

ROB Reorder Buffers.

hLRC Heterogeneous Lazy Release Consistency.

PR Private Read-only.

PW Private Written.

SR Shared Read-only.

SW Shared Written.

OS Operating System.

VIPS Valid/Invalid Private/Shared (set of cache coherence states).

MOESI Modified Owned Exclusive Shared Invalid (set of cache coherence states).

MESI Modified Exclusive Shared Invalid (set of cache coherence states).

SWEL Shared Written Exclusivity-Level (set of cache coherence states).

UNITD Unified Instruction/Translation/Data Coherence.

DiDi Dictionary Directory.

FAC First Accessing Core.

TI Thread Identifying.

VM Virtual Memory.

MMU Memory Management Unit.

PTE Page Table Entry.

PPN Physical Page Number.

IPI Inter-Processor Interrupts.

PI/VI Physically/Virtually Indexed.

PT/VT Physically/Virtually Tagged.

ASID Address Space Identifier.

NoC Network On Chip.

SLL Shared Last-Level.

MSHR Miss Status Holding Register.

GEMS General Execution-driven Multiprocessor Simulator.

SLICC Specification Language for Implementing Cache Coherence.

CACTI Cache Access and Cycle Time Information.

IPC Instructions Per Cycle.

PID Process Identifier.

List of Figures

List of Tables

Abstract

Most of the data referenced by sequential and parallel applications running in current chip multiprocessors are referenced by a single thread, i.e., private. Recent proposals leverage this observation to improve many aspects of chip multiprocessors, such as reducing coherence overhead or the access latency to distributed caches. The effectiveness of those proposals depends to a large extent on the amount of detected private data. However, the mechanisms proposed so far either do not consider either thread migration or the private use of data within different application phases, or do entail high overhead. As a result, a considerable amount of private data is not detected. In order to increase the detection of private data, this thesis proposes a TLB-based mechanism that is able to account for both thread migration and private application phases with low overhead. Classification status in the proposed TLB-based classification mechanisms is determined by the presence of the page translation stored in other core's TLBs. The classification schemes are analyzed in multilevel TLB hierarchies, for systems with both private and distributed shared last-level TLBs.

This thesis introduces a page classification approach based on inspecting other core's TLBs upon every TLB miss. In particular, the proposed classification approach is based on exchange and count of tokens. Token counting on TLBs is a natural and efficient way for classifying memory pages. It does not require the use of complex and undesirable persistent requests or arbitration, since when two or more TLBs race for accessing a page, tokens are appropriately distributed classifying the page as shared.

However, TLB-based ability to classify private pages is strongly dependent on TLB size, as it relies on the presence of a page translation in the system TLBs. To overcome that, different TLB usage predictors (UP) have been proposed, which allow a page classification unaffected by TLB size. Specifically, this thesis introduces a predictor that obtains system-wide page usage information by either employing a shared last-level TLB structure (SUP) or cooperative TLBs working together (CUP).

Resum

La major part de les dades referenciades per aplicacions paral·leles i seqüencials que s'executen en CMPs actuals són referenciades per un sol fil, és a dir, són privades. Recentment, algunes propostes aprofiten aquesta observació per a millorar molts aspectes dels CMPs, com és reduir el sobrecost de la coherència o la latència d'accés a memòries cau distribuïdes. L'efectivitat d'aquestes propostes depen en gran mesura de la quantitat de dades detectades com a privades. No obstant això, els mecanismes proposats fins a la data no consideren la migració de fils d'execució ni les fases d'una aplicació. Per tant, una quantitat considerable de dades privades no es detecta apropiadament. A fi d'augmentar la detecció de dades privades, aquesta tesi proposa un mecanisme basat en les TLBs, capaç de reclassificar les dades com a privades, i que detecta la migració dels fils d'execució sense afegir complexitat al sistema. Els mecanismes de classificació en les TLBs s'han analitzat en estructures de diversos nivells, incloent-hi sistemes amb TLBs d'últim nivell compartides i distribuïdes.

Aquesta tesi presenta un mecanisme de classificació de pàgines basat en inspeccionar les TLBs d'altres nuclis després de cada fallada de TLB. Concretament, el mecanisme proposat es basa en l'intercanvi i el compte de tokens. Comptar tokens en les TLBs suposa una forma natural i eficient per a la classificació de pàgines de memòria. A més, evita l'ús de sol·licituds persistents o arbitratge, ja que si dues o més TLBs competeixen per a accedir a una pàgina, els tokens es distribueixen apropiadament i la classifiquen com a compartida.

No obstant això, l'habilitat dels mecanismes basats en TLB per a classificar pàgines privades depenen de la grandària de les TLBs. La classificació basada en les TLBs resta en la presència d'una traducció en les TLBs del sistema. Per a evitar-ho, s'han proposat diversos predictors d'ús en les TLBs (UP), els quals permeten una classificació independent de la grandària de les TLBs. Específicament, aquesta tesi introdueix un predictor que obté informació d'ús de la pàgina a escala de sistema mitjançant un nivell de TLB compartida (SUP) or mitjançant TLBs cooperant juntes (CUP).

Resumen

La mayor parte de los datos referenciados por aplicaciones paralelas y secuenciales que se ejecutan en CMPs actuales son referenciadas por un único hilo, es decir, son privados. Recientemente, algunas propuestas aprovechan esta observación para mejorar muchos aspectos de los CMPs, como por ejemplo reducir el sobrecoste de la coherencia o la latencia de los accesos a cachés distribuidas. La efectividad de estas propuestas depende en gran medida de la cantidad de datos que son considerados privados. Sin embargo, los mecanismos propuestos hasta la fecha no consideran la migración de hilos de ejecución ni las fases de una aplicación. Por tanto, una cantidad considerable de datos privados no se detecta apropiadamente. Con el fin de aumentar la detección de datos privados, proponemos un mecanismo basado en las TLBs, capaz de reclasificar los datos a privado, y que detecta la migración de los hilos de ejecución sin añadir complejidad al sistema. Los mecanismos de clasificación en las TLBs se han analizado en estructuras de varios niveles, incluyendo TLBs privadas y con un último nivel de TLB compartido y distribuido.

Esta tesis también presenta un mecanismo de clasificación de páginas basado en la inspección de las TLBs de otros núcleos tras cada fallo de TLB. De forma particular, el mecanismo propuesto se basa en el intercambio y el cuenteo de tokens (testigos). Contar tokens en las TLBs supone una forma natural y eficiente para la clasificación de páginas de memoria. Además, evita el uso de solicitudes persistentes o arbitraje alguno, ya que si dos o más TLBs compiten para acceder a una página, los tokens se distribuyen apropiadamente y la clasifican como compartida.

Sin embargo, la habilidad de los mecanismos basados en TLB para clasificar páginas privadas depende del tamaño de las TLBs. La clasificación basada en las TLBs se basa en la presencia de una traducción en las TLBs del sistema. Para evitarlo, se han propuesto diversos predictores de uso en las TLBs (UP), los cuales permiten una clasificación independiente del tamaño de las TLBs. En concreto, esta tesis presenta un sistema mediante el que se obtiene información de uso de página a nivel de sistema con la ayuda de un nivel de TLB compartida (SUP) o mediante TLBs cooperando juntas (CUP).

Chapter 1

Introduction

This chapter briefly describes the context in which this dissertation is set and the reasons that have motivated it (Section 1.1). Then, we define the objectives aimed by this dissertation (Section 1.2). Following, we show the contributions that such objectives have originated (Section 1.3). Finally, we outline the structure of the remaining chapters in this dissertation (Section 1.4).

1.1 Context and Motivation

Gordon E. Moore stated in 1965 that the number of transistors in a dense integrated circuit doubles every two years, since transistors get smaller every successive technology. This prediction, referred to as Moore's Law, proved accurate for many decades, where computers have rapidly evolved.

Currently, most high-performance processors are compound by billions of transistors, commonly organized by integrating multi- and many-core systems into a single chip (i.e., chip multiprocessors or CMPs). Core count on these CMPs is rapidly growing to cope with Moore's Law. Most architectures already offer quad-core processors, reaching up to dozens of processors. For instance, Intel has recently presented a 72-core CMP [34], or the Kalray's MPPA2 with up to a 256-core processor [38]. These large-scale CMPs are organized as tiled architectures, designed as arrays of identical or nearly-identical blocks, namely *tiles*. CMP tiles are compound of one or several cores, one or several levels of caches, and a network interface (router) connecting all tiles in a point-to-point interconnect. The cores of a tiled CMP usually share one or more memory modules, constituted into shared-memory multiprocessors. In this context, a cache coherence protocol is required so the memory is accessed consistently from different processors. Hence, as long as the number of cores rapidly grows, implementing low-latency and scalable coherence protocols in shared-memory CMPs leads to new challenges for future CMP architectures.

On the one hand, *snooping-based* protocols send broadcast requests to all other system' processors through a bus, or bus-like interconnect. Thus, even though snooping-based protocols provide low-latency cache-to-cache misses, the bandwidth requirements for protocols grow ex-

ponentially with the core count. Consequently, snooping protocols are only suitable on small core-count CMPs. Nonetheless, there are many approaches which try to reduce the traffic generated [19, 9, 43, 44], but scaling to larger systems is still troublesome.

On the other hand, *directory-based* protocols send requests to the home directory memory, which track the sharers of each memory block. Then, the directory responds with data or forwards the request to the appropriate processor(s). Therefore, directory-based approaches require less network bandwidth, representing a more scalable solution. However, consulting the directory implies an indirection that is placed in the critical path. Furthermore, the directory structure employed increases on-chip area and leakage power as the core count grows. Many proposals try to address the directory scalability problem [89, 27, 74, 22, 88, 11, 26] in order to scale to large-scale CMPs.

Another important design aspect of CMPs is the organization of the last-level cache (LLC). The wire delay latency of accessing each particular cache bank of a multi-banked shared memory prevails over the access latency to the bank itself. This design choice is referred to as *NUCA* (Non-Uniform Cache Architecture) structure. NUCA caches are designed to reduce the number of off-chip accesses [41]. However, NUCA access latency is strongly dependent on the cache bank where a particular block is mapped. Consequently, the average access latency to NUCA caches increases with the number of cores, jeopardizing their scalability. Some works have addressed this inefficiency [21, 49, 31, 32].

Among the approaches tackling the scalability problem for future CMPs, some increasingly appealing solutions try to discern the private (i.e., accessed by only one thread) or shared (i.e., accessed by two or more threads) nature of accessed data, since they provide a wide range of data optimizations to improve coherence protocols and the organization of the LLC. For instance, Cuesta *et al.* propose Coherence Deactivation, which identifies private [22] and read-only [23] (non-coherent) blocks, and avoids storing those blocks in the directory cache, since they do not require coherence maintenance. Therefore, directories exploit their limited storage capacity more efficiently and their access latency is reduced. Alternatively, Hardavellas *et al.* [31] and Li *et al.* [46, 48] keep private blocks in the NUCA bank of the requesting processor to reduce the access latency to NUCA caches. Ros and Kaxiras [71] propose an efficient and simple cache coherence protocol by implementing a write-back policy for private blocks and a write-through policy for shared blocks. Finally, End-to-End SC [79] allows instruction reordering and out-of-order commits of private accesses from the write-buffers, since they do not affect the consistency model enforced by the system.

The observation behind these proposals is that most referenced data both for sequential and parallel workloads are private. Therefore, data optimizations are commonly applied to private blocks (i.e., private-based optimizations). As a consequence, the effectiveness of most classification-based data optimizations relies on the amount of private data detected. However, although being a decisive factor, some previous proposed classification mechanisms do not provide runtime classification (e.g., classification status is obtained at compile time), which limits the precision of the mechanism. Moreover, some classification approaches do not detect shared-to-private transitions (i.e., non-adaptive classification) [22, 31, 43]. Non-adaptive approaches may miss-classify most accessed data as shared at some point in applications running for a long time, ultimately neglecting the advantages of the classification. Discern store memory operations allows determine read-only data, which represents a 82 percent (on average) of the memory blocks accessed [23]. Therefore, detecting read-only data significantly improves the potential of a classification mechanism. Finally, some classification approaches store the data sharing status in the cache or the directory structures [11, 25, 33, 62, 88], limiting the applicability of the classification, since the private-shared information is obtained after the cache miss.

In order to cope with these drawbacks, this thesis introduces a novel family of page classification mechanisms relying on the presence of the page translation in the system translation lookaside buffers (TLBs), namely *TLB-based* classification. TLB-based classification exploits proximity of cores in a CMP, scrutinizing other system' TLBs through fast core-to-core communication upon every TLB miss, retrieving the page sharing information alongside the page translation entry. This way, TLB-based classification mechanisms accelerate the page table walk process while performing a page-level data classification.

1.2 Objectives

This section presents the objectives of this dissertation. Classification approaches improve system performance and scalability mainly through private-based data optimizations. Thus, the higher the amount of private data detected, the more the benefits obtained from the optimization applied. The main goal is to avoid all classification overheads, while pursuing all desirable properties for classification approaches. In order to achieve this goal, some specific objectives need to be accomplished:

- Design new classification alternatives in order to efficiently perform a runtime classification based on the presence of translation entries in the system' TLBs, while accelerating page translation process through TLB-to-TLB transfers.
- Detect shared-to-private transitions in order to improve private detection according to the current page access pattern, resulting in what we call a temporality-aware classification scheme.
- Thoroughly analyze page lifetimes on TLBs in order to determine an ideal classification, settled by the concurrence of accesses to a page.
- Design a prediction mechanism in order to bring TLB-based classification accuracy closer to the ideal.
- Test the benefits of our classification mechanisms through different data optimization techniques, providing evidences that affirm TLB-based classification as the best suited approach for data classification.

1.3 Thesis Contributions

The aforementioned objectives have originated several contributions, which are briefly summarized in this section.

1.3.1 TLB-based Classification Mechanisms

The TLB-based classification mechanisms proposed in this thesis are a new family of protocols designed to dynamically discern the sharing status of the data based on the presence of page translations in the system TLBs. Therefore, the classification is at page granularity. TLBs are employed to store the classification information. Different TLB-based classification approaches have been proposed:

Snooping TLB-based classification

This thesis rests on a *Snooping TLB-based classification*, or SnoopingTLB, as a starting point. The goal of SnoopingTLB is to achieve an adaptive classification that accounts for temporarily-private pages and tolerates thread migration. The TLB-based classification mechanism is based on inquiring the other cores' TLBs in the system (through TLB-to-TLB requests) on every TLB miss. TLBs reply to the requester indicating whether or not they are caching the page translation, which is included in the response if they do, thus accelerating the table page walking process. This way, the TLB suffering the miss naturally discovers whether blocks belonging to a page may be currently stored on a remote cache and therefore the page is shared, or, on the contrary, no TLB is currently storing the translation and the page is private. TLB-to-TLB transfers are based upon the observation that core-to-core communication in CMPs is much faster compared to traditional processors. Other works also benefit from this observation with different aims [63, 81]. Snooping TLB-based classification targets purely-private, single or multilevel TLB structures.

TLB-based classification with a shared last-level TLB

We exploit the use of a distributed shared last-level TLB structure (DirectoryTLB) similar to the NUCA cache organization [42], performing a page-level classification while avoiding most TLB traffic overheads of the snooping TLB-based classification approach. Such a distributed TLB organization has been previously suggested [47], however it has not been extensively explored. The shared TLB structure is accessed after every TLB miss and eviction, offering up-to-date sharing information. Note that this scheme has some similarities to a cache directory in the context of cache coherence, and thus its name.

Token-counting TLB-based classification

Token-counting classification, namely *TokenTLB*, is a novel classification mechanism implemented directly in the TLB structure and inspired by Token coherence [51, 53, 55] protocols. TokenTLB is based on the observation that, unlike Token coherence, applying tokens for classification does not require issuing persistent requests nor complex arbitration mechanisms. Persistent requests are a special type of request meant to solve races occasionally caused when several cores want to write data at the same time. These requests are a source of complexity for the coherence protocol, possibly being one of the main causes why Token coherence has not been implemented in commodity systems. However, TokenTLB avoids these races by only aiming at classifying data. When two or more TLBs race for accessing the same page, tokens are naturally distributed among the TLBs, and the page will be consequently classified as shared.

TokenTLB reduces network consumption compared to snooping TLB-based proposals. Only TLBs holding extra tokens provide them along with the page translation, which leads to about one response per TLB miss. Furthermore, token-based classification makes it possible to naturally and immediately identify a shared TLB page entry transitioning to private, improving private detection over any other TLB-based classification approach in this dissertation.

1.3.2 Prediction-based Classification Mechanisms

TLB-based classification is determined by the presence of page entries in TLBs, despite the fact that they may have ceased to be accessed. This makes classification accuracy sensitive to the TLB size. Ideally, the sharing condition of a page should be settled by the concurrence of accesses to that page. Therefore, decoupling classification from TLB size in TLB-based classification mechanisms requires a mechanism able to predict whether a page is going to be accessed or not in the near future. To this aim, different prediction approaches have been proposed.

Usage predictor for TLBs (UP)

This prediction strategy represents another starting point in this thesis, resembling the one employed in the Cache Decay approach [40]. UP determines whether or not a page is going to be accessed in the near future. In particular, the predictor uses one saturated counter per TLB entry that is periodically increased according to an internal period (i.e., predictor period) and reset on every memory access to the page. When a TLB entry is predicted not to be used (its counter is saturated) and it is probed with a TLB-to-TLB request, the entry is invalidated. When a TLB entry is invalidated, the core is disqualified as a potential sharer of the page. This way, the usage predictor effectively increases the amount of private data detected for large or multilevel TLBs.

Forced-sharing usage predictor

Employing shorter periods for usage predictions may prematurely invalidate more translations, which in turn causes the TLB miss rate to rocket, producing more network inquiries, and ultimately, invalidating even more TLB entries. It can be seen as a positive feedback situation, or ping-pong invalidations when only two nodes are involved.

In order to avoid ping-pong invalidations, which could make the TLB usage predictor unattractive, a slight modification of the prediction strategy is introduced, namely *Forced Sharing*. The forced-sharing strategy avoids prediction-induced invalidations after the first prematurely invalidated entry is detected. A translation entry is considered prematurely invalidated when the page is reaccessed soon after being invalidated (i.e., the invalidated entry is still in the TLB). In order to avoid further miss-predicted page invalidations in other cores' TLBs, a special *forced-sharing* TLB miss request is sent. This request overrides the prediction, and grants a classification based solely on the presence of page translation entries in other TLBs.

Shared usage predictor (SUP)

As part of our proposed DirectoryTLB classification scheme for shared last-level TLBs, we attune UP to the new environment, namely Shared Usage Predictor (SUP). The key observation in this work is that invalidations caused by the previous predictors do not necessarily lead to higher detection of private pages. In fact, many TLB entry invalidations just reduce the number of sharers, indiscriminately increasing TLB miss rate, without the certainty of improving the amount of private data. Hence, UP is applied blindly, without considering how the classification status will evolve, despite being supposedly at its service.

SUP relies on the shared second-level TLB in order to track the sharers count, and announces a page sharer falling into disuse to the home TLB tile as soon as is predicted as so. This way, translation invalidations are only performed when the shared second-level TLB discerns a reclassification opportunity, improving prediction accuracy, acting as a natural filter to blind TLB invalidations, and avoiding most prediction overheads.

Cooperative usage predictor (CUP)

SUP averts invalidating translations based only on their local usage prediction, based on the information stored in the classification directory (i.e., a shared last-level NUCA-like TLB level). Similarly, in order to improve the prediction accuracy for purely-private TLB structures, *Cooperative Usage Predictor* (CUP) has been proposed. CUP exploits TLB cooperation (i.e., TLBs willingly working together for a common purpose or benefit) in order to: (i) perform a system-wide page usage prediction instead of a per-core prediction, naturally discovering reclassification opportunities without relying either on other TLBs requesting the page or the presence of a shared last-level TLB; (ii) neglect most aimlessly invalidated TLB entries, as invalidation is postponed until it may expressly serve a better data classification, avoiding performance degradation.

1.4 Thesis Outline

This dissertation starts with the introductory chapter (Chapter 1). Following, Chapter 2 describes the background of data classification, presenting the main classification alternatives constituting the current state of the art. The simulation environment is described in Chapter 3. Chapter 4 presents and evaluates snooping TLB-based classification approaches for different single and multilevel TLB structures. Then, Chapter 5 introduces the token-counting TLB-based classification approach. Usage prediction is first introduced in Chapter 6, and extended with our Cooperative Usage Predictor in Chapter 7. This dissertation ends with Chapter 8, summarizing the conclusions and displaying the main contributions to the research field.

Chapter 2

Concepts and Background

This chapter presents the basics and terminology for private-shared data classification and private-based optimizations. For the sake of brevity we cover the main concepts, giving insight on the classification approaches assumed in this thesis.

2.1 Introduction

First, this chapter describes some private-based data optimizations to show the interest of performing a private-shared classification in Section 2.2. Next, we discuss the state-of-the-art classification alternatives (Section 2.3) and how they fit into the desirable features for classification mechanisms (Section 2.3.1). Finally, some virtual address space concepts are introduced, along with some techniques to improve TLB performance, in Section 2.4.

2.2 Classification-based Optimizations

Data classification mechanisms are gaining interest as they allow many optimizations regarding hardware management of CMP components such as caches, interconnect or coherence protocols, based on their sharing status. For instance, while all cache protocols assume that all the accessed data may be shared at any time, data will not require coherence maintenance while being private. There are many other recent examples in the literature showing the large variety of optimization for a classification scheme that discerns the private (or read-only) nature of data. This section describes some of key classification-based data optimizations.

2.2.1 Interconnect Optimizations

In bus snooping protocols there is no tracking information about block sharing, and thus every cache coherence request needs to be broadcast to all nodes. Therefore, snooping-based protocols are commonly faster than directory solutions for cache coherence, since they avoid indirections. However, their main disadvantage is their limited scalability. On the one hand, frequent snooping causes message racing, which increases cache access time and consumption. On the other hand, network bandwidth for broadcasts grows rapidly with the number of cores.

RegionScout

In snooping-based protocols broadcast on private memory is unnecessary, as it will result in a global miss. Moshovos [57] proposed RegionScout, a family of filters for *snooping* coherence protocols that avoid requesting, receiving or forwarding non-coherent data through broadcast messages when accessing private blocks, ultimately constraining network bandwidth, latency and consumption.

RegionScout filters comprise two structures local to each node: (i) a *not shared region table* (NSRT), a very small, cache-like structure storing regions (contiguous memory space that is power of two size) that are known to be not shared; and (ii) a cached region hash (CRH), a filter that records superset of all regions locally cached.

Whenever a node issues a memory request for a block for the first time, other nodes respond as corresponds to the coherence protocol, but including information regarding any other cached blocks for the same region as the block that has been requested. The node that caused the miss records the region in its NSRT, which resolves the block status as not shared. Following misses within the same region by the same node will result in private accesses, avoiding the coherence broadcast. Then, if a different node requests a block within the same region (which would miss in its own NSRT), the existing NSRT entry must be invalidated in order to ensure correctness.

The CRH structure represents an up-to-date imprecise record of regions that are locally cached. Basically, CRH is a counter for each hashed region, that is increased after every L2 cache miss within a given region and decreased after every L2 cache eviction. This way, after receiving a coherence request, a given node consults its CRH to know if the access results in a region miss or if it may have blocks stored.

Subspace Snooping

Kim *et al.* [43] propose *Subspace Snooping*, which extends *RegionScout* by tracking the sharers of a page, and snoop requests are sent only to those nodes in the subspace through a multi-cast message. Subspace Snooping maintains the page status by combining updates of page table entries and TLB entries. Therefore, it allows fast accessing the sharing information without requiring extra hardware support. Furthermore, sharing information is extended with a bitmap (i.e., sharing vector), tracking the sharers of each page. After every TLB miss the fine-grain sharing information is updated both in the page table and the TLBs.

If pages cease being accessed from a given core (e.g., all its stored blocks are evicted), the information kept in the sharing vector may be outdated, as it still accounts for the obsolete sharers. This information is considered as subspace pollution, and need to be dynamically addressed. To do so, a TLB may remove itself from the sharing vector after evicting from

the local cache the last cacheline of a given page. The corresponding TLB entry must also be invalidated. To check whether a page is cached or not, a counting bloom filter-based technique [18] is employed.

2.2.2 Cache Optimizations

This section presents some cache optimizations which aim is to improve cache access latency and avoid address translation latency by discerning the sharing nature of the accessed data.

Reactive NUCA

While a NUCA cache is a common design for last-level caches (LLC), increasingly on-chip wire latency might as much as halve the potential performance on server workloads. Hardavellas *et al.* [31] and Kim *et al.* [46, 48] propose a mechanism where private blocks are kept on the local (or close to local –neighboring) NUCA bank, and replicates distant shared blocks in order to reduce access latency and balance capacity constraints. Consequently, block mapping in R-NUCA is guided by its classification:

- Logically divides the LLC into overlapping clusters of neighboring slices, replicating instructions at cluster granularity. The cluster size allows trading off access latency to instruction blocks for cache capacity. Small clusters provide low latency access at the cost of higher data replication and thus, lower capacity per cluster. Conversely, large clusters result in higher access latency, but less replication degree. Instead of standard address interleaving, instruction clustering technique for R-NUCA indexes blocks into a size-4 fixed-center cluster employing a rotational interleaving, where each is core is assigned with a rotational ID (RID).

- Maps private data into the local LLC slice of the requesting core. Logically, it can be seen as a size-1 cluster (i.e., loosely or tightly connected computers) mapping.

- Shared read-write data is placed at fixed address-interleaved locations in a size-N cluster (where N is the size of the system).

PS-TLB

Y. Li *et al.* [47] avoid pipeline stalls when obtaining the virtual to physical address translation by introducing a small buffer structure close to the TLB, namely partial sharing buffer (PSB). Private translations are stored locally for low-latency access. Then, when a page becomes shared the translation is distributed as in NUCA memories among all cores' PSBs. Therefore, a shared translation is obtained with lower latency and lesser storage resources than a second-level TLB scheme.

The PS-TLB architecture works as follows. After a request for a virtual address request, the L1 and L2 TLBs are sequentially checked in order to retrieve the physical address, which is obtained after hitting on either TLB level. Conversely, if a miss occurs in the L2 TLB, it may be present either in the PSB home tile or the page table (or both), and thus both are searched in parallel. If the PSB home hits on a valid entry, it replies to the requester with the corresponding translation. In that case, the translation from the page table is simply discarded when received on the requesting core. If the request misses on the PSB home, the translation is resolved on the page table and the translation is stored in the local private TLB hierarchy if

the page is private, or in both the TLB structure and the PSB home if the PSB is shared. This way, cores without the specific translation are not stalled upon TLB shootdowns, avoiding some of the shootdown process overhead. Furthermore, PSBs are not flushed on shootdowns, and their entries may be employed when reaccessing a page after a context switch or a migrating thread.

2.2.3 Coherence Protocol Optimizations

Core count rapid growth in current CMPs, alongside the implementation of a shared-memory programming model, leads to the requirement of efficient coherence maintenance among data in private caches. These circumstances bring new challenges to make coherence protocols scalable and provide increasing performance. This section reviews some proposals based on the characterization of the accessed data.

Coherence Deactivation

Directory-based coherence is one of the best suited protocols in terms of scalability and is present on most commodity processors nowadays. The directory is in charge of tracking data sharers in order to invalidate outdated copies on writes, and obtain the most recent copy on reads. This way, sequential consistency is granted for directory-based protocols. However, on larger CMPs, the directory cache may suffer from scalability problems. Directory area and latency overheads increase in order to avoid evictions, as the eviction of a directory entry usually entails the invalidation of blocks on the lower memory hierarchy levels. Due to the limited size or associativity of directory caches or the lack of a backup directory, a system may produce frequent invalidations of directory entries, which dramatically increases the number of *Coverage* misses [67] (i.e., cache misses caused by invalidations on the directory cache due to its limited capacity) and, therefore, results in performance degradation.

However, the coherence problem arises only for shared resources (present in multiple local caches). Thus, if a block is known to be present only in a single private cache (i.e., private), coherence maintenance may be bypassed. In this regard, *Coherence Deactivation* identifies private [22] and read-only [23] (non-coherent) blocks, and avoids storing those blocks in the directory cache. Therefore, directories exploit their limited storage capacity more efficiently as long as classification mechanism becomes more accurate, while the availability of the directory is improved for blocks that really need coherence (i.e., shared blocks).

Since coherence deactivation overrides the coherence protocol for non-coherent accesses, a *recovery* operation is required when a page becomes coherent again, in order to avoid inconsistencies. Recovery is a costly system-wide operation in charge of atomically updating TLBs' sharing status and return to a coherent state. To this end, the blocks of a non-coherent page that becomes coherent must be either evicted from the cache (flushing-based recovery) or updated in the directory cache (update-based recovery). Once the recovery mechanism has finished, the directory cache is in a coherent state according to the new page classification.

VIPS

Ros and Kaxiras [71] propose an efficient and simple directory-less and broadcast-less cache coherence protocol which requires only three stable states, namely *VIPS* (i.e., Valid/Invalid Private/Shared). Although sequential consistency is not supported with VIPS for all applications, its definition is still satisfied for Data-Race-Free (DRF) applications.

VIPS implements a dynamic write policy: a simple write-through policy (i.e., store operations traverse the underlying storage before confirming completion to host) for shared blocks and an efficient write-back policy (i.e., store operations are directed to cache and completion is immediately confirmed) for private blocks, which represents the great majority of write misses on commercial workloads. Then, on a second step, VIPS-M waits for a synchronization point with acquire semantics (e.g., lock, barrier, wait) on DRF applications to be exposed to the hardware in order to selectively flush (i.e., self-invalidate) shared data (employing a private-shared classification scheme). This way, the requirement for tracking block readers is avoided. When a node writes into a shared data, the sharers will self-invalidate on the next synchronization point. Then, with the addition of self-downgrade (i.e., write-back) upon a release synchronization (e.g., unlock, barrier, signal), the need for a directory cache or snooping broadcast messages is completely averted.

This coherence scheme simplifies the coherence protocol, resulting in less static and dynamic energy consumption compared to conventional MESI directory protocols. Nonetheless, any synchronization that relies on races (e.g., spin-waiting races, which are employed for signaling, locking, and barrier primitives) comprises the effectiveness of the protocol due to the lack of explicit invalidations, as it repeatedly self-invalidate and re-fetch in order to obtain the spin flag value updated by writes. To solve this, a new, simple, and transparent *callback* mechanism has been proposed [72], which are set by reads involved in a spin waiting and satisfied by writes. A callback signal blocks any read to a variable that is waiting for a write, while it has not occurred since its last invocation. This way, the benefits of explicit invalidation are obtained, while the cost of an invalidation protocol is avoided (e.g., employing a directory to track all data).

Timestamps-based coherence

Dealing with the scalability issues of directory protocols, timestamp-based hardware coherence protocols present an appealing alternative.

TC-Strong. This is a time-based coherence protocol for GPUs, namely Temporal Coherence (TC) [80], based on globally synchronized counters. With TC-Strong, these synchronized counters are maintained in the GPU cores and L2 controllers, allowing to self-invalidate cache blocks and maintain coherence, thus eliminating coherence traffic, and reducing are overhead and protocol complexity. Small timestamps fields are added to the L1 and L2 cache entry format. The L1 timestamps indicates when a specific L1 entry is invalidated, while L2 timestamp indicates when all L1 cachelines have been self-invalidated (L2 entries are never evicted while their timestamp has not expired). TC-Strong implements an optimization to eliminate write-stalling for private data. An L2 line reads are permitted only while data are private. Writes to private L2 lines are only effective if they are from the core that originally performed the read. Then, store requests from the L1 include their local timestamps, which are matched against the global timestamp in the L2 to check when a core is performing a private write.

TC-Release++. TC-Release [87] is a port from TC to maintain coherence in general purpose CMPs. Private cache lines with TC-Release in write-back caches do not need to maintain

timestamps and self-invalidate upon expiration, leading to higher L1 cache hit rate. This observation is further extended to shared read-only lines, as they do not require coherence maintenance. Therefore, all private or read-only L1 cache lines are kept in the cache as long as possible.

2.2.4 Memory Consistency Models

The memory consistency model of a shared-memory multiprocessor provides a formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by the system.

End-To-End sequential consistency

Sequential consistency (SC) is arguably the most intuitive behavior for a shared-memory multithreaded program. A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and behaves as an interleaving of the memory accesses from its threads. Furthermore, language-level SC improves programmability on a CMP. Modern programming languages as C++ or Java provide SC only for DRF programs, while for racy programs only weak consistency models are provided.

End-to-End SC [79] allows instruction reordering and out-of-order commits from the reorder buffers (ROB) of private accesses from the store buffers, since they do not affect the consistency model enforced by the system. It also allows a load to a shared-read-only location to commit from the ROB without waiting for the store buffer to drain.

Racer: TSO Consistency

TSO (Total Store Ordering) is the memory model currently employed in one of the most common family of processors (TSO-x86). With TSO there is always a single, global order of writes to shared memory from all cores.

Racer [73] is a coherence approach based on self-invalidate and self-downgrade that provides TSO consistency. This is guaranteed by detecting read-after-write races (RAWR) at run time and treating them as synchronization. Since private data do not affect consistency, Racer causes self-invalidations of only shared data in the local cache of a racing reader. On the other hand, while private stores are free to coalesce in the local cache, Racer employs a store buffer to coalesce shared stores, while keeping the order seen by other cores. Therefore, when a racy read occurs, all stores previous (in program order) to the conflicting one are seen by the reader. Finally, private accesses are not tracked for race detection, thus incurring small hardware requirements and avoiding false positives at the RAWR bloom filters. Overall, Racer provides TSO consistency outperforming even state-of-the-art SC-for-DRF models.

2.3 Data Classification Mechanisms

All the approaches seen in the previous section require a classification mechanism to perform their data management optimization. This section first discusses some desirable features of classification schemes which are key for their effectiveness. Then, we describe the main state-of-the-art classification mechanisms and how they cope with those features.

2.3.1 Desirable Features of Classification

Firstly, the classification should be performed with low-overhead in terms of traffic, performance, and area. Although this is key for every classification approach, there are some other main properties for classification approaches which are decisive in order to make classification attractive:

- Knowing the classification before the memory access reaches the cache structure is critical to the applicability of the classification mechanism. This ***a-priori*** knowledge of data sharing status is a requirement for many data optimizations (e.g., *Coherence Deactivation* [22] or *Reactive NUCA* [31]).

- A block is commonly accessed concurrently from two or more cores (i.e., shared) only during a particular period, becoming private again after some time, since the cache lines are evicted after inactivity periods. A classification approach should be able to detect data transitioning back to private, or else most data would end up as shared on applications running for a long time, miss-classified due to *temporal* data patterns. This property, namely classification ***adaptivity***, permits invoking the same schedulable entity (e.g., application's thread) from different system components (i.e., thread migration) without incurring in miss-classified shared accesses.

- Besides private and shared characterization, detecting non-written regions of data (i.e., ***read-only*** classification) provides more insight to classification, which ultimately allows better performance for classification applications that support read-only data optimizations. Therefore, when a store memory operation occurs, the sharing status evolves into a *written* state. The classification private-shared dichotomy is extended to four states: Private Read-only (PR), Private read-Write (PW), Shared Read-only (SR), and Shared read-Write (SW).

- Runtime classification information is decisive, as privacy cannot be always guaranteed at compile time. Commonly, complex compiler analyses (e.g., data reuse analysis or data disambiguation, among others) are usually employed to guarantee the privacy of the accessed data. Conversely, runtime classification performs an ***accurate***, efficient data classification.

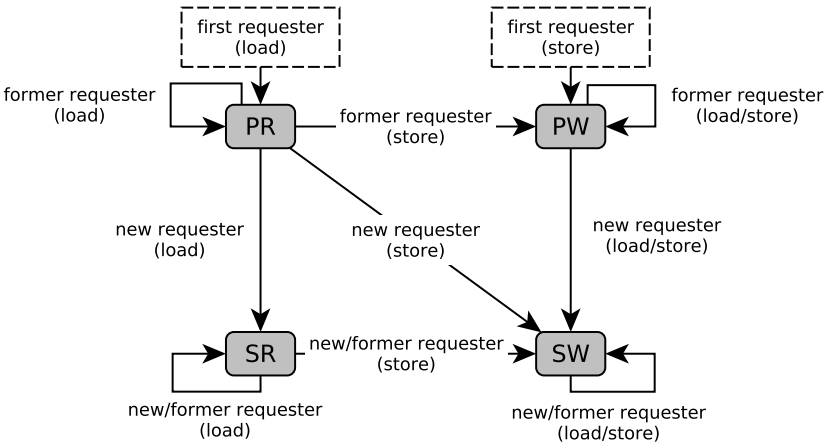


Figure 2.1: State transition diagram for OS-based classification status.

2.3.2 OS-based Classification

Some mechanisms classify at page-level aided by the operating system (OS) [22, 23, 31, 43, 47]. OS-based mechanisms do not require additional hardware support because they take advantage of existing OS structures (i.e., page table and TLBs). An OS-based classification considers a page as private on a page table fault. Then, the first time the virtual-to-physical address translation is requested after a TLB miss, the requesting core is annotated in the page table (keeper [22] or FAC -first accessing core- [47] field). On subsequent accesses to the page table entry, the keeper field is compared with the current requester. If the keeper field matches, it means that the keeper's TLB suffered an eviction and the page may continue as private. Conversely, if the keeper field does not match on a private page table entry, then the page is reclassified as shared. To this end, each page table entry adds a P bit that indicates the page state (private or shared). The P bit is also included in the TLB entries to allow a fast access to the page state for those cores that have the page entry in the TLB. When a page changes from private to shared, the core having the page as private must be notified in order to update the sharing status of its TLB accordingly. Moreover, OS-based classification has been also extended to support shared read-only data [23].

Figure 2.1 shows the state transition of the sharing status for a page in an OS-based classification approach. The left side of the diagram corresponds to the basic private and shared states, and the right side introduce the written states when read-only data is accounted for. Note how when a page falls either as shared or written, it remains as so for the rest of the execution time (until the page is evicted from memory), since OS-based classification is mostly non-adaptive.

2.3.3 Directory-based Classification

Directory-based mechanisms [62, 33, 11, 88, 24] store the sharing status in either the directory or the last level cache structure (LLC-based mechanism), by adding some additional states in the corresponding memory. Figure 2.2 represents the state digram for SWEL [62], a cache coherence protocol that discerns five combinations of the states Shared, Writen, and Exclusivity Level. SWEL employs a LLC-based classification mechanism to discern the sharing status. Private data is stored in the L1 cache, while shared data reside at the shared L2 cache. Sharing status is determined at block level after the cache structure is accessed. The first time that a block transitions to Shared, the L1 cache copies of the block are invalidated through a broadcast message. Once the blocks classification transitions to either Shared or Written it remains as so

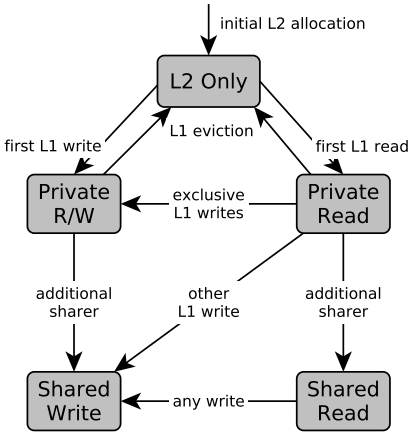


Figure 2.2: State transition diagram for the SWEL protocol.

until its eviction (i.e., until the block generation time ends). Then, the classification becomes private again if reaccessed.

These proposals allow performing a runtime, software-transparent classification at page or cache-line granularity. Cache-line granularity avoids spatial aliasing problems, on what data may be considered shared on broader granularity schemes, whereas blocks are never concurrently accessed. However, directory-based approaches might add prohibitive storage requirements or deal with dual-granularity complexities.

Directory-based classification mechanisms commonly require explicit notification to the directory upon L1 cache misses and replacements in order to accurately determine the generation of every cache line and classify accordingly [24]. In this scheme, shared directory lines in the directory may be evicted without invalidating cache lines (i.e., self-contained directory). This allows smaller, yet efficient directories. However, this approach requires DRF semantics and self-invalidate cache lines on the next synchronization point. Moreover, an additional logarithmic sharer count field is required per directory entry in order to revert classification from shared to private. Even though the storage requirement increase may be excessive for directory-based approaches to support classification adaptivity, it is required in order to provide sequential consistency (SC).

Finally, LLC-based classification can employ a Cache Decay [40] or any similar technique in order to shorten cache line generation time according to its access pattern, which ultimately favors a more accurate private data detection.

2.3.4 Compiler-assisted Classification

Compiler-assisted approaches [46, 48] exploit representative patterns existing in a huge variety of data-parallel applications to perform a classification into private or shared. In these benchmarks, each thread derives its own set of local variables with thread-dependent values to specify which regions of each array to access. These local variables are called Thread Identifying (TI) variables [46]. The compiler needs to identify these TI variables to determine how each thread accesses different portions of memory. One of the commonly used methods for specifying TI variables is to pass different values to parallel threads as function arguments. Another common way to specify TI variables in multi-threaded applications is through *mutex* lock directives to protect global variables. This type of code is much more difficult for a compiler to analyze.

Table 2.1: Properties of classification schemes

	A-priori	Read-Only	Adaptive	Accurate
Directory-based	✗	✓	✓	✓
OS-based	✓	✓	✗	✓
Compiler-assisted	✓	✓	✓	✗
TLB-based	✓	✓	✓	✓

Moreover, there are particular programming structures, such as loops and conditionals, that determine the memory access pattern of the threads (i.e., Thread Identifying Structures).

However, compiler-assisted data classification must remain conservative, as they deal with the difficulty of knowing at compile time (i) whether a variable is going to be accessed or not, and (ii) in which cores the data will be scheduled and rescheduled [35]. Furthermore, compiler assisted-approaches are not transparent to software, as they require recoding and/or recompilation for legacy software.

2.3.5 Classification Based on Programming Languages

Finally, some approaches based on the properties of programming languages [69, 70] can precisely identify *extended data-race-free* (xDRF) regions (i.e., a set of DRF regions acting as one unique region) at compile-time. DRF regions may be interleaved with non-DRF (*nDRF*) regions, but they do no break data-race-free semantics. OpenMP programming model is employed to discern sharing status, since it is controlled by dedicated synchronization constructs (e.g., atomic, critical), which are easily identified statically.

Then, finer granularity compile-time classification is complemented by an OS-based classification to increase the accuracy with runtime page-level information, alleviating the conservative actions of a static classification. The benefit of the combined classification approaches works both ways, since the static approach resets the classification to private at the boundaries of xDRF regions, allowing the OS-based classification to become adaptive These static-dynamic classification approaches, despite being very accurate, are not applicable to most existing codes, since the compiler must unequivocally identify xDRF regions.

2.3.6 Summary

Table 2.1 summarizes the main properties of the previously introduced state-of-the-art classification approaches. In the first place, Directory-based classification needs to access the LLC or the directory cache (i.e., traversing the L1 cache) in order to obtain the sharing status of the block. This mechanisms are not applicable to some private-based optimizations, since they need the classification to be known before missing in the L1 cache.

OS-based mechanism is based on accesses to the page table in order to hint for concurrent accesses from multiple cores, which transitions the page classification status to shared. However, when a page ceased to be used in a core, the page table is not updated. Consequently, OS-based classification is not temporality-aware, which implies that: i) shared-to-private transitions are not detected, a page must be evicted from main memory in order to be reclassified to private, and ii) a page is possibly miss-classified as shared since it might not be currently accessed from two or more cores (e.g., due to a migrating thread). Since data access patterns change throughout different phases of the application lifetime [39, 78], adaptivity is fundamental to achieve a good classification accuracy.

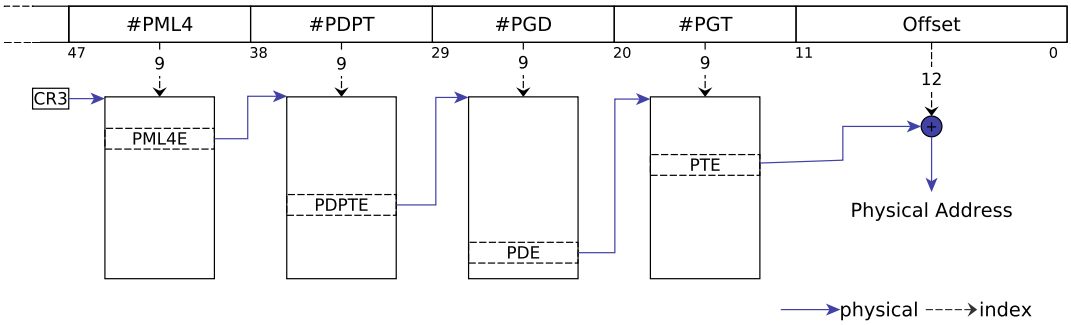


Figure 2.3: Example of a translation process (page table walk) in the x86 64.

Read-only classification (i.e., detecting non-written regions of data) is also a far-reaching property, as merely accounting for shared-read-only blocks they can reach in some cases up to 48.7% of all accessed blocks [23]. Most data classification mechanisms can detect read-only data at the cost of raising the hardware complexity, increasing the network consumption or adding storage requirements.

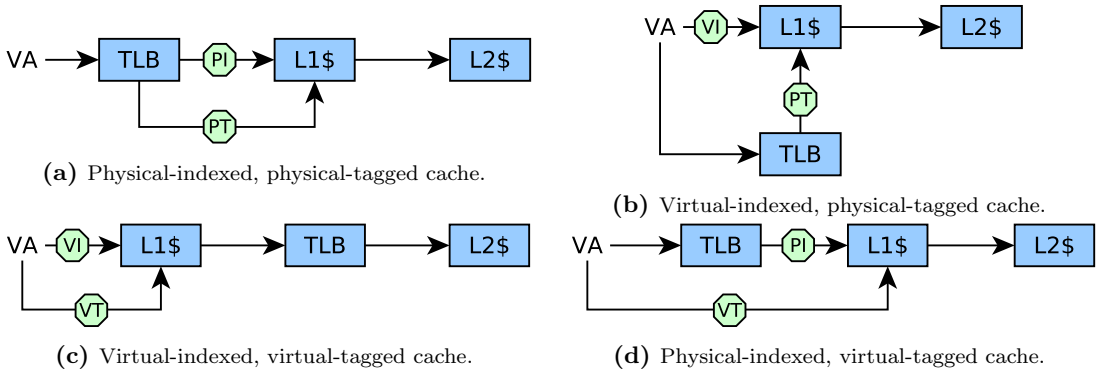
Finally, compiler-assisted approaches need to be conservative, because private condition for memory accesses in these approaches cannot be proved in most cases, which compromises the *accuracy* of the mechanism. Mechanisms based on the programming languages (i.e., static-dynamic approaches) rely on OpenMP directives to discern block-level, adaptive sharing information at compile time, and an OS-based classification mechanism to obtain runtime, page-level, sharing information. Nonetheless, these mechanisms are only suited for applications that identify xDRF regions, which limits its applicability.

This thesis introduces a new family of TLB-based protocols that aim to reach all desirable properties for a classification mechanism.

2.4 Architecting and Managing TLBs

This section first introduces terms and concepts in the context of TLB management. Then, we discuss some techniques and design choices for architecting TLBs and improve their performance.

Address translation is the process regulating the access to physical memory given a virtual memory (VM) address. Modern memory management units (MMUs) divide address space into pages, and therefore divide memory into a set of logical multi-level hierarchical structures called page tables. Each page table contains one page table entry (PTE) per page, mapping virtual page number (VPN) to physical page numbers (PPN), and additional bits representing meta-data associated to the mapping. Retrieving the page translation (*page table walk*) require multiple memory accesses, as page tables are composed at least by four hierarchical levels for common 64-bit systems (Figure 2.3). Moreover, the number of levels required for address translation dramatically grows with systems supporting virtualization (e.g., up to twenty-four memory accesses on x86-64 virtual address space [13], or fifteen memory accesses for the recent 32-bit ARMv7 virtual address space [2]). Consequently, multiple sequential memory accesses are required to retrieve the translation, which is added to the critical path. To avoid high-latency memory accesses, short latency instruction and data TLBs were introduced in order to store and fast access the result of a page table access (i.e., cache of mappings from the page table, a segment table, or both).



(a) Physical-indexed, physical-tagged cache.

(b) Virtual-indexed, physical-tagged cache.

(c) Virtual-indexed, virtual-tagged cache.

(d) Physical-indexed, virtual-tagged cache.

Figure 2.4: Diagrams for different cache addressing options.

2.4.1 TLB Overview

This section discusses concepts that offer some background to the challenges involved in the design and optimization of TLBs.

Coverage: coverage of TLB represents the sum of memory mapped by all its entries. Naturally, the higher the coverage the better, specially with applications with large working sets.

Superpage: a superpage refers to a virtual memory page that is power of two times bigger the system page size (e.g., 2MB, 4MB, etc.), which map to contiguous physical frames. Superpages improve TLB coverage, and can be stored in the same multi-grain TLB, or in a separate, single, full-associative TLB [58].

TLB shootdown: some events (e.g., page swaps, privileges adjustment, context switches, etc.) cause a change in the virtual-to-physical memory mapping. On such events, TLB entries caching outdated translations need to be invalidated by means of an operation referred to as TLB shootdown. Current CMPs rely on the OS to discover the set of TLBs holding the translation and perform a synchronization operation through costly Inter-Processor Interrupts (IPIs). The impact of TLB shootdowns increase linearly with the number of cores, which in turn leads to scalability bottlenecks [63].

Cache addressing options: on an L1 cache lookup, index and tag can be obtained either from virtual address or physical address, resulting in four different combinations (Figure 2.4) of physically/virtually indexed (PI or VI) and physically/virtually tagged (PT or VT):

- PIPT: cache uses the physical address for both the address and the tag. Although it is a simple design, it is slow, since both the index and the tag are obtained from the TLB, which is in the critical path (Figure 2.4a).
- VIPT: cache index is taken from the virtual address and tag is provided from the physical address by the TLB, after a TLB lookup. VIPT design diminishes the TLB access latency and allows consulting the cache line in parallel (Figure 2.4b). This is the most commonly used design choice.
- VIVT: index and tag are both obtained from the virtual address (also referred to as virtual caches). This scheme allows much faster lookups and energy savings, since the TLB only needs to be consulted after L1 cache misses. Therefore, the TLB is in the critical path for L1 cache misses. Furthermore, virtual to physical translation is required on L1 cache

evictions to perform a write-back to the physical L2 cache (Figure 2.4c). This option is not used because of the synonym problem.

- PIVT: cache index is taken from the physical address after a TLB lookup. Since index is required before tag, the TLB is in the critical path to access L1 (Figure 2.4d). This scheme is often claimed in the literature as useless and non-existing [36].

Synonyms and homonyms: the synonym problems arises when several different virtual addresses in the same process map to the same physical address, and thus several cache lines might end up storing data for the same physical address if the cache is virtually tagged. Writing to such locations leads to potential data inconsistency. Similarly, a virtually indexed cache might become inconsistent when the same virtual address in the same process maps to distinct physical addresses, which is referred to as homonyms.

2.4.2 Boosting TLB Performance

The number and size of TLBs is growing to effectively address the increasing application memory footprints and constrain the potential performance loss. Additionally, several solutions have been proposed to minimize TLB miss rate and latency, and thus improve TLB performance: tunning TLB size or associativity [20], multiple TLB levels, different page sizes (superpages) [82], or prefetching [75]. However, most of these proposal targeted uniprocessors. TLB design for CMPs need to be reevaluated.

TLB coherence

TLB shootdowns are originated by TLB coherence problems. The performance impact of TLB shootdowns in CMPs is addressed in some recent works which propose different hardware supported TLB coherence schemes:

UNified Instruction/Translation/Data (UNITD) Coherence. Romanescu *et al.* [63] propose a mechanism by which system TLBs participate in the cache coherence protocol, alongside data and instruction caches. The observation behind UNITD is that, while cache coherence is performed in hardware, TLB coherence is performed in software. Software solution for TLB coherence is an efficient, low-cost operation well-suited for small number of processors. However, for medium- to large-scale CMPs, moving TLB coherence to hardware reduce the performance impact of TLB shootdowns.

DiDi: A shared TLB directory. Villavieja *et al.* [84] propose DiDi, a directory-like solution for TLB coherence, which introduces a shared second level TLB acting as a dictionary directory (DiDi), tracking every address translation stored in first-level TLBs and their presence bitmap. DiDi ensure up-to-date information by intercepting all insertions and evictions on L1 TLBs. DiDi is designed to reduce the impact of TLB shootdowns in large-scale CMP systems by enabling lightweight TLB invalidation. On a TLB shootdown, the core send an invalidation to DiDi, and then is forwarded only to those cores storing the translation entry. Then, only those cores need of interrupting execution due to the TLB shootdown. Finally, all the translations in DiDi are tagged with a special *Address Space IDentifier* (ASID) field, since the shared TLB stores translations from different cores, potentially executing different address spaces. The ASID field in DiDi also helps to avoid problems with virtual address homonyms.

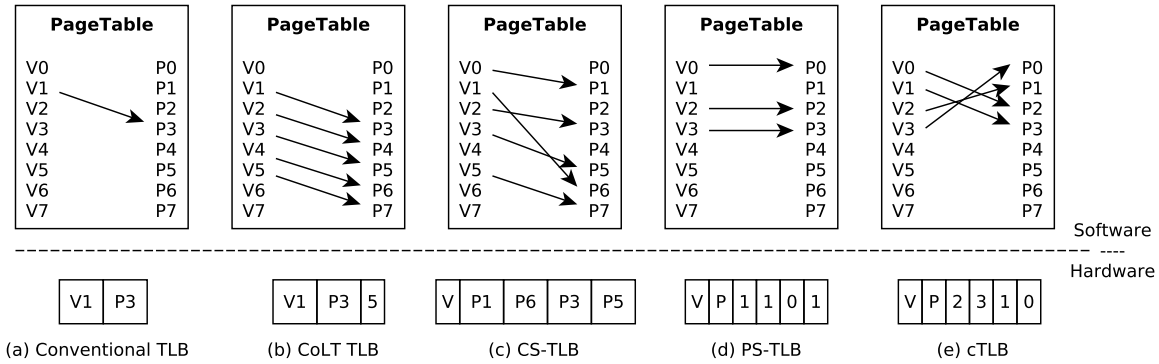


Figure 2.5: An illustration of (a) conventional TLB (b) CoLT (c) CS-TLB (d) PS-TLB and (e) cTLB. For each approach, the structure of a single entry and a page table with the PTEs that can be exploited is shown.

Exploiting spatial locality

There is a second regime of page allocation where the OS allocates contiguous physical page frames to contiguous virtual pages. Since superpages usually aggregate hundreds of pages (e.g., a 2MB superpage require 512 contiguous 4KB pages), these OS-assigned regions fall short. The following hardware techniques are orthogonal to superpaging, and exploit contiguous PTE spatial locality:

Sub-blocking. Talluri and Hill[82] propose two subblock TLB designs: complete-subblock TLB (CS-TLB), and partial sub-blocking (PS-TLB). CS-TLB (Figure 2.5c) associates a tag with an aggregated superpage-sized region (i.e., contiguous VPNs pointing to different non-contiguous PPNs), thus increasing TLB coverage. With PS-TLB (Figure 2.5d) all PTEs that have VPNs and PPNs with the same offset form the start of an aligned package are coalesced to a single entry, indicating the matches with a bitmap. PS-TLB entry is much smaller than CS-TLB entry, however it has stronger restrictions for constructing a superpage and requires minor OS changes.

Coalesced Large-reach TLBs (CoLT). Pham *et al.* [61] propose CoLT (Figure 2.5b) to exploit PTEs spatial locality to aggregate data and increase TLB hit ratio, but without the overheads of superpaging, such as complex algorithms and inflated I/O traffic. Therefore, while conventional TLB entries correspond to a single PTE, a CoLT entry maps to a region composed by dozens of contiguous, spatially-local PTEs. They assume a two-level TLB with set-associative TLBs for 4KB page size and an additional small, full-associative TLB for superpages, which is accessed in parallel with the L1 TLB. L2 TLB is inclusive with the L1 TLB, but not the superpage TLB. Three different variants of the technique are presented, merging many translation into the superpage TLB.

Clustered TLBs. Pham *et al.* [60] propose Clustered TLBs (Figure 2.5e), a multi-granular TLB architecture that exploits clustered spatial locality, in which a cluster of nearby virtual pages map to a clustered set of physical pages. Therefore, this clusters may be coalesced into the same TLB entry, providing robustness against memory fragmentation. The set of physical pages can be stored in one single clustered TLB (cTLB) entry. Conversely, entries without any spatial locality can be stored into a traditional TLB. Clustered TLBs does not require OS support and works even under fragmentation.

Inter-core sharing patterns

A TLB miss for a page translation with the same page size, virtual page, physical page, context ID and protection information as a translation previously accessed from a different core in a CMP is defined as an inter-core shared TLB miss. Parallel applications in CMPs exhibit inter-core TLB miss patterns among threads. Exploiting these patterns might reduce TLB misses, mitigate their impact, and optimize the capacity of the system' TLBs:

Synergistic TLBs. Srikantaiah *et al.* [81] observed that some applications show significant miss reduction when increasing the number of TLB entries (i.e., *borrower*), whereas others' miss rate reduction is negligible (i.e., *donor*). Moreover, duplicate translations in different TLBs waste TLB capacity. To overcome this, Synergistic TLBs was introduced in order to exploit inter-core sharing patterns by providing capacity sharing. Capacity sharing allows storing a victim translation from one TLB in another, which emulates a distributed-shared TLB. Synergistic TLBs also supports both translation migration (from *borrower* to *donor* TLBs), maximizing the TLB capacity and boosting both multiprogrammed and multi-threaded workloads; and translation replication, reducing the access latency of multi-threaded workloads for remote TLBs on large-scale CMPs. For both, Synergistic TLBs introduce *precise* and *heuristic* approaches where: i) *precise* maintains a large amount of access history, and ii) *heuristic* track local and remote hits.

Inter-Core Cooperative TLB prefetchers. Based on the observation that inter-core shared TLB miss are predictable, Bhattacharjee *et al.* [16] introduce Inter-Core Cooperative TLB prefetchers. These prefetchers assume a leader-follower role between TLBs, in order to push page translations into a follower's structure, namely Prefecth Buffer (PB). PB is placed in parallel with the TLBs. Accessing the PB avoids further inter-core shared TLB misses.

Shared Last-level TLB. Bhattacharjee *et al.* [15] presents a CMP with private, per-core L1 TLBs backed by a shared, centralized L2 TLB. Since further levels may be added to the TLB structure, this proposals is referred to as shared last-level (SLL) TLB. SLL TLB is accessed after L1 TLB misses and, since it is shared among cores, page translations in the SLL TLB are accessible by all system' TLBs, which avoids subsequent L1 TLB misses. This way, SLL TLB naturally exploits inter-core sharing patterns in parallel applications, at the cost of increasing the communication delay compared to a private L2 TLB scheme. The centralized design is not relied upon when scaling to a large number of cores, since centralized organizations increase end-to-end latencies as the core count grows. Furthermore, centralized TLBs require a high-bandwidth interconnect to be able to communicate with all the cores of a CMP.

To avoid the overhead of strict inclusion, SLL TLB is designed to be *mostly inclusive* of L1 TLBs. Therefore, on a miss, the translation is stored in both the L1 and the SLL TLB, whereas evictions occur independently for each entry. This approach provides near-perfect (e.g., 97%) inclusion. However, upon a TLB shootdown, both levels need to be checked.

Chapter 3

Simulation Environment

This chapter describes the simulation infrastructure and workloads used for all the evaluations carried out in this thesis. Experiments have entailed running several representative workloads on a simulation platform.

3.1 Introduction

We evaluate our proposal with full-system simulation using Virtutech Simics [50] along with the Wisconsin GEMS toolset [54], which enables detailed simulation of multiprocessor systems. CACTI [83] has been employed to obtain the latencies and dynamic consumption for the different memory system components and appropriately configure them on the simulator. Full-system simulation lets us to evaluate the proposed systems running realistic scientific applications on top of actual operating systems. The interconnection network has been modeled using the GARNET simulator [8]. ORION 2.0 [37] is employed to obtain power results for the simulated network, including links, interconnection routers, and other network on chip (NoC) components. The relationship between the different simulation tools employed is depicted in Figure 3.1.

Our proposal is evaluated through a wide variety of parallel workloads from several benchmarks suites, covering different sharing patterns and sharing degrees. Many of the applications employed to perform the evaluation are mainly from the SPLASH-2 benchmark suite [85], the PARSEC benchmark suite [17], and the ALPBenchs suite [45]. Moreover, some commercial workloads have also been used [10].

The rest of the chapter is structured as follows: Section 3.2 introduces the simulation tools employed to evaluate the proposals in this thesis. Next, Section 3.3 details the base system parameters modeled by our simulator. Section 3.4 discusses the metrics employed to evaluate our proposals. Finally, Section 3.5 describes the benchmarks and applications used for carrying out the evaluation, which run on top of the simulator.

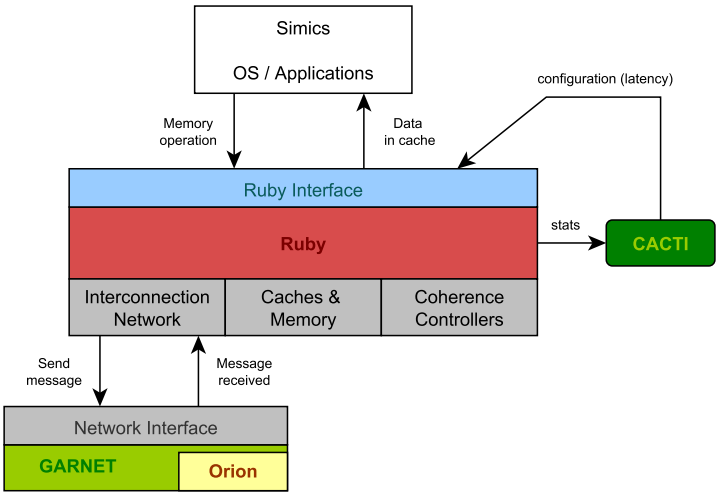


Figure 3.1: Relationship between the simulation tools employed.

3.2 Simulation Tools

This section describes the simulation tools employed in this thesis, which consisted mainly in full-system simulations on top of Simics-GEMS simulator. Area requirements, and cache and TLB latencies have been calculated with the CACTI tool.

3.2.1 Simics-GEMS

Simics [50] is a full-system multiprocessor simulator, trying to balance simulation accuracy and performance by modeling the complete final application. Simics allows us to evaluate our proposals through realistic workloads on top of real, unmodified operating systems. Full-system simulation captures the timing effects of the applications (e.g., the dynamic change of executed instructions based on different input data), whereas trace-driven simulations remain agnostic to dynamic behaviors.

GEMS (General Execution-driven Multiprocessor Simulator) [54] is a modular simulation infrastructure which leverages Simics full-system simulator to extend it with timing features, both for the memory system and microprocessors. Modular design provides flexibility through different available modules implemented in C++, although only Ruby has been used to test the proposals in this thesis.

Ruby is a queue-driven module that models the memory hierarchy, including caches, cache controllers, system interconnect, memory controllers and banks of main memory. Different coherence protocols may be modeled for Ruby, specifying its individual controller state machines behavior through SLICC (Specification Language for Implementing Cache Coherence), a domain specific language. Each controller consists in a per-memory-block with: states, events, transitions, and actions. SLICC compiler generates C++ code for different controllers, and optionally, HTML protocol specification. The timing is modeled by setting the latencies and modeling the wire delay on network communications. Interconnection network is modeled in detail at microarchitecture level with GARNET simulator [8].

Table 3.1: System parameters for the baseline system.

Memory Parameters (GEMS)	
Processor frequency	2.8GHz
TLB hierarchy	Exclusive
Split instr & data L1 TLBs	8 sets, 4-way (32 entries)
L1 TLB hit time	1 cycle
Unified L2 TLB	128 sets, 4-way (512 entries)
L2 TLB hit time	2 cycle
Page size	4KB (64 blocks)
Cache hierarchy	Non-inclusive
Cache block size	64 bytes
Split instr & data L1 caches	64KB, 4-way (256 sets)
L1 cache hit time	1 (tag) and 2 (tag+data) cycles
Shared unified L2 cache	1MB/tile, 8-way (2048 sets)
L2 cache hit time	2 (tag) and 6 (tag+data) cycles
Directory cache	256 sets, 4 ways (same as L1)
Directory cache hit time	1 cycle
Memory access time	160 cycles
Network Parameters (GARNET)	
Topology	2-dimensional mesh (4x4)
Routing technique	Deterministic X-Y
Flit size	16 bytes
Data and control message size	5 flits and 1 flit
Routing, switch, and link time	2, 2, and 2 cycles

3.2.2 GARNET Network Simulator

GARNET is a detailed cycle-accurate interconnection network model. GARNET models five-stage pipelined routers with virtual channel flow control. Other microarchitectural details are also modeled, such as flit-level input buffers or the routing logic among others.

GEMS L1 and L2 cache controllers and memory controllers communicate with each other, and GARNET captures network accesses through a built-in network interface (see Figure 3.1). A message is then broken into network-level units (flits) and routed to the appropriate destination. Response messages behave in a similar manner.

Furthermore, GARNET has the Orion power models [37] incorporated. GARNET feeds Orion with the amount of bit-switching, reads and writes on routers, arbiters activity, among others. Then, Orion returns the total network consumption, which is the sum of the energy consumed by all components, routers and links.

3.2.3 CACTI

CACTI (Cache Access and Cycle Time Information) [83] is an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. By integrating all these models together, users can have confidence that trade-offs between time, power, and area are all based on the same assumptions and, hence, are mutually consistent.

CACTI is continually upgraded as new technology is released. In this thesis we employ the version 6.5, which is a significantly enhanced version that primarily focuses on interconnect design. All TLB, cache and directory latencies, power consumption and area requirements for our implementations are obtained through CACTI calculations, assuming a 32nm technology.

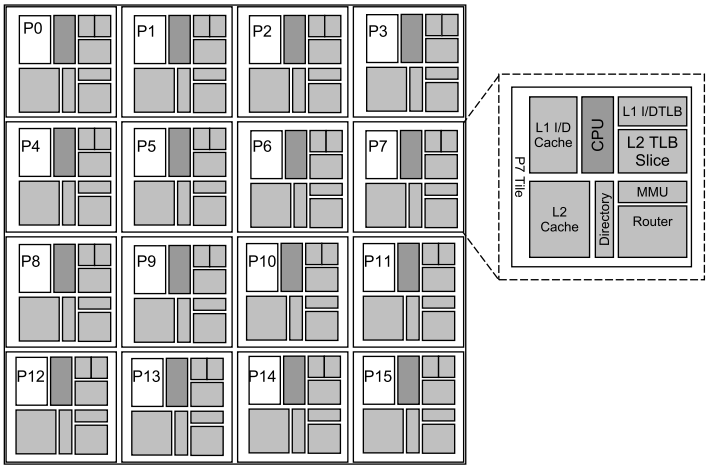
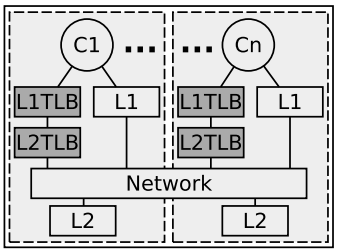


Figure 3.2: Baseline tiled CMP architecture with a two-level TLB structure.

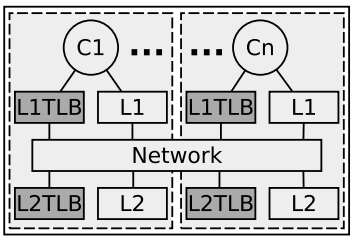
3.3 Simulated System

To evaluate our proposals we simulate 2x2, 4x4, and 4x8 replicated tiles organized as a tiled CMP (4x4 tiled CMP example in Figure 3.2), with two levels of TLB and two levels of cache on-chip. The private first-level (L1) TLB and cache are split into instruction and data. Then, a shared unified second-level (L2) cache which is physically distributed among the processing nodes, with in-order, single-issue processors. Although the mechanisms proposed in this thesis can be applied to caches with any indexing and tagging technique, for the sake of clarity we assume the common virtually indexed, physically tagged (VIPT) L1 caches.

Two different L2 TLB configurations have been evaluated: per-core private L2 TLBs (Figure 3.3a); and a distributed, shared L2 TLB organization (Figure 3.3b). The support for the various TLB organizations has been implemented in SLICC language. System parameters are shown in Table 3.1.



(a) Private two-level TLB organization.



(b) Shared TLB organization. Second level TLB is distributed and shared among cores.

Figure 3.3: Private and shared L2 TLB organizations.

The L2 TLB miss latency considers four memory references to walk the page table, as in the 48-bit x86-64 virtual address space. As previously noted, the cache and TLB latencies and energy consumption have been calculated using the CACTI tool [83], assuming a 32nm process technology. Configurations specific to each proposal are described in the corresponding chapter.

3.4 Metrics

The evaluation in this thesis comprises the study of the system performance, overall traffic consumption, and hardware overhead of our approaches. Performance evaluation is measured in terms of application's runtime (i.e., cycles) during its parallel phase. We employ cycles over instruction-per-cycle (IPC), since IPC is sensitive to the timing effects of multiprocessor workloads in full-system simulations, thus discouraging it for measuring the performance on coherence protocols. The instruction path of multi-threaded workloads can vary to a large extent due to the synchronization mechanisms employed by the operative system, for instance, when an application is stalled in a lock, dramatically increasing the count of completed instruction without progressing its execution whatsoever.

In addition, the traffic has been also reported. Specifically, the number of flits issued to the network has been accounted for and classified into TLB or cache traffic, discerning:

- *Cache Request*: number of flits issued as a consequence of coherence request messages.
- *Cache Response Control*: number of flits issued as a consequence of coherence control responses.
- *Cache Response Data*: number of flits issued as a consequence of coherence control responses including the data block information.
- *TLB Request*: number of flits issued as a consequence of TLB requests due to the classification mechanism.
- *TLB Response Control*: number of flits issued as a consequence of TLB control responses due to the classification mechanism.
- *TLB Response Translation*: number of flits issued as a consequence of TLB responses including the virtual-to-physical translation in order to accelerate the page translation resolution process.

Cache pressure is measured in order to determine the overheads and benefits of our proposed classification scheme. This way, cache misses are classified into 5C misses: Cold, Capacity, Conflicted, Coherence, and Coverage. Coverage misses [67] are those caused by invalidation on the directory cache due to the limited capacity, which cause an eviction of the block entry in the private caches. Moreover, we discern *Flushing* [68] misses, which are those caused by invalidated or evicted TLB entries, which in turn induces cache invalidations when employing a TLB-based classification approach.

We also evaluate the hardware overhead of our proposals. The overhead is shown in terms of area overhead, and is calculated both in terms of the number of bits and area percentage, calculated with CACTI 6.5 tool. A sensitivity study of the area overhead with the scale of the system has been also measured by varying the number of nodes in the CMP.

Furthermore, we measure the amount of data classified as private and shared at page level. The amount of private data is a good general metric for most classification approaches targeting classification-based data optimizations. As classification approaches in the evaluation operate at page granularity, we register every access to every page, and then the percentage of pages falling into each classification category is shown:

- *Private*: the page spent all of the execution time as private (i.e., accessed from a single thread).
- *Reclassified*: the page has been shared (i.e., accessed concurrently from two or more threads), and became private again afterwards.
- *Shared*: the page has been shared and remained in that state for the rest of the execution time.
  - *Shared-ReadOnly*: the page became shared and has never been written. Only for write-detection mechanisms.
  - *Shared-Written*: the page became shared and has been written. Only for write-detection mechanisms.

However, this metric is unfair for adaptive classification mechanisms, where shared pages are frequently reclassified as private. In order to capture the changing nature of data classification, the average percentage of cycles per page spent as private or shared has been also measured.

Finally, the best way to test the benefits of a classification scheme is through a classification-based data optimization. In this thesis we generally employ coherence deactivation (see Section 2.2.3), as our case study. Coherence deactivation greatly benefits from the accuracy of a classification approach. It can equally benefit from both a private/shared classification dichotomy and a private/read-only/shared-written classification scheme. Furthermore, the cost of data reclassification is low, and is extensively offset by its potential improvements. However, other data optimizations have been also employed to test the proposals in this thesis. Depending on the data optimization used to test the classification mechanism, different specific metrics are employed to show their particular effects.

All the experimental results reported in this thesis correspond to the parallel phase of each program. Benchmark checkpoints for applications running for some time have been employed to ensure that memory is warmed up, thus avoiding the effects of page faults. Then, each application runs again up to the parallel phase, where each thread is scheduled in a particular core. The application is then run with full detail during the initialization of each thread before starting the actual measurements. In this way, we warm up both TLBs and caches to avoid cold misses.

3.5 Benchmarks

Our proposals have been evaluated through a wide variety of parallel workloads from several benchmarks suites, covering different sharing patterns and sharing degrees: *Barnes*, *Cholesky*, *FFT*, *Ocean*, *Radiosity*, *Raytrace-opt*, *Volrend*, and *Water-NSQ* from the SPLASH-2 benchmark suite [85]; *Tomcatv* and *Unstructured* are two scientific benchmarks; *FaceRec*, *MPGdec*, *MPGenc*, and *SpeechRec* belong to the ALPBenchs suite [45]; *Blackscholes*, *Swaptions*, and *x264* come from PARSEC [17]; and finally, *Apache*, and *SPEC-JBB* are two commercial workloads [10].

Table 3.2 shows the input size of each application. Due to the slowness of full-system simulations, the input sizes have been appropriately scaled down. For scalability studies (32 cores or more) only a subset of the applications has been considered. All the reported experimental

Table 3.2: Benchmarks and input sizes

Benchmarks	Input size
SPLASH 2 (8)	
Barnes	8192 bodies, 4 time steps
Cholesky	Input file tk15.O
FFT	64K complex doubles
Ocean	258 × 258 ocean
Radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10
Raytrace-opt	Teapot
Volrend	Head
Waternsq	512 molecules, 4 time steps
Scientific benchmarks (2)	
Tomcatv	256 points, 5 time steps
Unstructured	Mesh.2K, 5 time steps
ALPBench (4)	
FaceRec	Script
MPGdec	525_tens_040.m2v
MPGenc	Output of MPGdec
SpeechRec	Script
PARSEC (4)	
Blackscholes	simmedium
Fluidanimate	simsmall
Swaptions	simmedium
x264	simsmall
Commercial Workloads (2)	
Apache	1000 HTTP transactions
SPEC-JBB	1600 transactions

results correspond to the parallel phase of these benchmarks. Following sections give a brief description of each application.

3.5.1 SPLASH 2

Barnes

The Barnes application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time-steps, using the Barnes-Hut hierarchical N-body method. Barned allows multiple particles per leaf cell. The computational domain is represented as an octree with leaves containing information on each body, and internal nodes representing space cells. Most of the time is spent in partial traversals of the octree (one traversal per body) to compute the forces on individual bodies. The communication patterns are dependent on the particle distribution and are quite unstructured. No attempt is made at intelligent distribution of body data in main memory, since this is difficult at page granularity and not very important to performance.

Cholesky

The blocked sparse Cholesky factorization kernel factors a sparse matrix into the product of a lower triangular matrix and its transpose. That is, given a positive definite matrix A, the program finds a lower triangular matrix L such that $A = LL^T$. Cholesky factorization proceeds in three steps: ordering, symbolic factorization, and numerical factorization. The numerical factorization is very efficient, due to the use of supernodal elimination techniques. Supernodes are sets of columns with nearly identical non-zero structures, and a factor matrix will typically contain a number of often very large supernodes. The primary data structure in this program is the representation of the sparse matrix itself. The data sharing patterns for each supernode are as follows. A supernode may be modified by several processors before it is placed on the task queue. Once this happens, it is read by a single processor and used to modify other supernodes. After it has completed all its modifications to other supernodes, it is no longer referenced by any processor.

The only interactions between processors occur when they attempt to dequeue tasks from the global task queue and when they attempt to perform a number of simultaneous supernodal modifications to the same destination column. Both of these cases are handled with locks.

FFT

The FFT kernel is a complex one-dimensional version of the radix-$\sqrt{n}$ sixstep FFT algorithm, which is optimized to minimize interprocessor communication. The data set consists of the complex data points to be transformed, and another complex data points referred to as the *roots of unity*. Both sets of data are organized as $\sqrt{n} \times \sqrt{n}$ matrices partitioned so that every processor is assigned a contiguous set of rows which are allocated in its local memory. Communication occurs in three matrix transpose steps, which require all-to-all interprocessor communication. Every processor transposes a contiguous submatrix of $\sqrt{n}/p \times \sqrt{n}/p$ from every other processor, and transposes one submatrix locally. The transposes are blocked to exploit cache line reuse. To avoid memory hotspotting, submatrices are communicated in a staggered fashion, with processor i first transposing a submatrix from processor $i + 1$, then one from processor $i + 2$, etc.

Ocean

The Ocean application studies large-scale ocean movements based on eddy and boundary currents. The algorithm simulates a cuboidal basin using discretized circulation model that takes into account wind stress from atmospheric effects and the friction with ocean floor and walls. The algorithm performs the simulation for many time steps until the eddies and mean ocean flow attain a mutual balance. The work performed every time step essentially involves setting up and solving a set of spatial partial differential equations. For this purpose, the algorithm discretizes the continuous functions by second-order finite-differencing, sets up the resulting difference equations on two-dimensional fixed-size grids representing horizontal cross-sections of the ocean basin, and solves these equations using a red-back Gauss-Seidel multigrid equation solver. Each task performs the computational steps on the section of the grids that it owns, regularly communicating with other processes. Synchronization is performed by using both locks and barriers.

Radiosity

This application computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method. A scene is initially modeled as a number of large input polygons. Light transport interactions are computed among these polygons, and polygons are hierarchically subdivided into patches as necessary to improve accuracy. In each step, the algorithm iterates over the current interaction lists of patches, subdivides patches recursively, and modifies interaction lists as necessary. At the end of each step, the patch radiosities are combined via an upward pass through the quadtrees of patches to determine if the overall radiosity has converged. The main data structures represent patches, interactions, interaction lists, the quadtree structures, and a BSP tree which facilitates efficient visibility computation between pairs of polygons. The structure of the computation and the access patterns to data structures are highly irregular. Parallelism is managed by distributed task queues, one per processor, with task stealing for load balancing. No attempt is made at intelligent data distribution.

Raytrace-opt

This application renders a three-dimensional scene using ray tracing. A hierarchical uniform grid (similar to an octree) is used to represent the scene, and early ray termination and antialiasing are implemented, although antialiasing is not used in this study. A ray is traced through each pixel in the image plane, and reflects in unpredictable ways off the objects it strikes. Each contact generates multiple rays, and the recursion results in a ray tree per pixel. The image plane is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used with task stealing. The major data structures represent rays, ray trees, the hierarchical uniform grid, task queues, and the primitives that describe the scene. The data access patterns are highly unpredictable in this application. The application has been optimized by removing a lock acquisition for a ray ID which is not used.

Volrend

This application renders a three-dimensional volume using a ray casting technique. The volume is represented as a cube of voxels (volume elements), and an octree data structure is used to traverse the volume quickly. The program renders several frames from changing viewpoints, and early ray termination and adaptive pixel sampling are implemented, although adaptive pixel sampling is not used in this study. A ray is shot through each pixel in every frame, but rays do not reflect. Instead, rays are sampled along their linear paths using interpolation to compute a color for the corresponding pixel. The partitioning and task queues are similar to those in Raytrace. The main data structures are the voxels, octree and pixels. Data accesses are input-dependent and irregular, and no attempt is made at intelligent data distribution.

Water-Nsquarred

This application evaluates forces and potentials that occur over time in a system of water molecules. The forces and potentials are computed using an $O(n^2)$ algorithm (hence the name), and a predictor-corrector method is used to integrate the motion of the water molecules over time. At each time step, the processors calculate the interaction of the atoms within each molecule and the interaction of the molecules with one another. For each molecule, the owning processor calculates the interactions with only half of the molecules ahead of it in the array. Since the forces between the molecules are symmetric, each pair-wise interaction between molecules is thus considered only once. A process updates a local copy of the particle accelerations as it computes them, and accumulates into the shared copy once at the end. Most synchronization is done using barriers, although there are also several variables holding global properties that are updated continuously and are protected using locks.

3.5.2 ALPbench

FaceRec

FaceRec uses the CSU face recognizer. Face recognizers recognize images of faces by matching a given input image with images in a given database. This application uses a large database (called subspace) that consists of multiple images. The objective of phase recognition is to find the image in the subspace that best matches a given input image (i.e., a single column vector with thousands of rows). A match is determined by taking the *distance* or difference between two images. In this application, differently from the base CSU face recognizer, a separate input image is compared with each image in the subspace to emulate a typical face recognition scenario (e.g., a face of a subject is searched in a database).

FaceRec is first trained with a collection of images in order to distinguish faces of different persons. Moreover, there are multiple images that belong to the same person so that the recognizer is able to match face images against different expressions and lighting conditions. Then, the training data is written to a file so that it can be used in the recognition phase. At the start of the recognition phase, the training data and the image database are loaded.

MPEG-2 encoder

MPGenc converts video frames into a compressed bit-stream. A video encoder is an essential component in VCD/DVD/HDTV recording, video editing, and video conferencing applications. Many recent video encoders like MPEG-4/H.264 use similar algorithms. A video sequence consists of a sequence of input images, which are in the YUV format; i.e., one luminance (Y) and two chrominance (U,V) components. Each encoded frame is characterized as an I, P, or B frame. Each frame consists of 16x16 pixel macroblocks. Each macroblock consists of four 8x8 luminance blocks and two 8x8 chrominance blocks, one for U and one for V. I frames are temporal references for P and B frames and are only spatially compressed. On the other hand, P frames are predicted based on I frames, and B frames are predicted based on neighboring I and P frames.

MPEG-2 decoder

MPGdec decompresses a compressed MPEG-2 bit-stream. Video decoders are used in VCD/DVD/HDTV playback, video editing, and video conferencing. Many recent video decoders, like MPEG-4/H.264, use similar algorithms. Major phases for MPGdec include variable length decoding (VLD), inverse quantization, IDCT, and motion compensation. The decoder applies the inverse operations performed by the encoder. First, it performs variable-length Huffman decoding. Second, it inverse quantizes the resulting data. Third, the frequency-domain data is transformed with IDCT to obtain spatial-domain data. Finally, the resulting blocks are motion-compensated to produce the original pictures.

SpeechRec

SpeechRec converts speech into text. Speech recognizers are used with communication, authentication, and word processing software and are expected to become a primary component of the human-computer interface. The application has three major phases: feature extraction, Gaussian scoring, and searching the language model / dictionary. First, the feature extraction phase creates 39-element feature vectors from the speech sample. The Gaussian scoring phase then matches these feature vectors against the phonemes in a database. It evaluates each feature vector based on the Gaussian distribution in the acoustic model (Gaussian model) given by the user. In a regular workload, there are usually 6000+ Gaussian models. The goal of the evaluation is to find the best score among all the Gaussian models and to normalize other scores with the best one found.

3.5.3 PARSEC

Blackscholes

The blackscholes application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE)

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

where V is an option on the underlying S with volatility σ at time t if the constant interest rate is r. There is no closed-form expression for the Black-Scholes equation and as such it must be computed numerically. The program divides the portfolio into a number of work units equal to the number of threads and processes them concurrently. Each thread iterates through all derivatives in its contingent. The input set of choice has 16,384 options.

Fluidanimate

This Intel RMS application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. Its output can be visualized by detecting and rendering the surface of the fluid. The force density fields are derived directly from the Navier-Stokes equation. Fluidanimate uses special-purpose kernels to increase stability and speed. The scene geometry employed by Fluidanimate is a box in which the fluid resides. All collisions are handled by adding forces in order to change the direction of movement of the involved particles instead of modifying the velocity directly. Every time step, Fluidanimate executes five kernels. The input set includes 35,000 particles and 5 frames.

Swaptions

The swaptions application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The HJM framework describes how interest rates evolve for risk management and asset liability management for a class of models. Its central insight is that there is an explicit relationship between the drift and volatility parameters of the forward-rate dynamics in a noarbitrage market. Because HJM models are non-Markovian the analytic approach of solving the PDE to price a derivative cannot be used. Swaptions therefore employs Monte Carlo (MC) simulation to compute the prices. The program stores the portfolio in the swaptions array. Each entry corresponds to one derivative. Swaptions partitions the array into a number of blocks equal to the number of threads and assigns one block to every thread. The input set performs 32 swaptions and 10,000 simulations.

x264

The x264 application is an H.264/AVC (Advanced Video Coding) video encoder. It is based on the ITU-T H.264 standard which is now part of ISO/IEC MPEG-4. In that context the standard is also known as MPEG-4 Part 10. H.264 describes the lossy compression of a video stream. H.264 encoders and decoders operate on macroblocks of pixels which have the fixed size of 16 x 16 pixels. Various techniques are used to detect and eliminate data redundancy. The most important one is motion compensation. It is employed to exploit temporal redundancy between successive frames. Motion compensation is usually the most expensive operation that has to be executed to encode a frame. It has a very high impact on the final compression ratio. The videos used for the input sets have been derived from the uncompressed version of the short film *Elephants Dream*. The number of frames determines the amount of parallelism. In this work we employ 640 x 360 pixel images ($\frac{1}{3}$ HDTV resolution), and 8 frames.

3.5.4 Other Scientific and Commercial Workloads

Tomcatv

Tomcatv is a 200-line mesh generation program. Tomcatv contains several loop nests that have dependences across the rows of the arrays and other loop nests that have no dependences. Since the base version always parallelizes the outermost parallel loop, each processor accesses a block of array columns in the loop nests with no dependences. However, in the loop nests with row dependences, each processor accesses a block of array rows. As a result, there is little opportunity for data re-use across loop nests. Also, there is poor cache performance in the

row-dependent loop nests because the data accessed by each processor is not contiguous in the shared address space.

Unstructured

Unstructured is a computational fluid dynamics application that uses an unstructured mesh to model a physical structure, such as an airplane wing or body. The mesh is represented by nodes, edges that connect two nodes, and faces that connect three or four nodes. The mesh is static, so its connectivity does not change. The mesh is partitioned spatially among different processors using a recursive coordinate bisection partitioner. The computation contains a series of loops that iterate over nodes, edges and faces. Most communication occurs along the edges and faces of the mesh. Synchronization in this application is accomplished by using barriers and an array of locks.

Apache

Apache is a popular open-source web server used in many internet/intranet environments. In this application, the focus is on static web content serving. Apache uses the Scalable URL Request Generator (SURGE) as the client. SURGE generates a sequence of static URL requests which exhibit representative distributions for document popularity, document sizes, request sizes, temporal and spatial locality, and embedded document count. The application runs 10 SURGE client threads per processor, and set the client think time to be zero. The experiments used a repository of 2000 files, with total size of approximately 50 MB, generated by SURGE using its default parameters. The system was warmed up for 80,000 transactions, and the results were based on runs of 1,600 transactions.

SPEC-JBB

The SPEC-JBB benchmark evaluates the performance of server side Java by emulating a three-tier client/server system. SPEC-JBB has been developed from the ground up to measure performance based on the latest Java application features, including: both a pure throughput metric and a metric that measures critical throughput; support for multiple run configurations, enabling users to analyze and overcome bottlenecks; and support for virtualization and cloud environments.

Chapter 4

TLB-based Classification Mechanisms

This chapter introduces a mechanism that is capable of classifying data into a private-shared scheme based on the presence of page translations stored in the system TLBs. TLB-based classification avoids some of the major problems encountered on state-of-the-art classification techniques.

4.1 Introduction

The basic mechanism introduced in this chapter represents the starting point for this work.[1] The proposed classification scrutinizes the network upon TLB misses in order to discover other TLBs storing the page translation entry (i.e., other processors that are possibly accessing the same page) and discern the sharing status. In addition, this network exploration allows retrieving the page translation beforehand from a sharer, thus accelerating the page translation process.

Checking (and possibly updating) the sharing status upon every TLB miss also allows accounting for temporarily private pages (e.g., as a consequence of thread migration). Since data access patterns change throughout different phases of the application lifetime [39, 78], we claim that a temporal-aware detection mechanism is fundamental to achieve a good accuracy when detecting private pages. Note that the more private data detected, the more benefits can be obtained when the classification scheme is applied to a classification-based data optimization.

Lastly, the sharing information collected after the miss is stored in the TLB. Thus, avoiding adding storage overhead in a directory-like structure [62, 33, 11] or in the page table [43, 22, 31, 71, 47].

[1] SnoopingTLB was first proposed by Ros *et al.* [68], and then extended and re-evaluated in the scope of this thesis.

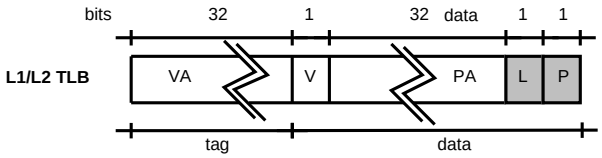


Figure 4.1: L1/L2 TLB entry with the extra fields in gray.

4.2 TLB Miss Resolution through TLB-to-TLB Transfers

First, we describe a simple mechanism that allows cores to retrieve information about the privacy of page and address translations from another core's TLB instead of the page table. "Walking" the page table, often broken down into several levels, may imply several memory references. This mechanism, although simpler, has some similarities with *Synergistic TLBs* [81]. This way, upon a TLB miss, getting the page information from a remote TLB is faster compared to performing a complete page table walk.

TLB-to-TLB transfer works as follows. On a TLB miss, a page table walk process is started in order to get the address translation from the page table. Simultaneously, the core snoops the other TLBs in the system by issuing a *page_info* request. When a core receives the page_info request, it checks its TLB and, in case of finding the page entry, the translation is sent to the requester by means of a short response message. Since most TLBs employ a Process Identifier (PID) field in their entry formats in order to avoid flushing the TLB on every context switch, we can employ the PID information to adequately distinguish different virtual spaces from different applications when sharing a page translation through the TLB-to-TLB miss resolution mechanism. When the first positive response is received, the page address translation is stored in the TLB and the memory request can proceed, therefore canceling the page table walk. Although upon every TLB miss the described mechanism snoops other TLBs, TLB misses are not frequent, thus keeping traffic overhead and energy consumption low and not jeopardizing its scalability for small- or medium-scale systems. However, broadcast messages are not reliable for larger systems, as they are do not scale well with the number of cores.

4.3 Snooping TLB-based Private-Shared Classification

Snooping TLB-based classification deals with the fact that data can be requested by multiple cores and stored in their private caches during the application runtime although being actually private (due to thread migration) or not shared at the same time (due to different phases of applications). These data access patterns where a private data block is replicated in more than one cache as a consequence of the owner migration process is also known as passive sharing [30, 28]. The proposed detection mechanism (i) is aware of the temporality in memory references and discerns passive sharing patterns, (ii) employs techniques to solve TLB misses from other CMP cores (i.e., TLB-to-TLB transfers), and (iii) does not require extra hardware structures.

As noted in Section 3.3 we assume VIPT caches in the next sections. Nonetheless, our proposal is directly applicable to any other cache where the access to the TLB is required.

Table 4.1: Response messages and state transitions for TLB requests

State (in TLB or MSHR)	Address translation	Response type	Next state
Not Present	NO	Not_Present	Not Present
Requested (- or S)	NO	Requested	Requested (S)
Present (P or S)	YES	Present	Present (S)

4.3.1 Classification based on TLB-to-TLB Communication

TLB-based classification stores the sharing information along with the address translation in the TLB entries. Figure 4.1 shows how TLB entries are extended. The *Lock* (L) bit allows blocking memory accesses to the corresponding page while the sharing status is uncertain (e.g., while a TLB miss request is ongoing). Each TLB entry has a Private (P) bit that indicates whether the page is private or shared. The P bit for a page is set when there is just one core's TLB caching the translation (i.e., it does not cause a TLB miss when requesting page blocks). When several TLBs hold simultaneously an entry for that page, their P bits are clear.

On memory references, before accessing the L1 cache, a TLB lookup is performed to get the physical address of the requested block, as depicted in Figure 4.2. Table 4.1 summarizes the possible TLB entry states, the type of the response messages provided by the core's TLBs to page_info requests, and the information included in the *responses* (e.g., the address translation).

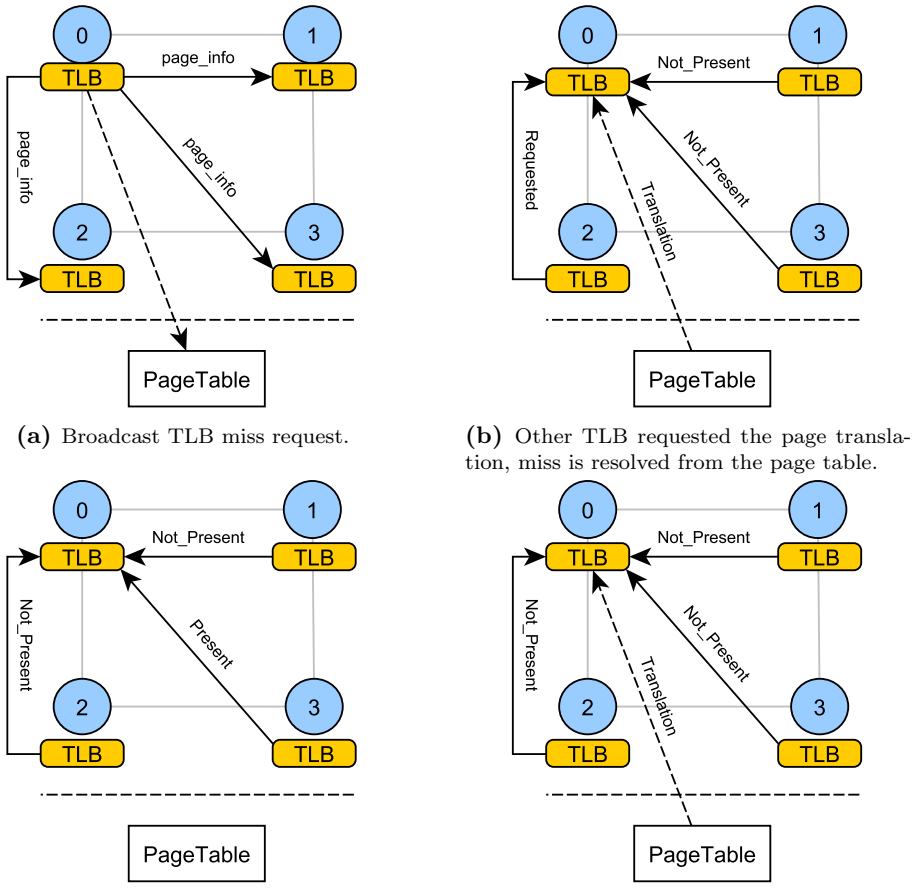
If a TLB miss takes place, a page_info request is sent to the other TLBs (Figure 4.2a). The goal of this request is to get both the address translation and the information about the use of the page by the other cores. If any of the receiving cores is presumably going to access the page[2], then the TLB entry is marked as shared. Figure 4.2b shows how TLB in core 2 requested the page translation when receives the translation request from core 0. Thus, page is shared, although the translation is retrieved from the page table. Conversely, Figure 4.2c shows how TLB in core 3 is holding a valid translation entry, which is provided to the requestor in core 0, resolving the page classification to shared. Otherwise, if no other TLB in the system is holding a valid page translation, the page will be classified as private and the miss will be resolved from the page table (Figure 4.2d).

4.3.2 TLB Coherency for Data Classification

Page-sharing information must be kept coherent in all the core's TLBs. To keep this information coherent, we use the transition diagram shown in Figure 4.3. Pages can be in three situations: (i) the page translation is not paged in any TLB; (ii) only one TLB holds the entry as private; (iii) one or several TLBs hold the entry, all of them as shared.

Figure 4.3 shows the TLB state transitions depending on the incoming requests and responses to guarantee TLB coherence. When a TLB entry is *Not Present* in a given core and a local TLB request is issued by that core, the entry transitions to the *Requested* state. On the reception of remote page_info requests in this state, the page transitions to the *Requested S* state. This way, when two or more TLBs send page_info requests at the same time, they will answer to each other as accessing the page and TLB coherence is guaranteed since every TLB will see the page as shared. The *Requested S* transitions to *Present S* once all responses have been received, regardless of their content. On the other hand, the *Requested* state transitions to

[2]Techniques about prediction accesses will be discussed in the following chapters.



(a) Broadcast TLB miss request.

(b) Other TLB requested the page translation, miss is resolved from the page table.

(c) TLB in core 3 holds the translation, and provides it to the requestor.

(d) No other TLB holds a valid translation, miss is resolved from the page table.

Figure 4.2: Different outcomes for TLB-to-TLB requests from C0.

Present S or *Present P* depending on the responses received from the other TLBs. The TLB state transitions to *Present S* if any other TLB is currently using the page (e.g., the TLB has the page translation stored), and to *Present P* otherwise. TLB evictions cause transitions to the *Not Present* state, but remaining TLBs are not notified about the evictions.

4.3.3 TLB-cache Inclusion Policy

Since TLBs maintain per-page sharing information that finally will affect the coherence management of memory blocks[3], our proposal prevents data blocks from being stored in the core's L1 cache if the translation of their page is not stored in the TLB, as private pages are not aware of other cache copies. Hence, no copies of the blocks belonging to a private page can exist in other L1 cache. Specifically, when a TLB entry transitions to *Not Present*, the cached blocks belonging to the corresponding page are evicted from L1 cache (and written back to the LLC when dirty). This policy would hardly affect system performance, provided that (i) a TLB entry transitions to *Not Present* when the page has not been accessed for some time, so most page blocks may have been already evicted from the cache, and (ii) it is likely that no block within this page will be accessed in the near future.[4]

[3]This assumption applies to Coherence Deactivation, among others. However, the next section will show specific inclusion policies and actions for different classification-based optimizations.

[4]However, as we will see in next chapters, aggressive usage predictor can increase the cache (and the TLB) miss rate.

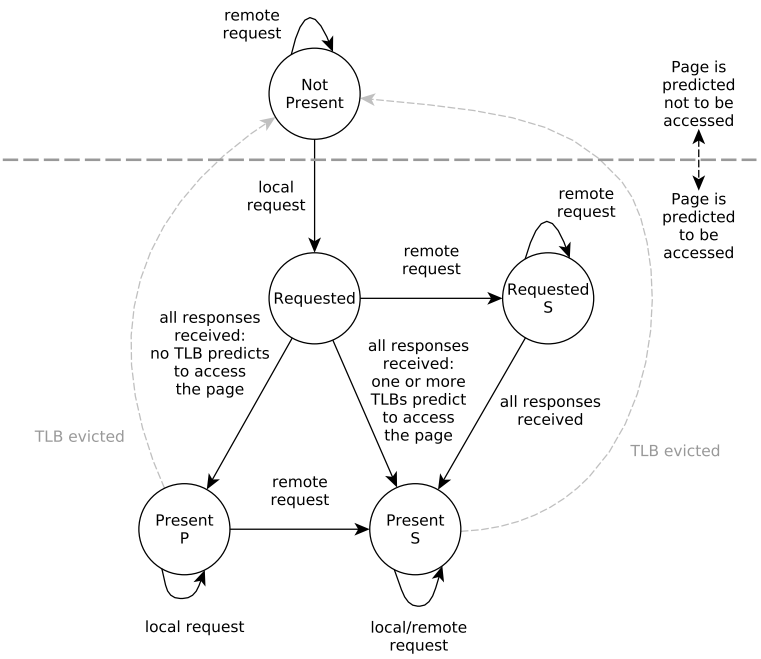


Figure 4.3: TLB state transition diagram

When evicting the blocks from the cache, the access to the corresponding TLB entry is blocked. To do that, each TLB entry includes a *Lock* (L) bit (Figure 4.1).

4.3.4 Actions Required upon Classification Changes

Our temporal-aware page classification mechanism can be applied to perform different classification-based optimizations (see Section 2.2 for more details). Depending on the optimization, when a page classified as private becomes shared, it may be required to trigger a recovery procedure to restore the coherence state of the blocks belonging to the page. This recovery procedure depends on the optimization for which the private-shared classification is being applied to. Also, the TLB-cache inclusion policy may vary depending on the purpose of the private-shared classification. Basically, it is necessary to perform the L1 flushing and the action required to restore the "normal" status of the lines, which usually matches with the recovery action. These actions are summarized in Table 4.2. In this case, Reactive NUCA requires LLC flushing on every transition to shared, as the address mapping varies with the classification. Conversely, Subspace Snooping requires no action when promoting the classification to shared.

Table 4.2: Actions due to TLB-cache inclusion and recovery

Application scenario	**TLB-cache inclusion**	**Recovery (P → S)**
Coherence Deactivation	L1 flushing	L1 flushing
Reactive NUCA	L1 and LLC flushing	LLC flushing
Subspace Snooping	L1 flushing	No action
VIPS	L1 flushing	L1 flushing

4.3.5 Multilevel Private TLBs

The TLB hierarchy of contemporary architectures includes split data and instruction private L1 TLBs and unified private L2 TLBs. Among the most common architectures adopting this TLB hierarchy we find AMD's K7, K8, and K10, Intel's i7 and Xeon, ARMv7 and ARMv8, or the HAL SPARC64-III [1, 3, 5, 6, 4]. So far, our proposal assumed single-level TLBs. This section explains the implications of implementing a Snooping TLB-based Classification approach in systems with private, multilevel TLB organization.

Main considerations

Differently from a single-level TLB classification scheme, in a multilevel TLB hierarchy, a L1 TLB miss does not trigger a TLB-to-TLB request. Instead, the private L2 TLB is consulted. On an L2 TLB hit, the page translation is obtained and cached in the L1 TLB, thus resolving the L1 miss. On a L2 TLB miss, the TLB-to-TLB request is initiated along with the access to the page table.

TLB-to-TLB requests lookup both the L1 and the L2 TLB looking for in-use TLB entries. TLB lookup can be performed in parallel. If there is a hit in any of the TLBs, then they respond with the page translation to the requester.

Implementation details

This section discusses the key implementation choices made for the proposed TLB-based classification scheme considering private two-level TLB hierarchies.

TLB inclusion policy. We employ an exclusive policy between the L1 and the L2 TLBs, since it maximizes the TLB capacity. Note that an increase in the number of TLB misses can dramatically degrade system performance. The downside of keeping exclusive TLBs is that, upon the reception of a TLB-to-TLB request, both TLB levels have to be accessed. We opt for performing this operation in parallel, thus incurring only the access latency of the L2 TLB, but at the cost of increasing the energy consumed when hitting in the L1 TLB. Since TLB-to-TLB requests are not frequent, we believe that this is the most appropriate design choice.

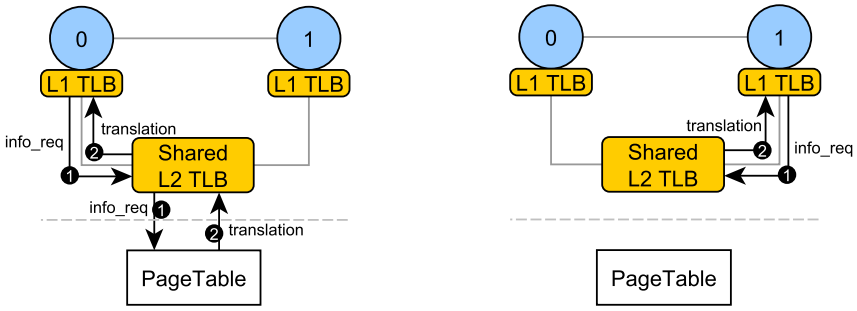
TLB consistency. The TLB hierarchy is shootdown-aware. As both TLB levels implement an exclusive policy, they can be checked in parallel when looking for the translation entry to be invalidated. Furthermore, this approach avoids incurring wrong (outdated) page classification, by flushing both the TLB and the cache.

4.4 TLB-based Classification with Distributed Shared TLBs

Current application memory footprints demand deeper TLB hierarchies. Using a centralized shared last-level TLB, as an alternative to purely private TLB organization, provides high performance improvement, specially on parallel applications [15]. However, prior work propose a centralized TLB level structure, which is not a scalable design.

In this section, we exploit the use of a distributed shared L2 TLB similar to the NUCA cache organization [42], with the aim of supporting data classification while avoiding some of the overheads associated to the snooping TLB-based approach. Such a distributed organization

has been previously suggested [47], however it has not been extensively explored. The baseline CMP considered in this work includes per-core L1/L2 TLBs, memory management unit (MMU), L1/L2 caches, and directory cache (Figure 3.2).



(a) L1 TLB request missing in the shared L2 TLB.

(b) L1 TLB request hitting in the shared L2 TLB.

Figure 4.4: Shared L2 TLB basic working scheme.

Figure 3.3 depicted how different TLB hierarchies are logically connected for both private (Figure 3.3a) and distributed shared (Figure 3.3b) last-level (L2) TLBs. The interconnection network assumed is a mesh topology. More details on the system architecture are described in Chapter 3. Figure 4.4 shows how a translation request from an L1 TLB misses in the L2 TLB (Figure 4.4a), accessing the page table in step 1. Then, when the translation is resolved in step 2, the shared L2 TLB stores a copy and responds to the requester TLB. Finally, a translation request for the same page from other TLB (or the same TLB after its eviction) is resolved by the shared L2 TLB (Figure 4.4b), thus exploiting inter-sharing access patterns. This way, by employing a shared L2 TLB, the classification is naturally obtained through unicast messages to the corresponding L2 TLB bank, issued upon every L1 TLB miss. The sharing status is stored in the TLB structure and accessible for all requesting cores through the network.

4.4.1 DirectoryTLB: Shared TLB for Data Classification

In this section we leverage a shared L2 TLB in order to track the sharing information, thus naturally classifying memory accesses into private or shared at page granularity. To this end, a counter (namely sharers counter) is associated with each second-level entry, recording up-to-date information about potential page sharers. We refer to this proposal as DirectoryTLB, as the shared TLB level acts as a directory for classification information, although it only tracks the number of page sharers, not the core identifier. Therefore, DirectoryTLB scales better compared to a cache directory for coherence purposes.

Figure 4.5 outlines the actions required by the proposed classification scheme in order to resolve memory operations through the TLB hierarchy, including recovery operation explained in Section 4.4.2.

Upon a memory access on a given core, prior to accessing the cache hierarchy, a TLB lookup is performed, looking for the virtual-to-physical address translation. On a L1 TLB hit, the sharing status information for that page is retrieved. Otherwise, on an L1 TLB miss, the L2 TLB is consequently accessed. Requesting the translation to the L2 TLB increases the sharers counter. Therefore, whether there is a miss in the L2 TLB (and thus the page table in main memory has to be accessed), or there is a hit and no other L1 TLB is currently holding the page (i.e., there is no other potential sharer), the page ends with a single sharer, and is thus marked as private. Otherwise, if the page had one or more sharers upon the reception of an L1 TLB miss request, it is marked as shared. In either case, the sharing status is specified

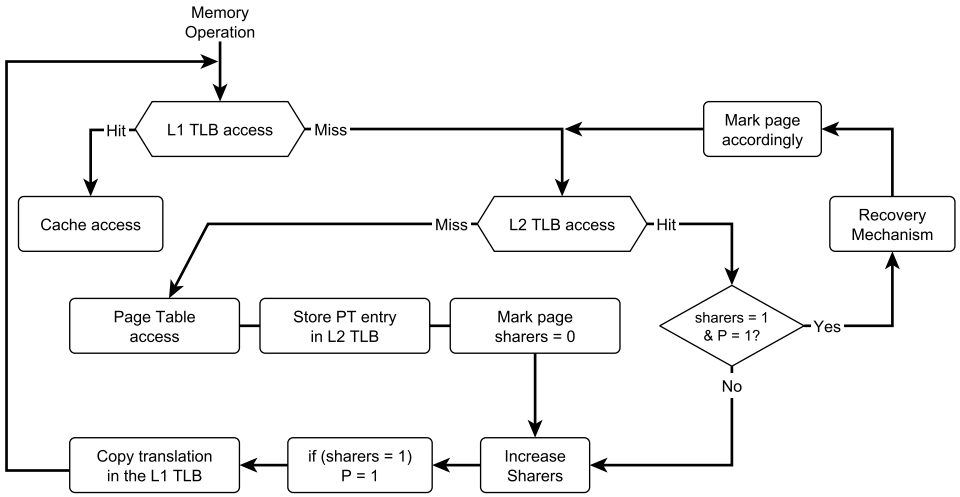


Figure 4.5: Block diagram of the general working scheme under a memory operation with DirectoryTLB.

alongside the response message containing the virtual-to-physical page translation to the upper TLB hierarchy level. The translation is finally stored in both TLB levels.

When the L2 TLB suffers an eviction, the sharing status is lost. In this case, in order to avoid classification inconsistencies, all L1 TLB entries holding the related page must be evicted (invalidating the corresponding L1 cache blocks due to the TLB-cache inclusion policy). Therefore, L1 and L2 TLBs implement an inclusive policy between them in order to ensure classification consistency.

Shared TLB entries require some extensions to support classification, as illustrated in Figure 4.6. A *Private* (P) bit specifies whether the page is private (bit set) or shared (bit clear). A *sharers* counter field tracks the number of current sharers of a page, allowing shared-to-private page reclassification. The *sharers* field is updated after every miss or eviction on L1 TLBs. This implies that L1 TLB evictions must be notified to the L2 TLB. This is essential to accurately unveil reclassification opportunities when the page ceases to have sharers. Since TLB misses and evictions are not frequent these notifications do not noticeably increase network traffic. The page sharing status is updated in the L2 TLB according to the sharers counter. Finally, a *keeper* field contains the identity of the holder of a private TLB entry. The *keeper* field helps to avoid broadcasts when updating the sharing state in the private TLBs when a transition to shared occurs. The *keeper* is updated every time the L2 TLB receives a

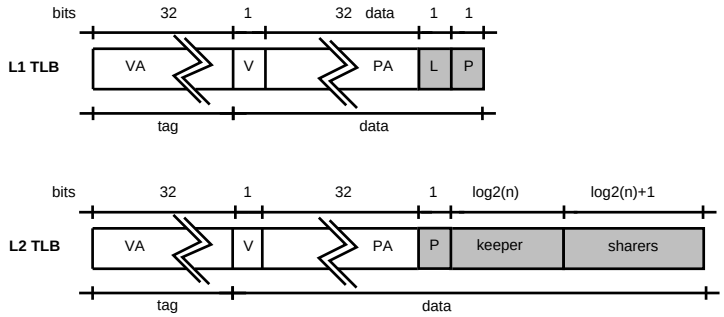


Figure 4.6: DirectoryTLB entries, with the extra field required for classification in gray.

request for a page and the current number of sharers is zero. In this case, the requester core's TLB becomes the new *keeper*.

The total storage resources required by this information is $1+log_2(N)$ bits for the *sharers* field (counting from 0 to N sharers, both inclusive), $log_2(N)$ bits for the *keeper* field, and one bit for P, where N is the number of cores in the system. This adds up to $2 + log_2(N) * 2$ total bits. Since the TLB entry data field often contains some unused bits [12] that are reserved, hardware overhead may be avoided by taking advantage of them. Anyhow, the shared second level TLB entry format requires only 10 bits assuming a 16-core CMP, or 22 bits for a 1024-core CMP. Therefore, for a TLB to support our proposed classification approach, the area overhead increases logarithmically with the system size, representing ∼14% or ∼23% of the L2 TLB area for a 16-core or a 1024-core CMP respectively, according to CACTI.

4.4.2 Classification Transitions with DirectoryTLB

This section reviews the actions required upon classification changes with our DirectoryTLB classification scheme.

When a given private page transitions to a shared state, P bits in the upper TLB level have to be updated, and some actions might be required depending on the data optimization, thus initiating a recovery operation (see Figure 4.5). Thus, a special *recovery* request is issued to the current page *keeper*. If an L1 TLB receives a *recovery request*, it performs the required actions. Then, the sharing status in the L1 TLB is securely set to shared and a *recovery response* is sent to the corresponding L2 TLB bank. Upon the reception of the *recovery response*, page sharing status is updated to shared in the L2 TLB.

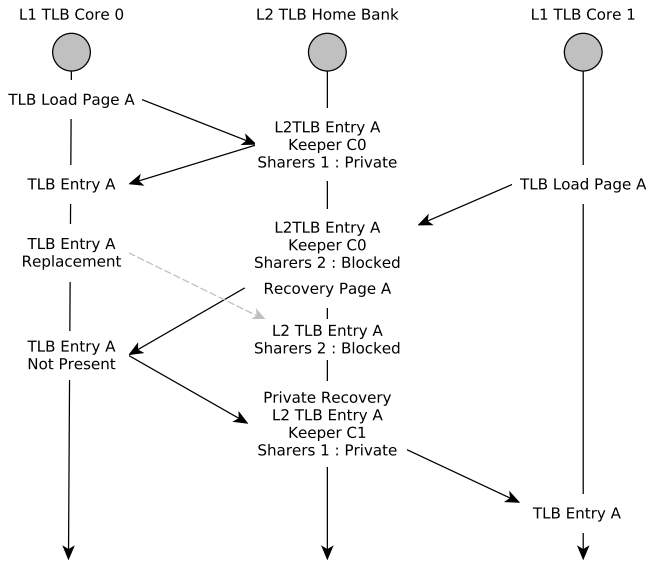


Figure 4.7: Coherence recovery mechanism resolved to Private. Page A in the keeper (*C0*) is evicted prior to receiving the Recovery message and thus, the Recovery is resolved to Private and the keeper is updated.

During a recovery process, a race may occur if the keeper evicts its TLB entry due to a conflict. Therefore, if the recovery request misses on the keeper's L1 TLB, a special *recovery* response is sent back with no further actions required (Figure 4.7). Then, the requester becomes the new keeper and the page remains as private.

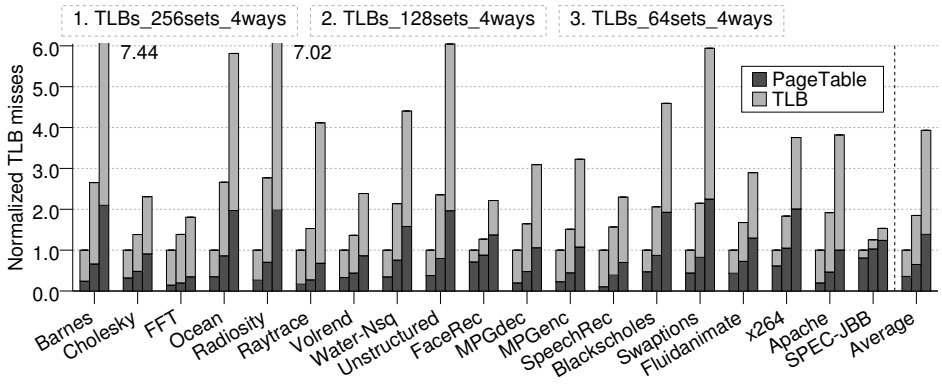


Figure 4.8: Distribution of TLB misses resolved by either other TLBs or the page table

4.5 Experimental Results

This section analyzes the proposed TLB-based classification schemes for different TLB structures. First, we show how the proposed TLB-to-TLB miss resolution mechanism behaves, and compare it with a shared last-level TLB structure. In order to test the virtues of TLB-based classification mechanism, we compare it with an OS-based classification approach [22]. Finally, we employ Coherence Deactivation as our study case to show the benefits of the classification scheme. System configuration and benchmarks are detailed in Chapter 3, any specific configurations are further detailed.

4.5.1 Inter-core Sharing Patterns: Fast TLB Miss Resolution

This section compares different TLB configurations behavior, focusing on how they exploit inter-core sharing patterns, prior to data classification.

TLB-to-TLB transfers

This section analyzes the benefits and overheads of the TLB-to-TLB miss resolution mechanism by means of a sensitivity study of single level TLB structures, with sizes ranging from 256 sets to 64 sets (all of them are 4-way associative).

TLB-to-TLB miss resolution avoids accessing the page table. Figure 4.8 shows the number of TLB misses normalized to those caused when using a 256-set TLB. Each bar shows the distribution of misses resolved by another TLB or the page table. We can see that the number of TLB misses increases for smaller TLBs. The number of TLB-to-TLB resolutions also increases, thus making its percentage almost independent of the TLB size. Particularly, 61.7%, 64.8%, and 64.8% of misses are resolved from a neighbor TLB for 256-, 128-, and 64-set TLBs, respectively.

The TLB-to-TLB miss resolution saves accesses to the page table, and therefore, reduces the TLB miss latency. This reduction translates into the improvements in execution time shown in Figure 4.9. Each bar shows the reduction in the number of cycles when compared to a configuration with the same TLB size but that does not implement TLB-to-TLB transfers. As we can observe, smaller TLBs implies more misses, and consequently, larger improvements in

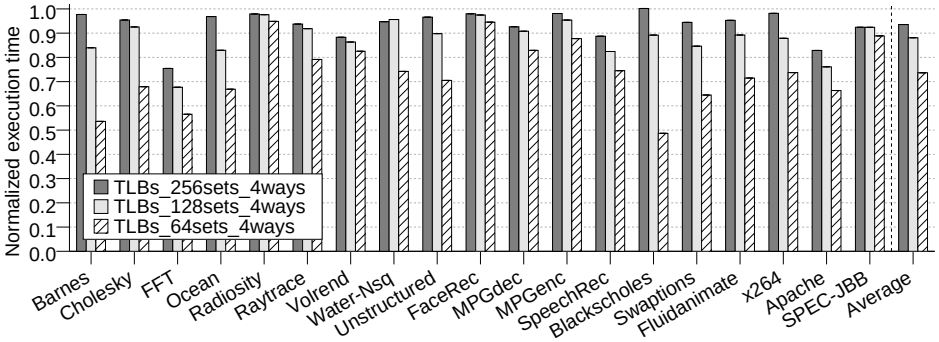


Figure 4.9: Improvements in execution time when using TLB-to-TLB transfers

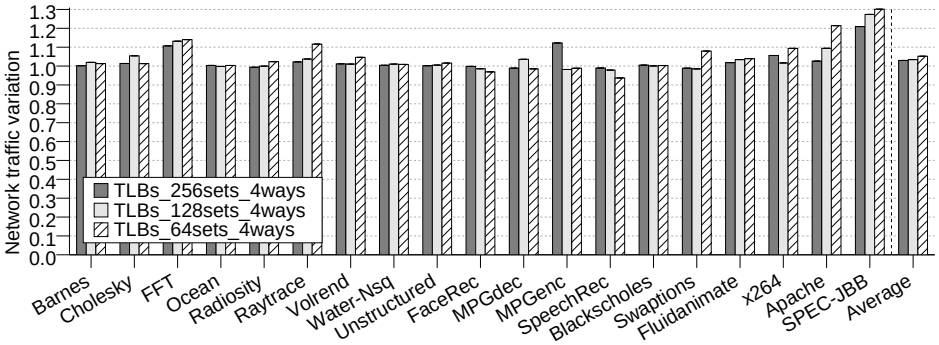


Figure 4.10: Variations in network traffic when using TLB-to-TLB transfers

execution time when using the TLB-to-TLB transfer mechanism. On average, execution time is reduced by 6.5%, 11.9%, and 26.4% for 256-, 128-, and 64-set TLBs, respectively.

On the other hand, the network traffic can increase due to the extra page_info requests issued. Figure 4.10 shows the normalized network traffic. Since TLB misses are not frequent, the increase in traffic is low. Only *Spec-JBB* and *Apache* (and for small TLB sizes) reach an overhead of 20%. This is because commercial applications have a high TLB-versus-cache miss rate. Overall, the overhead in traffic of the TLB resolution mechanism is just 2.9%, 3.4%, and 5.2% for 256-, 128-, and 64-set TLBs, respectively.

TLB-to-TLB Transfers against Shared TLB

Now we compare: (i) a system with a single-level TLB with TLB-to-TLB transfers used to accelerate TLB misses (*TLB*); (ii) a system with per-core private L2 TLBs and TLB-to-TLB transfers to accelerate L2 TLB misses (*P2TLB*); and (iii) a system with private L1 TLBs and distributed shared L2 TLBs (*S2TLB*). The size and associativity of the TLBs in this section are those of the baseline system (Chapter 3).

Figure 4.11 shows the number of accesses to the page table per kilo instructions, i.e., TLB misses that are resolved in the page table (TLB-MPKI). Notice that the y axis is plotted on a logarithmic scale in order to discern the different magnitudes of each application. When the

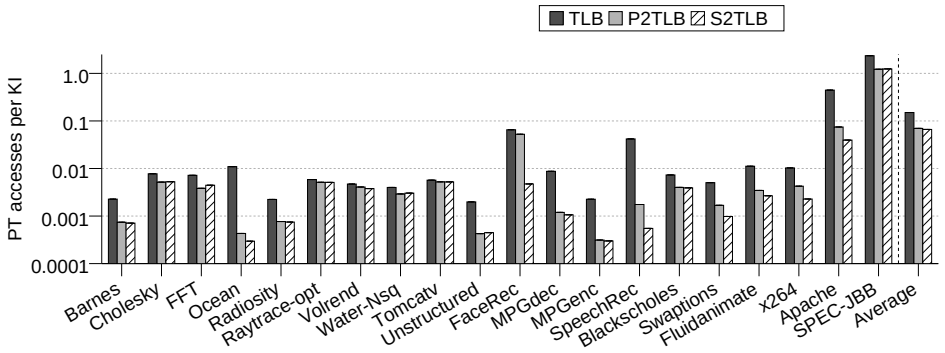


Figure 4.11: TLB misses ending up as page table accesses per 1000 instructions. No classification.

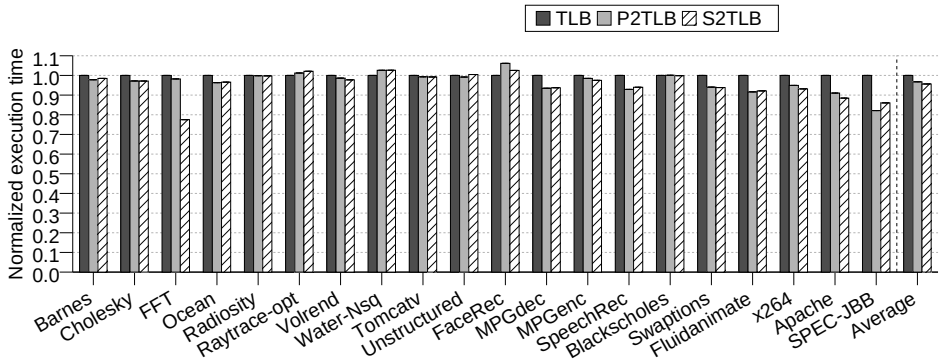


Figure 4.12: Execution time normalized to baseline without classification support.

fast TLB-to-TLB transfer miss resolution mechanism is implemented, TLB misses are only resolved by accessing the page table if no other TLB is currently holding the translation. As can be expected, including a second level TLB (*P2TLB* or *S2TLB*) reduces the average number of accesses to the page table, compared to a single-level TLB structure (*TLB*), even with fast TLB miss resolution through broadcast TLB transfers. However, in some cases, *S2TLB* can be observed effectively exploiting the size of the L2 TLB over the private TLBs approach. For instance, SpeechRec or Apache, among others, reduce the number of accesses to the page table to a greater extent (up to 91% less TLB misses with Apache), showing how a shared TLB configuration extraordinarily helps preventing redundant page translation copies. Finally, the number of TLB-MPKI is remarkably low, less than 0.07 misses on average using a two-level TLB structure. Differently, using a single-level TLB retains TLB-MPKI slightly over 0.15 misses on average, despite implementing TLB-to-TLB transfers.

Accessing the page table in the main memory after a TLB miss implies performing an expensive page walk operation. Therefore, when the page table access is avoided, execution time is improved. Figure 4.12 shows how *S2TLB* slightly improves execution time by only 1% on average compared to *P2TLB* with the same total size, and up to 4.4% compared to the single-level *TLB* scheme. Despite the fact that the shared L2 TLB reduces the number of accesses to the page table, the improvement is comparatively low as a result of the small absolute amount of TLB misses reported in Figure 4.11. In the case of SPEC-JBB or Apache, where the MPKI reported is greater, the improvement over the *TLB* scheme is more noticeable. Differently, in some benchmarks, as Barnes or SPEC-JBB, a private L2 TLB slightly

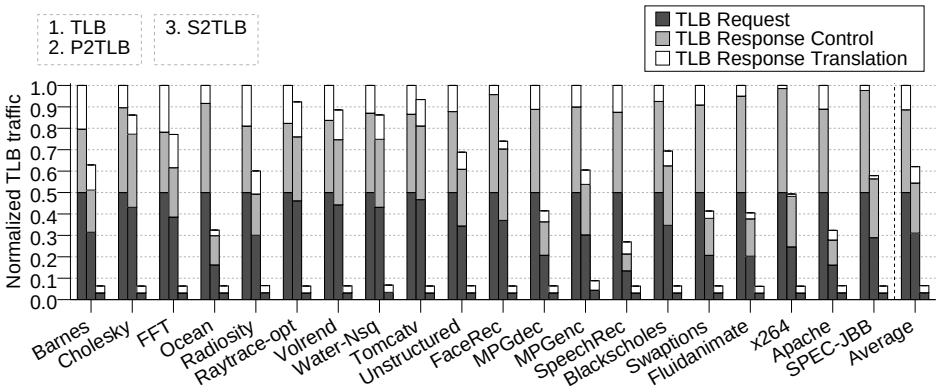


Figure 4.13: Normalized traffic attributable to the TLB communication.

outperforms a distributed shared L2 TLB scheme. The performance shrinkage occurs on account of the additional latency of accessing a shared L2 TLB, which needs to traverse the network in order to reach the home TLB slice. Therefore, on benchmarks with high L2 TLB hit ratio or accessing a great amount of private data, a greater access latency may hurt system performance, overmatching the potential benefits of inter-core sharing patterns exploitation. On the contrary, as FFT has more accesses to shared pages, *S2TLB* execution time improves.

Finally, TLB-to-TLB transfers significantly increase TLB traffic, since every TLB miss induces a broadcast message and many responses, potentially including many replicated translations. Figure 4.13 shows all network flits (the flow control unit in which network packets are divided — see Table 3.1) transmitted across the network, normalized to *TLB*. Essentially, *P2TLB* reduces the TLB traffic by 48.0% compared to *TLB*, as it first relies in the L2 TLB to resolve L1 TLB misses. Therefore, the broadcast TLB miss resolution mechanism is only invoked after an L2 TLB miss. In contrast, L1 TLB misses with a shared L2 TLB are resolved through unicast messages. As a consequence, while a shared TLB structure offers similar benefits to those of a system with TLB transfers, TLB network traffic with *S2TLB* is greatly reduced to barely 6.5% compared to *TLB*, reducing network consumption and benefiting system scalability.

4.5.2 TLB-based Classification

So far, different TLB hierarchies have been analyzed, without any classification scheme whatsoever. This section analyzes how temporal-aware TLB-based classification techniques work assuming systems with two-level TLBs, and evaluating their potential when applied to coherence deactivation for private data. First, using Snooping TLB-based classification for private TLB structures; then, the alternate TLB-based scheme proposed for distributed shared L2 TLBs.

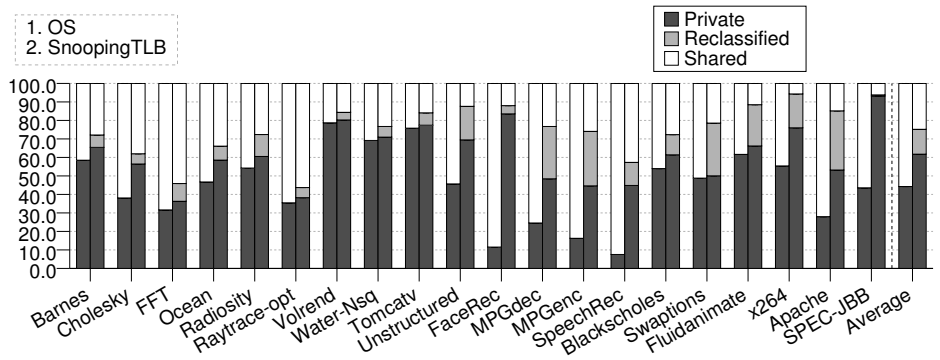


Figure 4.14: Private/Shared page classification with private TLBs.

Snooping TLB-based Approach

This section analyzes how the TLB-based classification scheme for multilevel TLB structures explained in Section 4.3.5 behaves (i.e., *P2TLB* in prior studies). The benefits of the classification scheme are tested by deactivating coherence maintenance.

Classification accuracy. A good, first, general metric to determine the effectiveness of TLB-based classification is the amount of private data detected, provided they do not allow *false private* classification (i.e., when the page is classified as private it cannot be simultaneously stored in more than one private cache). Figure 4.14 shows the amount of pages classified as private or shared by different classification mechanisms. *OS* is a non-adaptive OS-based classification mechanism. *SnoopingTLB* is our proposed TLB-based classification approach for multilevel private TLB structures. The characterization is extended to discern the amount of shared pages that are reclassified to private at least once (*Reclassified* –see Section 3.4). By differentiating reclassified data we offer more insight into the potential benefits of a temporal-aware classification. Note that for a page to be considered private in the figure, it must remain so for the entire execution time.

Particularly, *OS* only classifies 44.3% page as private on average, and once they become shared they remain on that classification status. Differently, *SnoopingTLB* classifies 61.8% of all accessed pages as *Private* (17.5% more than *OS*), and 13.4% of all shared pages are *Reclassified* to private at least once, proving the benefits obtained from an adaptive classification scheme. It is important to notice that thread migration is rare in the evaluated applications due to its short execution time. Furthermore, as far as the classification depends on the presence of a page translation in the system TLBs, the accuracy of *SnoopingTLB* is dependent on the size and associativity of the TLB structure.

Case Study: Coherence Deactivation. This section evaluates the impact of different classification approaches when applied to a classification-based optimization such as the coherence deactivation scheme. Baseline system for this study does not deactivate coherence nor use TLB-to-TLB transfers.

Coherence deactivation main interest is to alleviate the directory storage requirements. The more private data detected the lesser directory entries would be required. Figure 4.15 shows the average number of directory entries used per cycle normalized to the baseline system (*Base*). The OS-based classification (*OS*) avoids the storage of 27.7% entries in the directory cache. Additionally, our TLB-based classification approach for private TLB structures (*Snoop-*

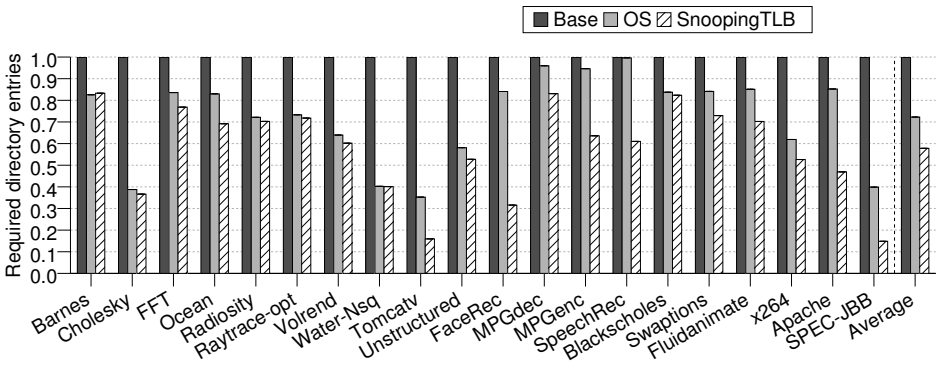


Figure 4.15: Average directory entries required per cycle normalized to baseline.

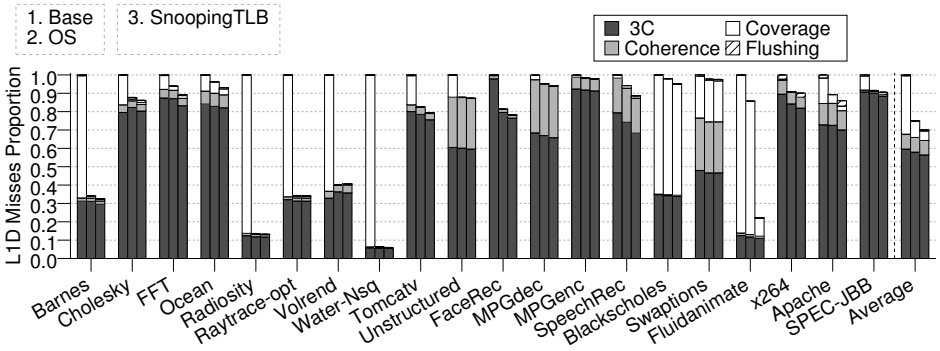


Figure 4.16: Normalized data L1 misses classified by its cause.

ingTLB) avoids 42.2% of the directory entries, approximately halving the directory storage requirements.

Reducing directory storage requirements reduces the misses induced in the private cache due to directory replacements. Figure 4.16 shows data misses classified by the cause of the miss. *3C* misses correspond to the classical compulsory, capacity and conflict misses; *Coherence* corresponds to misses produced by the coherence protocol; *Coverage* misses correspond to those misses induced by the inclusion policy between the directory cache and the private cache due to directory replacements; finally, *Flushing* misses are those originated due to the block invalidations originated by the classification mechanism, either due to recoveries on sharing status transitions or due to the TLB-cache inclusion policy on TLB-based approaches. As can be seen, nearly 1 out of 3 data cache misses with *Base* are misses produced due to the directory size constraints. Conversely, *OS* is very effective on reducing *Coverage* misses when applied to the coherence deactivation mechanism. Specifically, *OS* avoids on average 72.2% of all *Coverage* misses. In other words, *Coverage* misses represent only 11.8% of all cache misses with *OS* and coherence deactivation. Finally, *SnoopingTLB* further reduces *Coverage* misses to just 7.4% of all data cache misses. Furthermore, as expected, *Flushing* misses with *SnoopingTLB* represent less than 1% of all misses on average, since for a TLB to be evicted it should have not been accessed for a while. Therefore, most blocks have been probably evicted already and are not going to be accessed again soon.

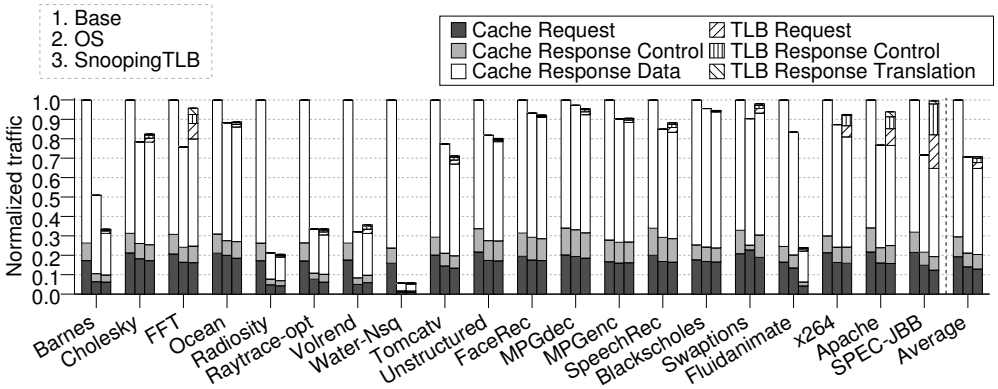


Figure 4.17: Network flits injected, classified into cache or TLB-traffic.

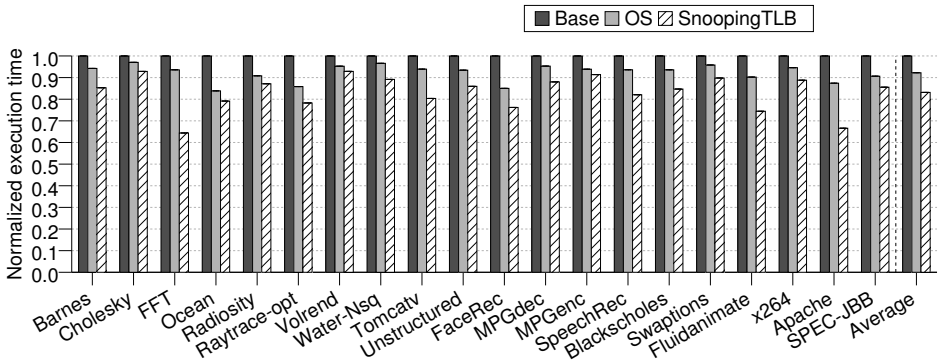


Figure 4.18: Execution time normalized to baseline when applied to coherence deactivation.

Furthermore, reducing cache misses leads to diminish network usage. Figure 4.17 depicts the total flits injected into the network, differentiating flits originated by the cache (i.e., coherence traffic) or the TLB (i.e., classification traffic) structures. *OS* classification reduces the cache traffic by 29.3%, on average, when deactivating coherence for private data. Similarly, *SnoopingTLB* reduces the total network flits injected by 29.2% on average. However, *SnoopingTLB* introduces traffic overhead due to TLB-to-TLB messages and TLB responses including the translation entry. However, TLB traffic represents less than 8.6% of the total network traffic, since TLB miss rate is generally low. As a result, despite the fact that cache traffic is reduced slightly more compared to *OS*, due to a better private data detection, total traffic is similar, on average, for both classification approaches.

Finally, reducing pressure on the directory and the cache has an impact on system performance, as shown in Figure 4.18, which represents the execution time normalized to *Base*. It can be seen as *OS* improves the application's execution time by 7.8% on average. Nonetheless, *SnoopingTLB* improves the execution time by up to 16.8%, on average, compared to baseline, nearly 9% better performance compared to *OS*. This improvement comes from two main causes: (i) the TLB-to-TLB fast miss resolution mechanisms, which avoids consulting the page table on main memory, and (ii) better classification accuracy, specially due to its classification adaptivity. As an illustration, applications with higher amount of shared data (e.g., *FFT*)

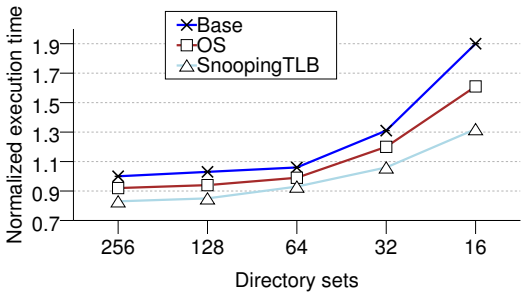
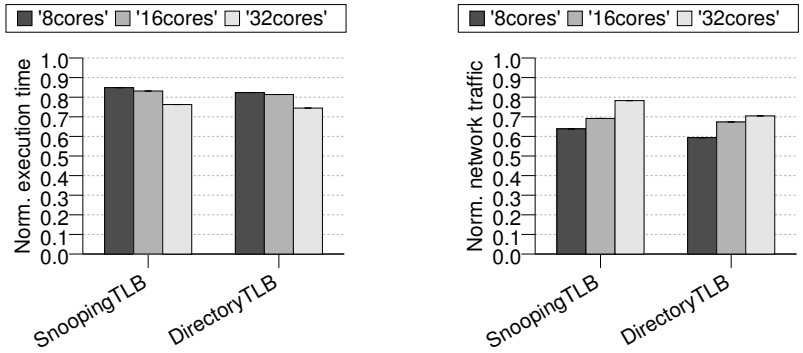


Figure 4.19: Average normalized execution time depending on the directory size.



(a) Normalized runtime. **(b)** Total flits injected.

Figure 4.20: Scalability analysis of TLB-based classification approaches.

perform better with *SnoopingTLB*, as more TLB misses are resolved through network on-chip communication. However, the impact of adaptivity is limited due to the relatively large size of the directory and the absence of thread migration in the simulations. *SnoopingTLB* removes more than 40% of directory entries, and thus the directory cease causing a performance bottleneck.

As a consequence, we may be interested in reducing the directory cache size to make it more scalable and fast. Reducing the directory size hurts execution time for all configurations. However, coherence deactivation might alleviate the performance shrinkage. Figure 4.19 shows the average execution time normalized to the baseline configuration and how it evolves with different directory cache sizes. The smaller the directory is, the faster the system performance drops. Using a 16-set directory, *Base* configuration nearly doubles its execution time compared to a 256-set directory. Conversely, both *OS* and *SnoopingTLB* lessen the performance penalty on smaller directory sizes. Specifically, *SnoopingTLB* reduces the average execution time with a 16-set directory by 30% compared to *Base*.

Scalability study. This section shows how TLB-based classification mechanisms for private (*SnoopingTLB*) and shared (*Directory*) TLB structures scale when deactivating coherence maintenance. Due to the slow pace of the simulation tools, this study is only performed using SPLASH 2 benchmarks and scientific application. All the results in Figure 4.20 are normalized to a system of the same core count, with a purely private TLB structure that does not perform data classification; therefore, coherence maintenance is not deactivated.

Figure 4.20a shows how our adaptive TLB-based classification approach improves execution time for both private and shared TLB structures when applied to coherence deactivation. Specifically, *SnoopingTLB* reduces the execution time by 23.7% on a 32-core CMP. The alternate classification approach for shared last-level TLBs (i.e., *DirectoryTLB*) behaves similarly, reducing the average application's execution time by 25.5% for a 32-core system.

The snooping TLB-based page classification mechanism proposed in this thesis aims for small- or medium-scale systems. As can be seen in Figure 4.20b, broadcast messages issued after every TLB miss do not scale well for larger systems. Particularly, *SnoopingTLB* generates nearly 9% more traffic than *DirectoryTLB* for 32 cores, which would ultimately offset coherence deactivation gains for larger systems. Conversely, leveraging shared TLB structures completely avoids broadcast requests, issuing unicast messages after missing on the first TLB level. As a consequence, *DirectoryTLB* considerably mitigates to a large extent the network traffic increase for a 32-core system.

4.6 Discussion

This section discuses some architectural alternatives and different options, and how they may affect to our TLB-based classification approach.

4.6.1 Large or Multilevel Private Caches

Due to the TLB-L1 inclusion policy, after every TLB eviction, the blocks pertaining to the page are flushed from the private cache. The number of lines visited within the cache is not related to the cache size but to the page size (64 cache lookups for 4KB pages). However, the number of blocks effectively invalidated upon a flush operation may increase with the cache size since more blocks belonging to an evicted page may be stored in a larger private cache, ultimately increasing cache misses. In this scenario, the TLB should scale accordingly to the cache size or use a multi-level TLB approach in order to dramatically reduce the total flush operations performed.

Additionally, it is also applicable to configurations with two or more levels of private caches. In this case, performing the page flushing requires the invalidation of the corresponding page blocks at every private cache in the hierarchy. This action can be performed in parallel. Some techniques, like Direct-to-Data (D2D) cache [76] or a Split Cache Hierarchy [77] can help reducing the amount of cache searches to be performed.

4.6.2 Virtual Caches

Virtually indexed, virtually tagged (VIVT) caches (also known as virtual caches) do not require TLB accesses for cache hits, which can result in faster lookups and less power consumption than the physical caches. Fortunately, the address translation is performed anyway on every cache miss, since coherence is kept for physical addresses. Also, L1 hits do not issue coherence actions. These two characteristics allow our proposal to be completely applicable to virtual caches.

4.6.3 Large Page Sizes

Our proposal can also work in systems implementing large pages (with 2MB being the most common alternative on x86 architectures). However, the eviction of the entries for large pages from the TLB will require a more expensive cache flushing. In order to overcome the latency overhead, diverse approaches could be employed. For example, a simple counter can be added to the TLB entry indicating the number of *live* or cached blocks for the evicted TLB entry. Alternatively, a presence vector tracking large memory regions could be used [88], which would set the region limits to be flushed. Other proposals may also help reducing the latency overhead [76, 77] These approaches aim on reducing the amount of required lookups when flushing.

Alternately, based on the observation that the majority of pages accessed per core on systems with superpage support are the smallest pages [59], we could avoid classifying superpages and consider them as coherent without significantly damaging system performance, while completely avoiding large pages overhead. Current systems supporting multiple page sizes implement multiple TLB structures to do so, thus the TLB actually storing large pages could classify them as shared without extra hardware support.

4.6.4 Design of the Classification Directory

In this chapter we introduce a mechanism that leverages an inclusive, shared, NUCA-like TLB level in order to track classification information, acting similarly to an in-cache directory [64, 29]. For instance, some recent Intel micro-architectures use this cache directory scheme, with an inclusive LLC, where its tags contain core valid bits (CVB) to indicate which core private caches contain copies of the block [56].

While storing classification information is arguably far more area-scalable than any cache directory (where each directory entry has to track which cores are storing each cache line), leveraging a SLL TLB still enforces an inclusive policy between the private TLBs and the shared TLB level. As the classification scheme also entails a TLB-cache inclusion, evicting a shared TLB entry might end up adding significant pressure to the cache hierarchy.

To avoid this, the classification information might be maintained in a dedicated directory structure, thus alleviating the pressure and requisites in the TLB hierarchy. However, a classification directory adds an indirection on TLB misses and evictions, both for private or shared TLB structures, which would increase the complexity of the classification scheme. Moreover, translation information can be replicated in the classification directory to completely avoid broadcast requests on TLB misses for private TLB hierarchies, ultimately resembling a shared last-level TLB in the hierarchy.

4.7 Conclusions

This chapter introduces snooping TLB-based classification, an effective temporal-aware mechanism for private-pages detection aiming on medium-scale CMPs. This mechanism snoops other cores in the system on a TLB miss looking for stored page translation entries. This way, SnoopingTLB classifies pages accessed by several cores at different points in time (e.g., thread migration or program phase changes) as private. This leads to a bidirectional page re-classification, from private to shared, and vice versa, that results in a significantly more accurate classification scheme when compared to an alternative runtime non-adaptive OS-based

classification approach. Furthermore, TLB-based classification allows low-latency TLB miss resolution though TLB-to-TLB transfers.

Finally, we extend TLB-based classification approach by leveraging a distributed shared last-level TLB, namely DirectoryTLB. Therefore, DirectoryTLB allows to naturally discover the sharing access pattern through the L1 TLB misses received from different cores to the home L2 TLB tile. This approach leads to similar classification accuracy to its snooping counterpart, but completely avoids harmful broadcasts on TLB misses. As a result, it represents a far more scalable approach.

Chapter 5

Token-counting TLB-based Classification Mechanism

This chapter introduces TokenTLB, a TLB-based mechanisms that classifies pages based on token counting. The chapter thoroughly analyzes TokenTLB, and compares against other runtime classification techniques, both TLB- and OS-based.

5.1 Introduction

In the previous chapter we have introduced a snooping TLB-based classification approach that performs an adaptive classification that accounts for temporarily private pages and thread migration. Snooping TLB-based classification relies on TLB-to-TLB transfers to inquire other cores' TLBs in the system to naturally discover whether blocks belonging to a page may be currently stored on a remote cache and therefore the page is shared, or, on the contrary, the page is currently private. However, TLB-to-TLB transfers generate a large number of responses, possibly including the translation, from every core in the system after every TLB miss. Furthermore, frequent broadcast requests and responses are not supposedly scalable to large-scale systems, as the number of messages increase proportionally with the system size.

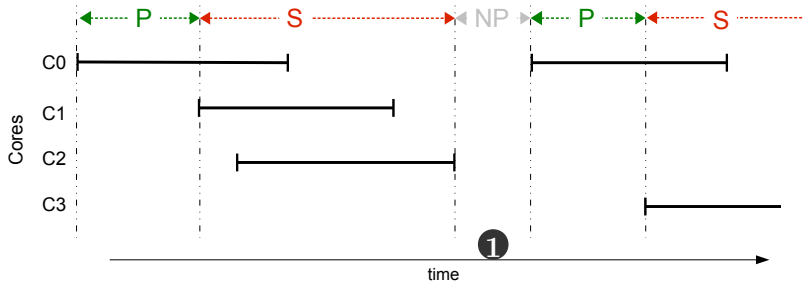
Additionally, a distributed shared TLB structure has been explored in order to alleviate network pressure. However, a shared TLB structure is not a common architectural design in commodity processors. In addition, page reclassification to private requires the translation to be completely removed from all TLBs in the system to occur, both for private and shared TLB structure classification schemes, thus limiting the accuracy of the classification mechanism. Ideally, reclassification should be performed when only one TLB is currently holding the translation. Finally, write detection (i.e., discerning between read-only and read-write pages) has not been explored, since it requires an expensive synchronization operation among TLBs in the system, and would entail increasing network consumption, specially in a snooping TLB-based classification scheme. Therefore, so far TLB-based classification scheme is limited to the private-shared dichotomy.

In what follows, we introduce TokenTLB, a token-counting TLB-based classification mechanism. The main contributions of TokenTLB are:

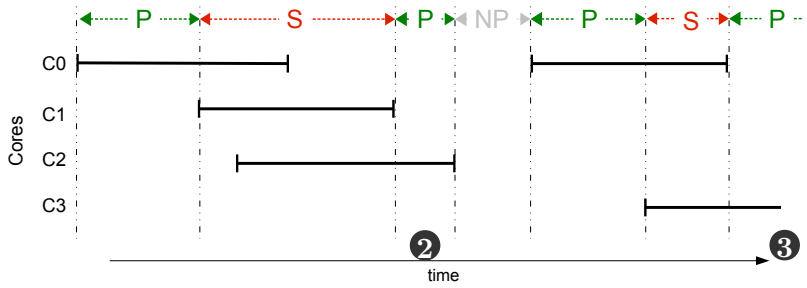
- TokenTLB reduces network consumption compared to snooping TLB-based classification, while allowing shareability among TLBs. Only a subset of TLBs provide tokens along with the page translation, which leads to about one response per TLB miss on average.
- TokenTLB extends classification characterization by detecting read-only pages, while performing an adaptive classification based on count and exchange of tokens.
- Token-based classification makes it possible to naturally and immediately identify a shared TLB page entry transitioning to private.
- TokenTLB introduces a predictor capable of resolving TLB misses through unicast messages, thus increasing scalability. The predictor relies on a small buffer called Token Predictor Buffer (TPB), which is in charge of short-time storage of potential token-holding TLBs.

5.2 TokenTLB

This section describes TokenTLB, a token-counting classification scheme that tries to reach all desirable properties for a classification mechanism as seen in Chapter 2: perform the classification prior to accessing the cache hierarchy; implement a fully-adaptive classification able to carry out an accurate reclassification; improving classification characterization by discerning write accesses and recognizing read-only pages; and performing an accurate run-time classification valid for any code.



(a) Adaptive TLB-based classification.



(b) Fully-adaptive token-counting classification.

Figure 5.1: Adaptivity versus Full-adaptivity in TLB-based classification.

Thus, TokenTLB introduces the concept of full-adaptivity, as seen in Figure 5.1. Snooping TLB-based classification performs an adaptive classification as it allows a shared page to tran-

sition to private again, on temporarily private application phases (Figure 5.1a). However, for a reclassification to occur, a page must loose its sharing information (i.e., the page trnaslation must be evicted from all system TLBs ❶), and then it will be classified as private if brought back again. Full-adaptivity (Figure 5.1b) allows TokenTLB to dynamically transition a page classification to private while its translation remains stored in a TLB (e.g., ❷ and ❸), since tokens may be recovered after other page translation entries are evicted from their TLBs.

5.2.1 Introduction to Token-counting in TLBs

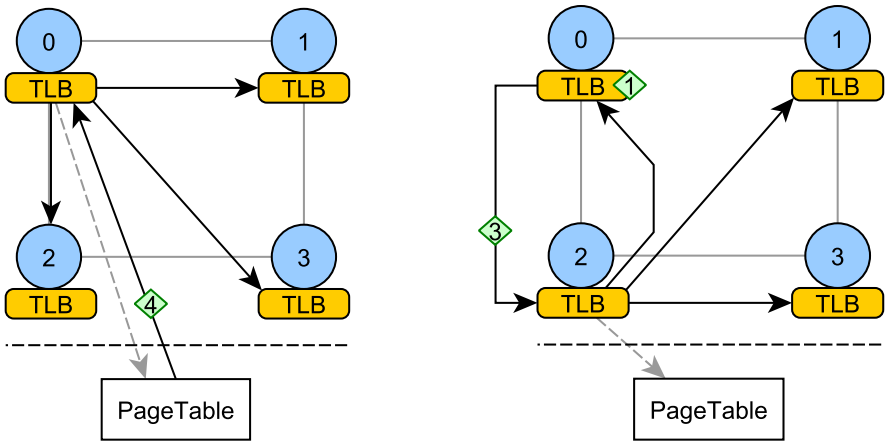
TokenTLB associates a fixed number of tokens with each translation entry. In a system with N cores, there must be N tokens per entry. New tokens cannot be generated, and tokens cannot be destroyed. Tokens are exchanged through TLB-to-TLB messages alongside the page address translation. A TLB page entry is classified according to its token count: private if it holds all tokens (N), shared while holding a subset of all page's tokens (from 1 to $N-1$), and invalid when holding no tokens. Finally, only TLB entries holding at least two tokens may reply to a translation request from the network.

Additionally, in order to track whether the page is read-only or not, there is a *written* flag (W) associated with each translation entry, which is sent alongside the tokens on TLB transactions. The *written* flag increases the classification scope by adding extra classification categories:

- Private Read-only (PR) page: Only one processor is currently accessing the page blocks. All accesses has been loads during the page lifetime. It is, N tokens and $\neg W$.
- Private read-Write (PW) page: Only one processor is currently accessing the page blocks. It has been written at least once during current page lifetime. It is, N tokens and W.
- Shared Read-only (SR) page: At least two processors are currently accessing the page blocks. All accesses has been loads during the page lifetime. It is, (tokens > 0 & tokens $< N$) and $\neg W$.
- Shared read-Write (SW) page: At least two processors are currently accessing the page blocks. It has been written at least once during current page lifetime. It is, (tokens > 0 & tokens $< N$) and W.

5.2.2 Token Request upon TLB Miss

TokenTLB initiates the page table walk process after a TLB miss in parallel with a broadcast request snooping other cores' TLBs, as seen in Figure 5.2. Initially, the page table holds all N tokens for each page translation. Consequently, after the first TLB miss for a memory page, the page table delivers all the tokens to the requestor TLB (Figure 5.2a). From now on, tokens are held by TLBs and sent through messages on response to TLB miss requests, spreading across the core' TLBs. When a TLB receives a TLB translation request, it checks if it *owns* the translation entry (i.e., holds the page translation with two or more tokens in it) and if so, it answers the request with a short response message, keeping one token and sending the rest (Figure 5.2b). When the first translation response with tokens is received by the requesting TLB, the page table walk is canceled, tokens are annotated privately in the corresponding page TLB entry, and the memory access proceeds. By doing this, response traffic is constrained as only one TLB (usually the most recent in acquiring the translation) is allowed to answer in the common case. Furthermore, page access is unlocked sooner compared to snooping TLB-based



(a) Token request retrieving tokens from the page table.

(b) Token request exchanging tokens from the token holder.

Figure 5.2: Token Request example in a 4-core CMP.

approaches, where a requestor TLB has to wait for all other core's TLBs to answer in order to access the page.

Tokens are stored in an additional TLB field, namely *Tokens*, as seen in Figure 5.3. This field adds $log_2(N)$ bits to each entry (e.g., only 4 extra bits for a 16-core CMP) in the TLB. When all tokens are given away we rely on the valid/invalid bit (V) of the TLB entry to track it. In the case of the page table, it does not require dedicated hardware, but just one extra bit per entry (whether it has or has not all tokens), namely T, that can be one of the reserved bits in the page table entry. Compared to an OS-based approach with write detection [23], which requires $3+log_2(N)$ bits, our solution represents far lesser overhead, and a more scalable approach.

TLB misses allocate an entry in the Miss Status Holding Register (MSHR), which is deallocated only after acquiring both the page translation entry and at least one token for that page. Therefore, if the page walk process ends without delivering tokens with the page table translation (tokens are held by other TLBs), we have to wait for the first token response. Page access cannot be unlocked without acquiring sharing information (i.e., tokens). Note how, in some cases, more than one TLB may respond to the TLB miss request, since token exchange distributes tokens along the page translations stored in the system TLB (e.g., sending tokens

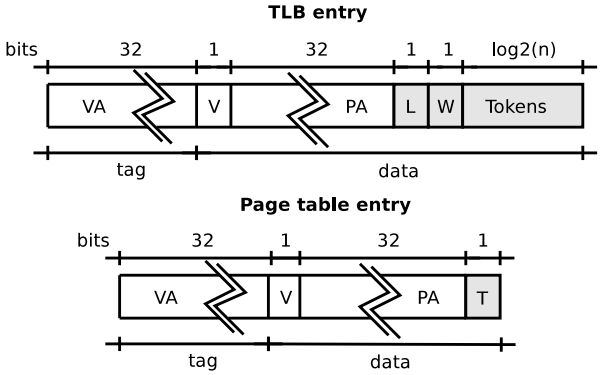


Figure 5.3: TLB and page table entry format. Shaded fields represent additional the fields required by TokenTLB.

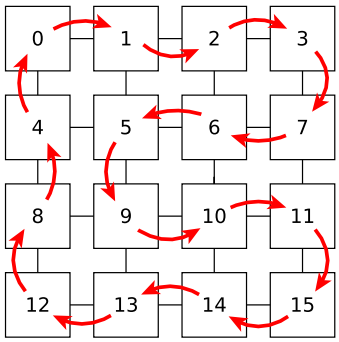


Figure 5.4: Path that token messages follow after TLB evictions for a 16-cores (4x4) mesh interconnect.

to the network after TLB eviction). Consequently, a page access can be temporarily classified as shared although it may be effectively private, as some tokens may be still in-flight. However, once a TLB obtains all N tokens, the page naturally becomes private.

Additionally, when a page is written for the first time, i.e., the W bit is set, this bit needs to be set in all copies of the translation stored in other TLBs. This bit remains set until the *global page generation time* (elapsed time from a page is first cached on a TLB to the moment it is evicted from the last TLB in the system) for that page ends. When the write happens in a private page, no actions are required. Otherwise, a message is broadcast to update the W bit in all TLBs holding tokens for that page, which produces a transition to shared-written (SW). Written information is sent alongside tokens as part of the page *sharing* information on TLB miss responses.

5.2.3 Token Release for Correctness

Tokens are neither created nor destroyed, but transferred. This means that the system must always guarantee the existence of N tokens for any given page translation. The page table either holds all N tokens or none. Otherwise, accessing the page table looking for tokens could become a costly frequent operation, which should be avoided.

When a TLB evicts a page entry that is holding a subset of tokens, a message is sent looking for a new holder (i.e., another TLB) for those tokens. On the contrary, if a TLB entry is holding all N tokens, they are sent back to the page table. Consequently, TLB evictions are required to be non-silent, adding some extra traffic consumption. However, non-silent evictions do not hurt system performance, as they are out of the critical path for memory accesses. Contrary to that, non-silent evictions grant a more dynamic classification, being fundamental for full-adaptivity in TokenTLB.

As previously noted, a subset of tokens on an evicting TLB entry must be transferred to a new holder. To do so, a message (*token_evict*) that only carries tokens, but not the translation, is sent to another TLB. Ideally, a TLB receiving a *token_evict* request should only accept tokens if it is already holding a valid TLB entry for that page or if it has a MSHR entry allocated as a consequence of a TLB miss for that page (Section 5.2.2). Otherwise, the message is sent to the next designated TLB. When the *token_evict* request finds a new holder, tokens are annotated in the new TLB, which responds with an acknowledgment to the original sender.

In order to avoid possible potential livelocks and to minimize traffic, the search of the new holder requires a neat exploration of the network. To this end, we add the concept of a

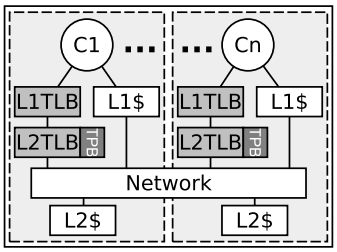


Figure 5.5: Logical TPB placement with a private two-level TLB organization.

logical network ring, which is a path that covers all nodes in the system. The path followed is dependent on the network topology. Figure 5.4 illustrates an example path for a 4x4 mesh interconnect. This path represents one of the minimum circular routes traversing all nodes in the interconnect.

However, livelocks can still occur when all the TLBs holding a page (and all of its tokens) try to simultaneously evict the associated entry. To solve this problem, tokens being carried on a *token_evict* message can be stored in the corresponding MSHR of a core whose TLB is involved in an eviction process, provided that its ID is greater than the requestor core ID. Token reception is acknowledged afterwards.

Note that the key condition for starvation avoidance in a scenario where multiple evicting TLBs for the same page endlessly pass on their tokens is the core ID comparison. As a result, if all TLBs simultaneously evict the same page, all N tokens will eventually end up in the TLB of the core with the greater ID, which will send all tokens back to the page table.

In short, now a TLB can simultaneously evict tokens while temporarily stores other tokens for that page. Thus, when that TLB receives an acknowledgment indicating that the evicting tokens were acquired by another TLB, it has to check its own MSHR again. If it is holding a subset of tokens for that page, it sends a new request message looking for another holder. In case the MSHR is not holding any tokens, the eviction process simply ends.

Finally, observe how a TLB-to-TLB request may fail to acquire tokens after a miss if all TLBs *owning* the translation entry have evicted their tokens short before receiving the request, and thus tokens are currently in-flight (entries with a single token are not allowed to answer). This is prevented by setting a timeout after a TLB miss, which resends the broadcast request if the page has failed to acquire any token at the expiring time. This is not a frequent event and it hardly increases network traffic. Also, as *token_evict* messages only take tokens and not the translation, the TLB miss is not resolved when receiving a *token_evict*. The missing TLB annotates the tokens in the corresponding MSHR entry and acknowledges its reception, but must remain waiting for a TLB response with the translation or the conclusion of the page table walk process to finish.

5.2.4 Token Predictor Buffer

In the search for a more scalable classification approach, TokenTLB reduces response TLB traffic and translation replication. However, a broadcast request is still sent after every translation miss. TLBs do not have previous information of possible token *owners* (TLBs holding two or more tokens for a page translation) at the time of the miss, therefore TLB misses still need to be resolved by flooding the network. However, some TLB misses occur shortly after its invalidation, and thus a potential token holder could be anticipated. Consequently, TLB

traffic would be further reduced, contributing to a more scalable classification approach. To this end, we introduce a predictor in charge of revealing other TLBs as potential token *owners*. Hitting on the predictor after the TLB miss issues an unicast request, thereby reducing TLB request traffic. This prediction based on previous recent history is similar to the one proposed by Martin *et al.* [52]. Other works also benefit from this observation with different aims [7, 65, 66].

The predictor consists of a small data buffer situated in parallel with the L2 TLB (Figure 5.5) called Token Predictor Buffer (TPB), storing the process ID, virtual address, and core ID. After giving tokens away with non-silent evictions, the receiver becomes a known potential *owner* of tokens and is therefore stored into the TPB. If an L1 TLB miss occurs, both the L2 TLB and the TPB are checked in parallel. If the L2 TLB misses and the TPB keeps information of a potential token holder for that page, an unicast request is sent and the TPB entry is deallocated. If the TLB receiving the request is still holding tokens, it positively responds with the translation and classification information. Otherwise, if the consulted TLB is not a token holder anymore, it negatively answers the TLB request and the conventional token broadcast TLB miss resolution mechanism is invoked in parallel with the page walk.

5.3 Read-only Data Optimizations and Full-adaptivity

When applying TokenTLB to a classification-based data optimization that supports read-only data, only blocks belonging to shared written (SW) pages might require special actions.

Any data optimization applied to a full-adaptive classification scheme must take special care in the implications and penalty of page reclassification. Note that even under a coherent state (SW), classification could transition again to private (PW) during the same *local page generation time*, provided that a TLB recovers all tokens.

Specifically, when a reclassification from private or shared-read-only to shared-written occurs in a core's TLB, all the blocks of that page must be evicted (see Section 4.3.4 for more details on specific actions) from all core's private cache (flushing-based recovery). To this end, a message is broadcast, and a page flush is performed through all system private caches for TLBs holding a valid page entry whenever a transition to shared-written occurs. This process must be atomic to avoid inconsistencies (accesses to that page are stalled until the recovery process is finished in all TLBs).

As with snooping TLB-based classification, page flushing occurs whenever a TLB entry is evicted from the private last-level TLB. The presence or absence of a TLB entry must imply the presence or absence of cache blocks for that page in the upper level cache structure in order to accurately classify data and avoid false privates based in the presence of a translation entry in the TLB structure. This will hardly affect performance since for a TLB to be evicted, it must not have been accessed for a while, and it is likely that blocks are neither present in the cache structure nor will be accessed in the near future (again, given that TLBs are not prematurely evicted –Chapters 6 and 7).

Finally, when a classification is demoted (i.e., from SW to PW) no actions are required. However, different data optimizations may require special actions. For instance, when employing Coherence Deactivation (see Section 2.2.3), we may potentially have accessed blocks of a page as coherent, and allocated the corresponding block entry in the directory cache. When evicting the directory entry due to a conflict, if it produces an invalidation in a currently private cache entry, an unnecessary cache miss may occur afterwards. Notice how the status for a

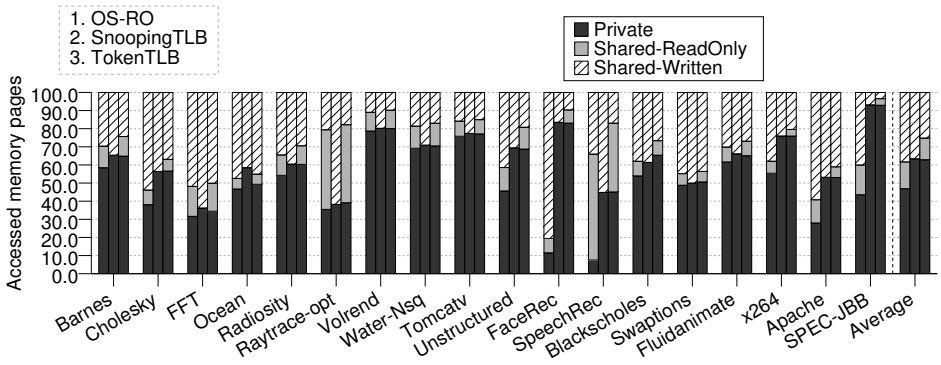


Figure 5.6: Private, Shared, and Written page proportion.

non-coherent block would be naturally restored by the recovery mechanism when transitioning to coherent again (after being accessed from another TLB). To prevent this to happen, if a block is found as non-coherent when evicting a directory entry (which sends an invalidation request to the cache pointed by it), an acknowledgment is sent to the directory but the block is allowed to remain in the cache as non-coherent.

5.4 Experimental Results

In order to check how TokenTLB classification behaves compared to previous classification approaches, we perform an in-depth study including the analysis of the Token Predictor Buffer. Baseline system has a private two-level TLB structure with the configuration described in Chapter 3. Coherence deactivation is again the case study in order to test the benefits of TokenTLB against previous classification mechanisms. Finally, a scalability study shows how the different classification schemes behave when increasing the core count.

5.4.1 Full-Adaptive Page Classification

This section demonstrates the classification accuracy and efficiency of TokenTLB compared to previous proposals.

Private and read-only data. The percentage of private and shared (read-only/written) pages is a good general metric for measuring the goodness of a non-adaptive classification approach. Figure 5.6 shows how pages are classified as Private, Shared-ReadOnly or Shared-Written by different classification mechanisms. *OS-RO* is a non-adaptive OS-based classification mechanism with Read-only detection. *SnoopingTLB* is the adaptive broadcast TLB-based classification approach introduced Chapter 4, and *TokenTLB* is the fully-adaptive token-based TLB classification approach, both with private two-level TLB structures. As *SnoopingTLB* is not able to distinguish shared-read-only or shared-written pages, all shared pages fall under the same classification category. However, for the sake of clarity, in the graph it appears as *Shared-Written* in all *SnoopingTLB* configurations. We observe as, averagely, the sum of private and shared-read-only pages for *OS-RO* does not suffice to outmatch private pages for *SnoopingTLB*, which represents a 63.4% of all accessed pages, proving the relevance of an adaptive approach. However, in some cases, as *Raytrace-opt* or *SpeechRec*, a lot of potential

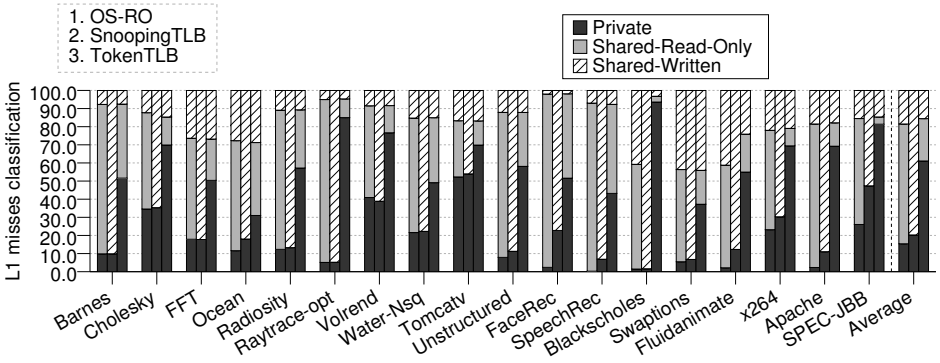


Figure 5.7: Data L1 Misses proportion classified as Private, Shared-Read-Only and Shared-Written.

classification precision is lost when write detection is not performed, as *OS-RO* overpasses *SnoopingTLB*.

However, this metric is unfair for adaptive classification mechanisms, where shared pages are frequently reclassified as private, since reclassification is not reflected in the figure. Specifically, this situation is shown at a greater extent in the case of TokenTLB since it unlocks page access after the first TLB response, which accelerates TLB miss resolution, but with ongoing evictions it might end up in a shared access while tokens are still in-flight. Therefore, in the figure it appears as shared while it is naturally reclassified as private short after its first access. However, computing both private and shared-read-only pages for *TokenTLB*, it improves page classification to 74.9%.

The aim of all classification mechanisms is to precisely classify memory accesses. While an adaptive mechanism may be able to freely reclassify pages, blocks are never individually reclassified during a local block generation time. Therefore, the more the page classification is kept as private or read-only, the more L1 cache misses will end up being treated as such. Figure 5.7 shows L1 data cache misses classification, which will determine how accesses to data blocks will be treated. Even though classification of private pages in Figure 5.6 was close between *SnoopingTLB* and *TokenTLB*, L1 data misses considered as private is greatly increased using *TokenTLB*, since, unlike *SnoopingTLB*, page reclassification occurs in a natural way during a page generation time. Specifically, *TokenTLB* is able to classify 61.1% of L1 data cache misses as Private on average, 40.8% more than *SnoopingTLB*. Also, note as, contrary to *SnoopingTLB*, *TokenTLB* and *OS-RO* are capable of recognize read-only pages, representing the 24.4% of L1 cache misses for *TokenTLB*, thus greatly enhancing the classification accuracy. Note that read-written cache misses represent less than 20% of all misses both for *OS-RO* and *TokenTLB*, with less than 3.4% difference between them both. When accounting for read-only accesses, classification is similar and it might seem as they perform similarly. However, *TokenTLB* amply surpass *OS-RO* in the amount of cache misses as private. Some data optimization are not suited for treating read-only data, therefore detecting private data to a greater extent is a desirable characteristic. Furthermore, while pages, and thus cache blocks remain in the same classification status once they are classified as *Shared-Written*, *TokenTLB* is fully-adaptive, and can reclassify a coherent block back to non-coherent during the page lifetime.

Token TLB-to-TLB exchange. One key benefit of TokenTLB classification over previous proposals is how it handles TLB-to-TLB transfers, obtaining their benefits (page classification, usage prediction allowance, and translation acceleration), while limiting the required responses.

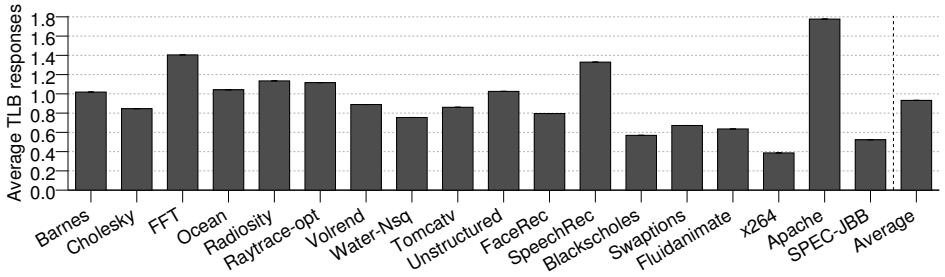


Figure 5.8: Proportion of TLB Responses issued after an L2 TLB miss.

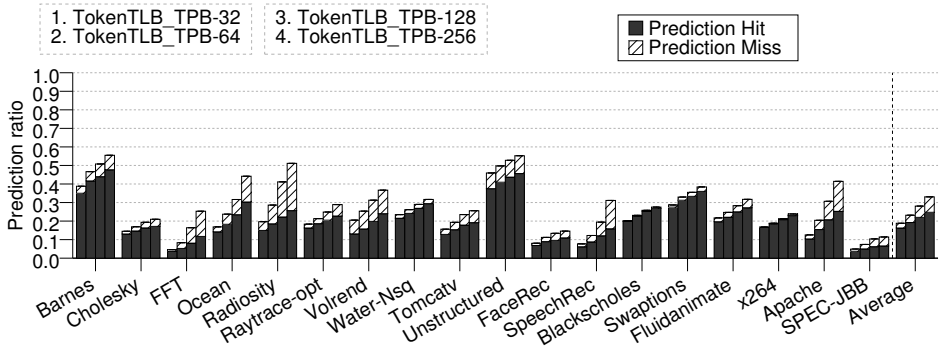


Figure 5.9: Success rate for TPB predictions.

As a result, TLB traffic is reduced, whereas system blockage waiting for collecting answers is avoided. Figure 5.8 represents how many average responses are sent after a TLB miss using *TokenTLB*. Take into account that broadcast TLB transfers for *SnoopingTLB* mechanisms require invariably $N-1$ (being N the number of cores in the CMP) responses after every TLB miss. On the contrary, *TokenTLB* requires just 0.93 responses per L2 TLB miss on average. The average falls below one due to the fact that, using TokenTLB, no TLB responses are sent nor expected when the tokens are held in the page table. In some cases, as *Apache* or *SpeechRec*, it goes beyond one response per L2 TLB miss. Note that TokenTLB allows more than one translation *owner* in the TLB structure, as evicted tokens may be accepted by the first valid TLB in the eviction ring, therefore in those cases two or more responses may be issued after the next TLB request.

Token prediction effectiveness. Hereby we briefly analyze the Token Predictor Buffer (TPB), which is conceived to avoid recurring to broadcast on TLB misses when there may be a known potential token owner. Therefore, it has an impact in the traffic and network consumption, but not in the execution time. TPB itself is just a small 4-way associative short-term buffer memory. Up to four different sizes have been evaluated in the study (32, 64, 128, and 256 entries).

Figure 5.9 shows the proportion of successful and failed predictions (i.e., number of remote TLBs currently holding tokens or not after being requested through an unicast prediction) with respect to total L2 TLB misses. It demonstrates that increasing TPB size affects positively to the accuracy of its predictions. Specifically, a 256-entry TPB avoids the broadcasts (prediction hit) by 24.7% out of a total of nearly 33% of TLB miss prediction tryouts on average. In some

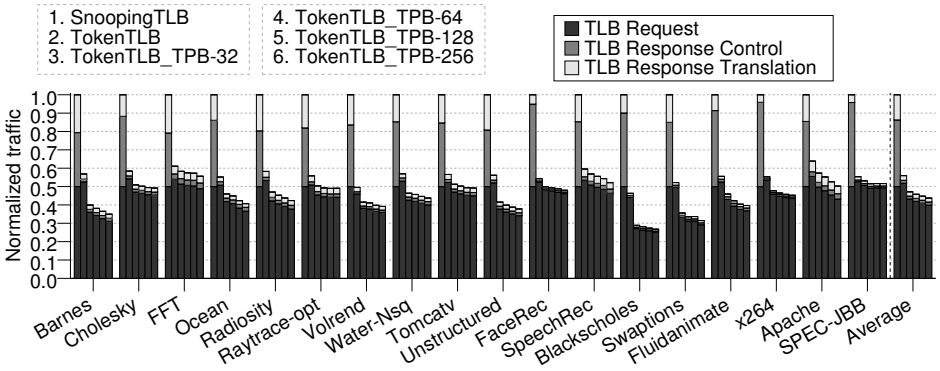


Figure 5.10: Relative TLB network traffic issued.

cases, as *Barnes* or *Unstructured*, around 45% of broadcast TLB misses are prevented by using the TPB.

Figure 5.10 details the TLB traffic compared to base *SnoopingTLB*. It can be observed as *TokenTLB* slightly increases TLB request traffic compared to *SnoopingTLB* due to the non-silent evictions performed. However it is greatly offset with the reduction in TLB response and translation traffic, as can be deduced from Figure 5.8, reducing overall TLB traffic by 44%. Additionally, TPB usage further decreases TLB traffic. Specifically, the greater TPB considered (256 entries) reduces TLB request traffic to a greater extent, as it entails making more predictions, reducing solely the TLB request traffic by nearly 20% regarding *TokenTLB*, and total TLB traffic up to 56.3%.

As a conclusion, *TokenTLB* detects written condition at page granularity, while achieving a private detection similar to previous adaptive mechanisms, seemingly losing accuracy. However, the strength of *TokenTLB* is the benefit obtained from immediate shared-to-private reclassification (full adaptivity), dramatically increasing the number of cache blocks misses classified as Private over any previous classification approach. Finally, TLB response and translation messages are greatly bounded using our approach. TPB inclusion reduces the total TLB traffic by more than half using TPB, thus increasing the scalability of the system.

5.4.2 Case Study: Coherence Deactivation

Coherence deactivation mechanism greatly benefits from the accuracy of a classification approach. It can easily benefit from both a private/shared classification dichotomy and a private/read-only/shared-written classification scheme. This section shows the benefits and overheads of applying TokenTLB to coherence deactivation compared to previous classification approaches. TPB is not included as a basis for the results shown in this section.

Coherence maintenance is deactivated when a block access is considered non-coherent by the classification mechanism. For *SnoopingTLB*, which only characterizes memory accesses into private/shared scheme, all shared pages are coherent. Differently, both *OS-RO* and *TokenTLB* also distinguish written pages, so only shared-written pages are considered coherent. Moreover, *SnoopingTLB* and *TokenTLB* are adaptive mechanisms, transitioning back and forth from private to shared. Figure 5.11 shows the average number of directory entries required per cycle for the different approaches studied, normalized to *Base*, which is a system with the same config-

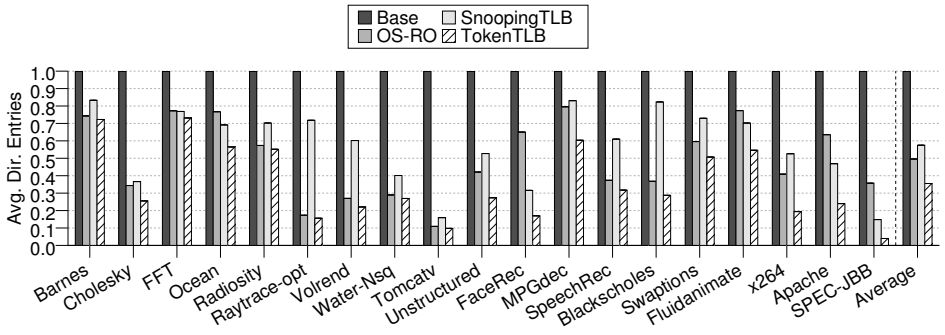


Figure 5.11: Average directory entries allocated per cycle

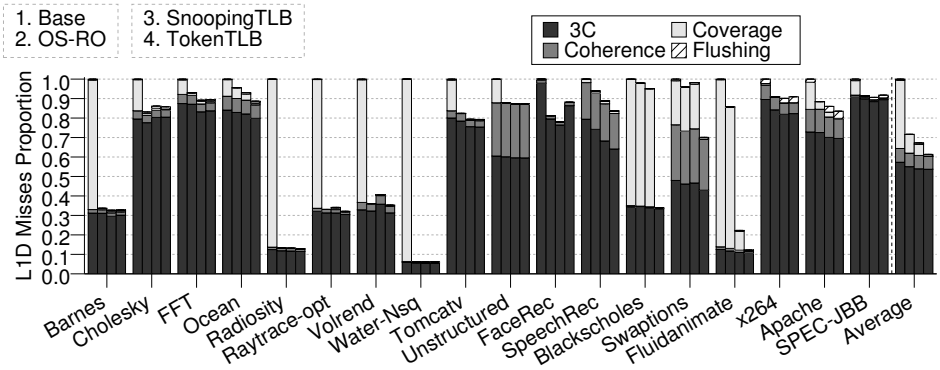


Figure 5.12: Data L1 Misses classified by its cause.

uration but without coherence deactivation. Constraining the directory usage is the ultimate goal for coherence deactivation and is strongly dependent on the accuracy of the classification mechanism. We observe as *OS-RO* improves directory usage by 52.8%, even reducing it over *SnoopingTLB* due to its read-only detection capability. Finally, *TokenTLB* greatly improves directory usage, requiring only 34.1% of the directory entries per cycle compared to *Base*.

Classifying more pages as non-coherent and improving directory usage also reduces L1 cache misses to a greater extent, especially Coverage misses (see Section 3.4). Figure 5.12 classifies data misses based on its cause, and thus, shows how all proposals lessen Coverage misses on the L1 due to apply coherence deactivation. However, unlike previous approaches, *TokenTLB* actually eliminates nearly all coverage misses. On the contrary, on average, coverage misses with *OS-RO* still represent 13.4% of the L1 cache misses. Finally, *SnoopingTLB* is somewhere in between, as it still suffers 8.7% of Coverage misses.

Reducing L1 data cache misses leads to a network usage reduction. Figure 5.13 shows the total flits injected into the network, classified into cache or TLB traffic. It can be observed as *TokenTLB* prevents nearly half the total network usage on average due to both the fully-adaptive classification and the write detection performed. However, *SnoopingTLB* only reduces it by 31%. Comparatively, as *OS-RO* does not require additional TLB traffic and has write detection capability, it reduces overall traffic over *SnoopingTLB* even though it performs a non-adaptive classification.

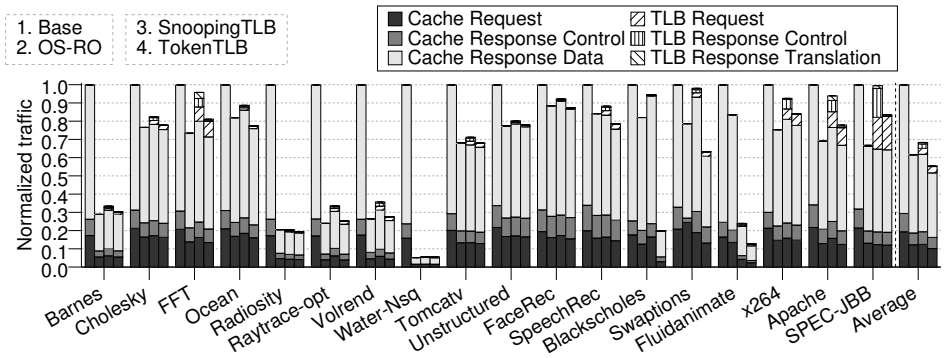


Figure 5.13: Network flits injected, classified into cache- or TLB-traffic.

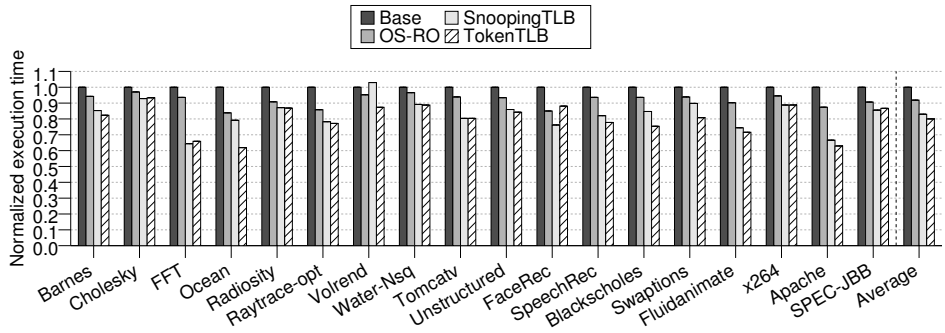


Figure 5.14: Execution time normalized to baseline.

Reducing cache misses and accelerating page translation through TLB-to-TLB transfers has a direct positive impact on execution time, which is improved by 20% using *TokenTLB* compared to *Base*, as shown in Figure 5.14. Also, execution time is reduced by 3% compared to *SnoopingTLB*, as *TokenTLB* unblocks page access earlier after TLB misses, and the L1 cache misses are reduced to a greater extent. As *OS-RO* does not benefit from fast TLB miss resolution through TLB-to-TLB transfers, its gaining is provided solely by coherence deactivation, reducing execution time just by 8.8% over *Base*.

Our proposal also entails a reduction in the cache hierarchy dynamic energy consumption, as shown in Figure 5.15. *TokenTLB* reduces overall consumption by 21.9% compared to the baseline, particularly L1 cache energy consumption, since cache pressure is reduced through coherence deactivation. Network consumption is also significantly decreased, although proportionally to the total consumption its impact is diluted. Comparatively, *TokenTLB* reduces the dynamic consumption by 5.7% with respect to *SnoopingTLB*, and 9% with respect to *OS-RO*.

All in all, applying TokenTLB to coherence deactivation improves the system scalability. It greatly reduces directory entries due to both its full-adaptive classification and the write detection capability, including accesses to Read-Only pages among non-coherent data. As a consequence, TokenTLB allows smaller and more efficient directory structures, while it effectively reduces L1 cache misses, network traffic, dynamic consumption, and improves system execution time over any previous classification mechanism.

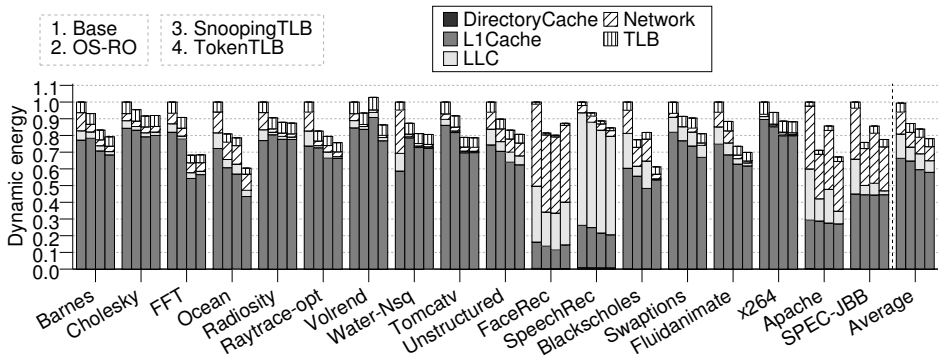
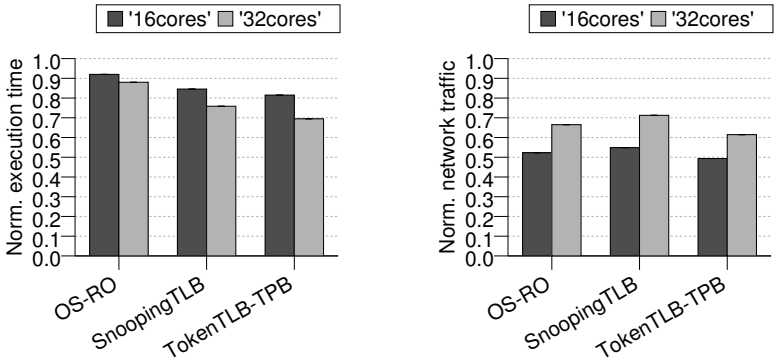


Figure 5.15: Dynamic energy consumption normalized to the base system.



(a) Normalized runtime **(b)** Total flits injected

Figure 5.16: Scalability analysis of classification approaches when applied to coherence deactivation.

5.4.3 System Scalability

This section shows how the different classification approaches scale when applied to deactivate coherence. Due to the slowness of the simulation tools, this study is only performed using SPLASH 2 benchmarks and scientific applications.

Figure 5.16a shows how *TokenTLB-TPB* scales better compared to *SnoopingTLB* and *OS-RO*. All results are normalized to baseline, without deactivating coherence maintenance. Specifically, *TokenTLB-TPB* reduces the execution time by 30.5% on a 32-cores CMP, while *SnoopingTLB* reduces it by 25.15% over the baseline. This difference evidences how to perform a more accurate classification entails better directory usage and ultimately better system performance.

Note how among the aims of *TokenTLB* is to achieve a more scalable system in terms of network usage compared to previous classification approaches. In this sense, Figure 5.16b shows how *TokenTLB-TPB* reduces traffic over the baseline to a greater extent, whereas *SnoopingTLB* jeopardize the system scalability with TLB snoop overhead. Expressly, *TokenTLB-TPB* maintains the traffic reduction up to 38.6% for a 32-core system. Conversely, *OS-RO* in only able to reduce the traffic by 32.5%, and *SnoopingTLB* merely by 28.7%, both for a 32-core system.

5.5 Conclusions

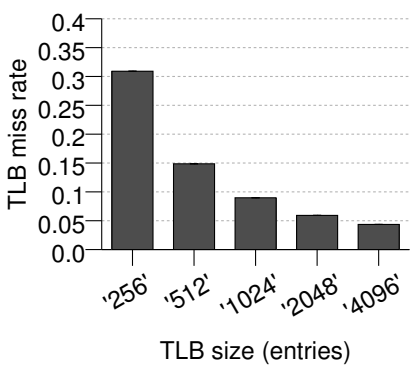
This chapter introduces *TokenTLB*, a novel TLB classification mechanism based on counting and exchanging tokens through TLB-to-TLB requests, where only TLBs *owning* the translation are allowed to answer. When tokens are used for classification instead of using them for coherency, the need for persistent requests is avoided since an owner token is not required. Moreover, token counting is highly efficient for performing an adaptive classification into a private-shared scheme, naturally detecting private phases of pages during its page generation time. Finally, classification dichotomy is extended by allowing write detection for the first time for an adaptive classification mechanism. By predicting token holders and sending unicast messages, the scalability of the system is improved. The proposed TokenTLB presents all the desirable characteristics of a classification scheme: the classification is known before the cache access (*a-priori*), detects *read-only* accesses, is *adaptive*, and is *accurate*.

Chapter 6

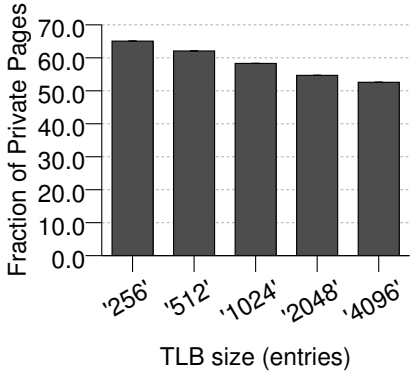
Prediction-based Classification Mechanisms

This chapter introduces a page usage prediction mechanism for TLBs, namely Usage Predictor (UP). A TLB usage predictor anticipate when a page is going to continue being accessed from a given core. This allows invalidating translation entries when (i) the predictor indicates that the corresponding page is not going to be accessed in the near future, and (ii) other TLB requests the page translation. This way, opportunity for private data detection with a TLB-based mechanism is significantly improved.

6.1 Introduction



(a) TLB miss rate evolution depending on TLB size.



(b) TLB classification accuracy depending on TLB size.

Figure 6.1: TLB size analysis overview.

So far, TLB-based approaches are seemingly the best data classification systems in terms of cost and area, enhancing system performance through TLB-to-TLB transfers. Nonetheless, forasmuch as the sharing condition of a page in a TLB-based classification mechanism is determined by the existence of the page translation in the system TLBs, its precision is strongly

dependent on TLB size. The disadvantage of this dependence is shown in Figure 6.1. For smaller TLBs, TLB miss ratio is considerably greater (Figure 6.1a). As expected, increasing the number of TLB entries from 256 to 4096 progressively improves TLB hit ratio. However, the percentage of private pages detected is reduced as TLB size is increased (Figure 6.1b). Page translations remain longer on larger TLBs, even though pages may not be accessed anymore. An unlimited-size, full-associative TLB structure would never miss twice on the same page translation lookup, but the private data detected would decrease to figures similar to a non-adaptive approach (e.g., OS-based classification [22]). Ideally, the sharing condition of a page should be settled by the concurrence of accesses to that page. In this way, TLB-based classification would accomplish a precise classification of private pages, be independent from the TLB size, and not hurt TLB hit ratio. Usage Prediction for TLBs is proposed seeking for this ideal classification.

6.2 Usage Predictor (UP)

In order to determine page-access simultaneity, each processor has to predict if it is going to access a page in the near future. This way, the use of data on pages within different phases of applications is accurately determined. TLB-based approaches make predictions based on the presence of the page translation in system TLBs. However, in order to make predictions independent from the TLB size, this chapter introduces a TLB Usage Predictor (UP), similar to the one proposed by Kaxiras *et al.* to save power in data caches [40]. UP is intended to be used alongside a TLB-based classification mechanism for purely-private TLB structures (e.g., *SnoopingTLB* or *TokenTLB*).

Note how in previous chapters classification can be seen as a naive "prediction" based in the translations stored in the system TLBs. This way, if the page entry is not present in the TLB, the core assumes that it is not going to be accessed in the near future. This situation can happen either because the page has been never referenced by the core or because the entry has been evicted from the TLB since it has not been referenced for some time (TLBs employ a least recently used –LRU– policy). This leads to a strong TLB size dependence.

Nonetheless, usage prediction works as follows. If the page entry is present in the TLB, a 2-bit saturated counter is kept. This counter is increased periodically according to a certain *disuse period* and is reset when any block within the page is accessed by the core (i.e., on a TLB hit), which implies a previous TLB hit in VIPT caches. If the counter for a given entry saturates, the entry falls into *disuse*. Cores consider disused TLB entries as not going to be accessed in a near future. Next, a disused TLB entry is invalidated when requested from the network due to a TLB miss from another core. Consequently, cores with invalidated TLB entries cease being accounted as potential sharers, increasing the opportunities for private data detection. Note how a bad predictor (i.e., incurring in many wrong predictions) can increase TLB and cache misses due to the TLB-cache inclusion policy. Finally, if the page entry is present and in-use, the core predicts that it will be accessed again.

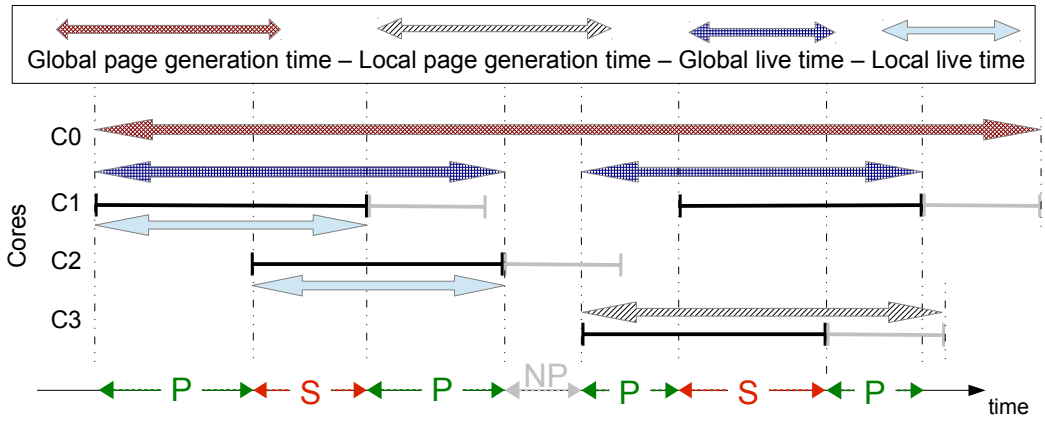


Figure 6.2: Page idealization: generation and live times.

6.2.1 Exploring the Limits of a Temporal Classification

In order to determine the precision limits of a TLB-based classification, and explore and smartly design these new prediction-based classification schemes based on current page usage, we first analyze how page sharing patterns evolve along time. To do this, we define the concept of *local page generation time* in a core as the time spent from a page is first accessed (and missed) on a core's TLB until the page is finally evicted from that TLB. A page may have several local generation times on the same core or in different cores. Notice also that the local page generation times of different cores may overlap or not along time, which will decide whether the page is private at a certain point in time (i.e., do not overlap) or shared. So, a page could be classified initially as private, transition to shared, then back to private and so on, depending on the overlapping of its local generation times in different cores. Thus, we define the *global generation time* as the elapsed time from a page is first cached on a TLB to the moment it is evicted from the last TLB in the system. This means that the global generation time expands along the overlapped local or individual generation times in cores. For private page generations, local and global generation times are equivalent.

However, this definition used to classify pages is just an approximation to the ideal. The sharing condition of a page in the system should be settled by the simultaneity of accesses to that page. If a page is not going to be accessed again from a given core, as far as the translation entry remains in the TLB it still counts as present in order to determine its sharing status, resulting in an inaccurate classification. As we are interested in analyzing the potential of an ideal page-based classification mechanism, page *dead time* (time from the last access to a page until its eviction in a core TLB) is excluded, therefore accounting only for the page *live time* (elapsed time since the page is fetched until its last access before being replaced in a core TLB) [86], i.e., the black lines on Figure 6.2. As with the page generation, we also define both local and global live times. This will allow us to determine to what extend this classification imprecision could affect to the effectiveness of the proposed page classification mechanism.

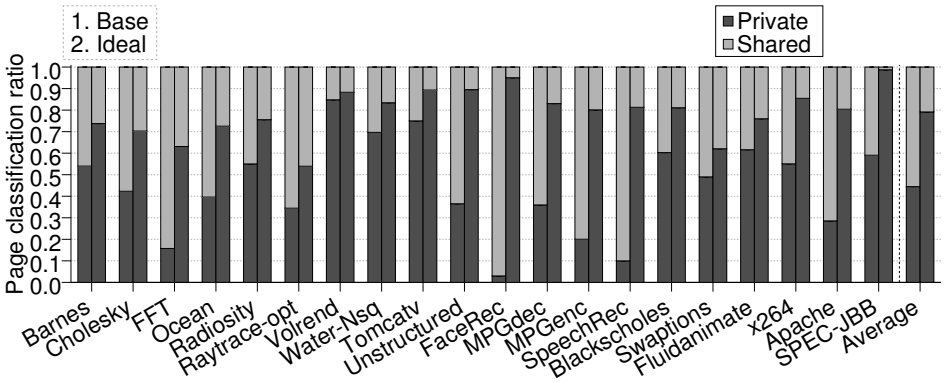


Figure 6.3: Idealized page classification comparison.

Idealization of page classification

We compare the page classification provided by an OS-based mechanism against the classification obtained from the ideal TLB-based mechanism defined above[1]. As the status of a page varies along time, to simplify the analysis we display a page as shared if at some point in time it has been classified as shared. In its turn, a page displayed as private has been always classified as private along time. That means that its live times on different cores never overlap. In other words, a page could live in different cores and remain private, as long as there is a strict exclusion on their live times. Notice that a particular case would be one in which the different lives of the page always elapse on the same core. On the contrary, the page will be classified as shared when its live times overlap at some point in time, despite the fact that its status may be private for most of the execution time.

Figure 6.3 shows the aforementioned comparison for both classification mechanisms: the *Base* bar represents the static classification mechanism performed by the OS, and the *Ideal* bar represents the page classification obtained when idealizing the generation time of a page, thus excluding the dead time. As can be seen, the *Ideal* mechanism is able to classify as private, on average, 35% more pages than the *Base* classification for a total amount of nearly 80% of pages.

In order to analyze the potential of an adaptive classification approach in which pages can be reclassified as private once after they have been shared, next we quantify the amount of lives of a shared page, with respect to the total number of lives of the page, in which it remains private. Figure 6.4 shows this percentage of private lives on shared pages (SP), and the percentage of shared lives on shared pages (SS), evidencing that one out of two shared page lives, on average, could be reclassified to private.

[1]Results are obtained with the system configuration defined in Chapter 3, working on top of a SnoopingTLB classification scheme.

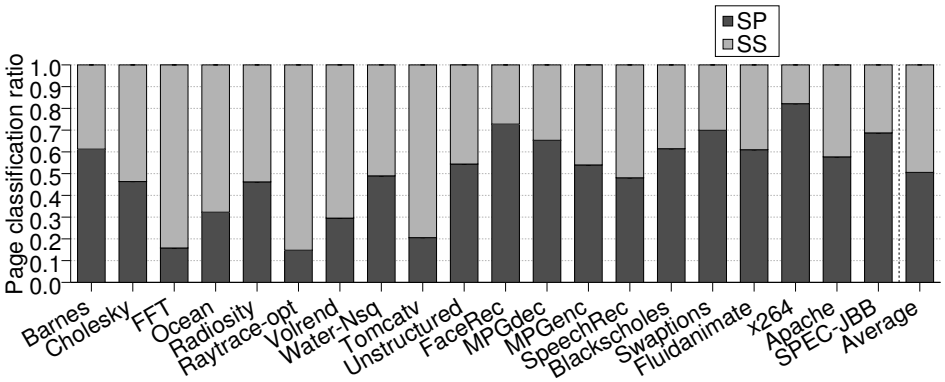


Figure 6.4: Page lives classification of shared pages.

6.2.2 Forced-Sharing Strategy

TLB usage prediction technique is quite effective to carry out a greedy classification of pages as private as a result of the capability to precisely predict the live time of a page (a thorough study is presented in Section 6.2.1). However, due to the high variability on the number of cycles of a page live time or the access interval itself, UP could, specially with low disuse periods, aggressively invalidate an entry that might be accessed again soon. Furthermore, the inclusion policy between the TLB and the L1 cache causes these premature invalidations to be specially harmful to the overall system performance, as it will evict L1 cache lines still being exploited, therefore dramatically increasing both TLB and L1 cache miss rate.

To mitigate this drawback of employing UP it may be preferred in some cases to ignore the prediction, allowing the translation entry to remain valid, and the page to become shared. To this end, it is required to have some kind of evidence of the possible occurrence of premature invalidations, so that it allows us to override the prediction. In this context, when a TLB miss occurs (TLB entry is invalid or *Not Present*) but the requested TLB entry is still allocated in the local TLB, we might suspect that a premature invalidation occurred, and therefore, it may be advisable to cancel the predictor-induced invalidation on other core's TLBs for the corresponding TLB snoop.

To carry out the prediction override, we propose the use of a special request, which will be referred to as *forced* request, as it indeed forces the classification of a page as shared that otherwise could have been classified as private. A forced request is sent on a TLB miss whenever the TLB entry, despite being in invalid state (i.e., *disused*, or *Decayed* in the figure), continues to be allocated in the TLB. On the arrival of a *forced* request to the remaining core's TLBs, invalidation is ignored for *disused* TLB entries, in anticipation of being accessed in the near future. Thus, the page is subsequently classified as shared. Figure 6.5 shows how a TLB behaves when receiving a *remote forced* request applying the *Forced Sharing* mechanism. Basically, when an entry is either on *Decayed P* or *Decayed S* states (i.e., the entry has already fallen into disuse), and it receives a *remote forced* request it transitions to *Present S*, forcing it to become shared and preventing the invalidation of other disused TLB entries, and consequently all the L1 cache entries that pertain to the same page.

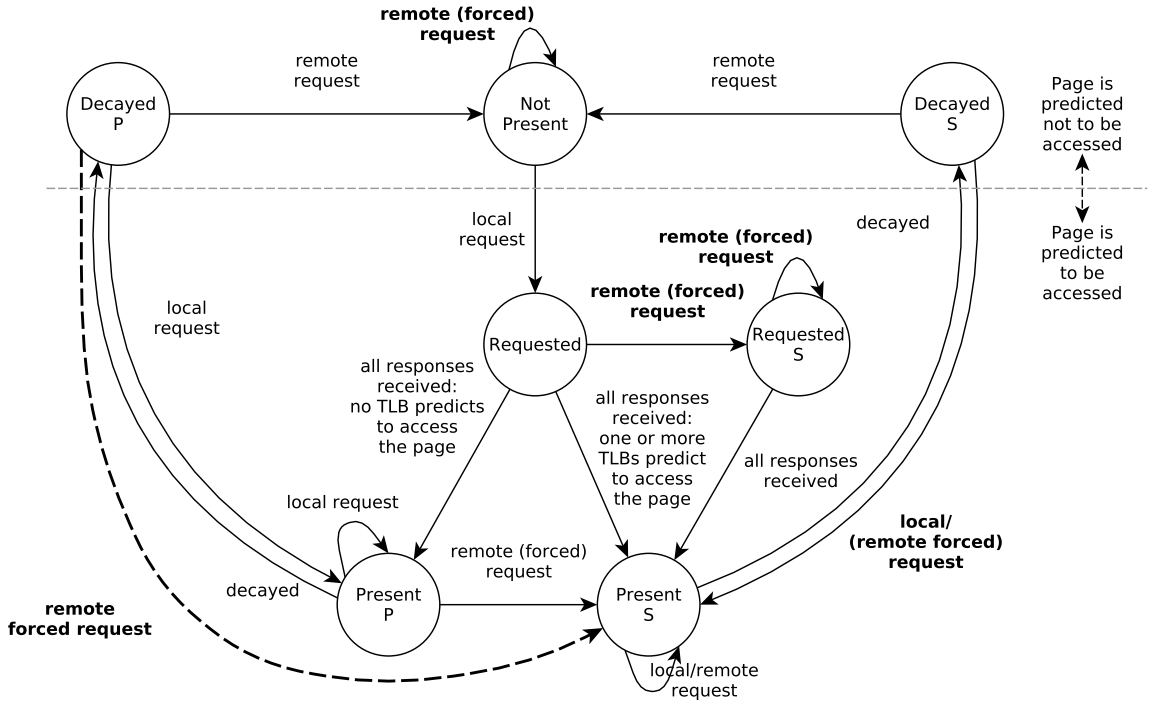


Figure 6.5: TLB state transition diagram with forced-sharing UP

6.3 Shared Usage Predictor (SUP)

In the context of a directory TLB-based classification mechanism (i.e., employing a shared last-level TLB), classification depends solely on the size and associativity of the private TLB structure (i.e., L1 TLB in our simulated environment). Again, usage prediction might be valuable to overcome this dependence, specially for deeper TLB structures. Unlike usage prediction for purely-private TLB hierarchies, which relies on broadcast TLB-to-TLB requests to discover the usage status of a TLB entry, DirectoryTLB sends a single request to the home L2 TLB bank. As a consequence, the TLB usage predictor needs to be reworked and tuned for this particular environment.

We propose a shared TLB usage predictor (SUP), which naturally avoids some of the overheads for snooping-based usage predictors. As seen in Chapter 4, the sharing status of a page is managed by the classification directory in the shared L2 TLB through the P bit and the

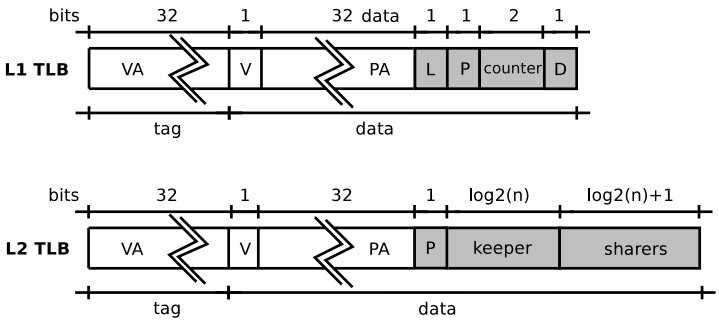


Figure 6.6: L1 and L2 TLB entries, with the SUP fields in gray.

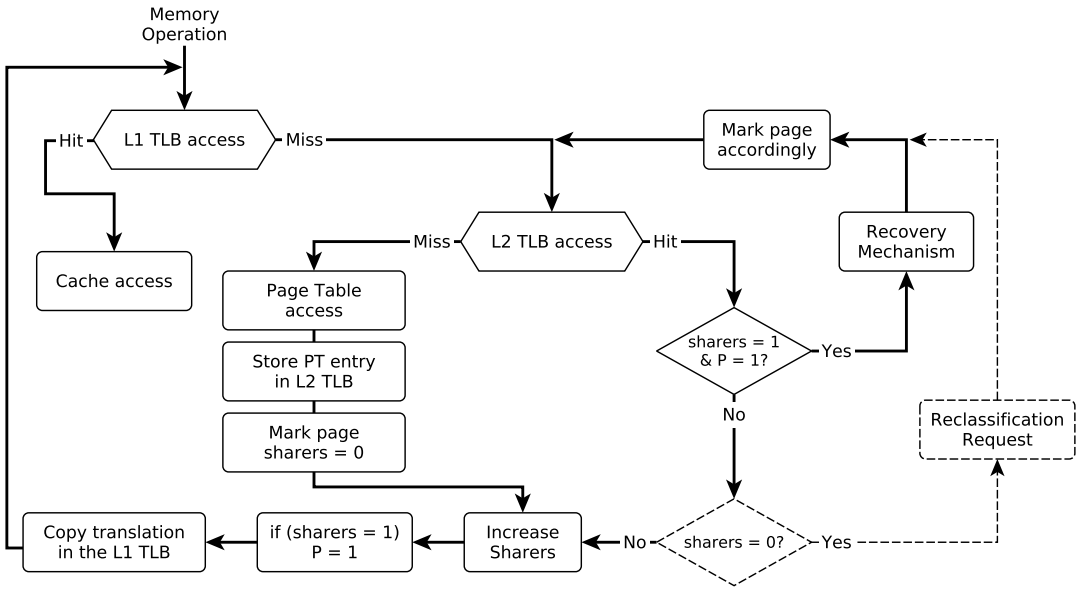


Figure 6.7: Block diagram of the general working scheme with SUP under a memory operation.

sharers count. Moreover, SUP employs the 2-bit saturated disuse counter only for L1 TLB entries (see Figure 6.6), and adds a new *Disused* (*D*) bit, which is set when the disuse counter saturates for the first time. Altogether, 4 additional fields are required per L1 TLB entry, for a total of 5 bits which are insensible to system size. Thus, the silicon area overhead, according to CACTI, is only ∼4% of the L1 TLB area for the baseline configuration (Chapter 3).

Every time the *D* bit is set, a *disuse announcement* is sent to the corresponding L2 TLB bank, which decreases the *sharers* counter. Therefore, we operate under the assumption that the page is not going to be reaccessed soon from a core that has already fallen into disuse. Later, if an access occurs, the counter is reset, but the *D* bit remains set. No more messages (e.g., disuse announcements) are sent to the L2 TLB bank while it remains so, even if a disused TLB entry is evicted (i.e., disused translations evict silently). While the *D* bit is set, sharers field has already been appropriately decreased and the count is updated. Therefore, the *sharers* field tracks only the number of sharers that are currently using the page. This way, reclassification opportunities are unveiled prematurely. However, reclassification requires probing L1 TLBs as they might have been reaccessed from the time they fell into disuse.

Reclassification process. In particular, every time an L1 TLB miss hits in the home L2 TLB bank and the *sharers* count is 0, a reclassification process is triggered (see Figure 6.7). Two different reclassification strategies are supported. On the one hand, in case a reclassification opportunity arises while the *P* bit is unset (i.e., the page has been shared), the home L2 TLB bank starts a reclassification attempt by sending a broadcast request (excluding the requester that initiated the reclassification). L1 TLBs reply according to the information of their predictor counters: i) if the counter is saturated (i.e., the page is currently not in use) the L1 TLB invalidates the page entry (flushing the blocks in the L1 cache) and responds with a NACK; ii) a reaccessed L1 TLB unset its *D* bit and responds with an ACK; iii) if not present, it still has to respond with a NACK. When all responses are collected, the *sharers* count is updated accordingly to the number of positive acknowledgments received. Finally, the miss that originated the reclassification is resolved (which increases the sharers count once more), setting *P* according to the resulting sharers count (i.e., private if there is a single sharer and shared otherwise).

On the other hand, when a reclassification process starts for a private page (the sharers count is 0 and the P bit is set), only the keeper needs to be probed with a unicast request. Therefore, a reclassification process might be used in order to keep the classification as private for longer. As in a broadcast reclassification, the *keeper* responds with a positive or negative acknowledgement depending on its currant usage prediction. Finally, the sharers count in the L2 TLB is either kept as 0 (and the page is brought as private for the requester, which becomes the new *keeper* and the only sharer that is accounted for), or restored to 1 (and the page subsequently transitions to shared after resolving the reclassification).

In addition, the *forced-sharing* optimization can be also adapted in the context of a shared L2 TLB, although premature invalidations are not probably going to hurt system performance under this configuration. The reason is that *disused* entries are only invalidated when the L2 TLB considers that a reclassification probe should take place, naturally acting as a filter for premature invalidations. Nonetheless, L1 TLBs send a *forced-sharing* request to the L2 TLB when the page is accessed and their corresponding entry is found stored in the TLB in an invalid state. If the miss triggers a reclassification, the probes issued to L1 TLBs just unset their respective D bits rather than invalidating the translations (independently from the status of the saturated counter). Thus, the page is kept as shared and can be securely reaccessed without incurring extra L1 TLB misses due to predictor-induced invalidations.

6.3.1 SUP Classification Transitions

Classification status is updated as a consequence of L1 TLB miss requests, evictions, or disuse announcements, and L2 TLB evictions. Figure 6.8 depicts the state-transition diagram.

Pages in the L2 TLB can be in four states: (i) not present; (ii) present but not in use in any L1 TLB ($sharers = 0$); (iii) present and private with only one L1 TLB holding the entry ($P = 1$ and $sharers = 1$); and (iv) present and shared with one or several L1 TLBs currently holding the entry ($P = 0$ and $sharers > 0$).

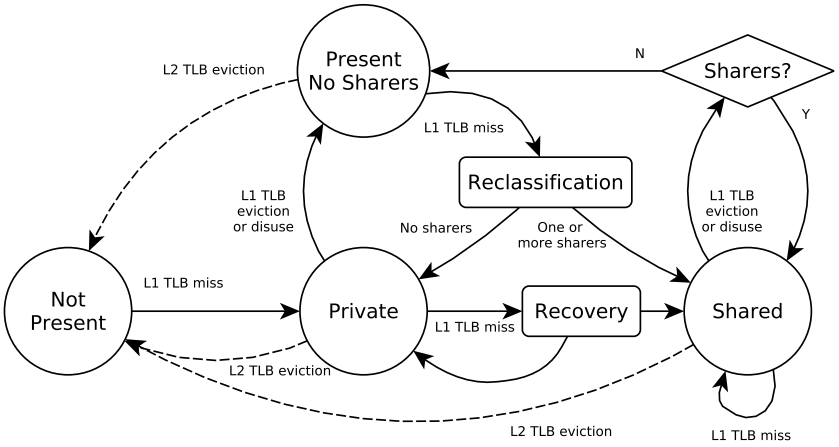


Figure 6.8: L2 TLB classification state diagram with SUP.

If the page translation is *Not Present* in the L2 TLB and a miss request is received, the page transitions to *Private* after "walking" the page table. Then, if a different L1 TLB misses while the page is in *Private* state, a recovery mechanism is initiated and the page transitions to *Shared*. Note, thought, that classification on a TLB-based mechanism with a shared TLB structure might transition back to *Private* if a race condition occurs, as explained in Section 4.4.2. On the other hand, receiving an eviction or a disuse announcement for a *Private*

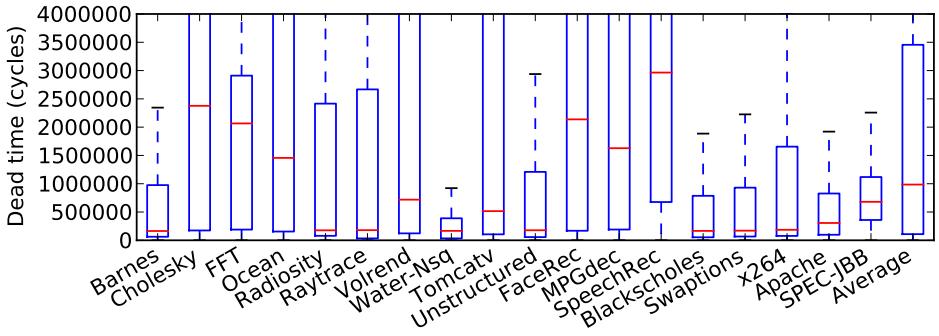


Figure 6.9: Average time (cycles) from last access to a page in a core to its eviction in the TLB

page causes a transition to *Present No Sharers*. In this state, the L2 TLB acts as a victim TLB for the next missing L1 TLB. When the miss occurs, disused entries might have silently reaccessed the page translation and the classification coherence must be assured by means of a *Reclassification* request, as described in Section 6.3. If the reclassification ends up with no sharers, the miss is resolved to *Private*. Otherwise, the miss is resolved to *Shared* and the sharers count is updated. Finally, further L1 TLB miss requests for a *Shared* page keep the page in the same state, just increasing the sharers count. Receiving evictions or disuse announcements for a *Shared* page decreases the sharers count. Finally, the page transitions to *Present No Sharers* only if the count reaches 0.

6.4 Approaching to the Ideal Scheme

Section 6.2.1 analyzed the potential of a TLB-based temporal-aware mechanism by idealizing the live time of every page in every TLB. Now, we are interested in measure the behavior of pages in the TLB in order to approach the classification to the idealized page generation time. To do that, rather than waiting for the TLB entry eviction, a predictor is employed as explained in this chapter to estimate when the last page access occurs and invalidate the translation. This early invalidation mechanism is based on a timer (i.e., predictor period). The timer needs to be small enough in time to tightly approximate to the ideal, but sufficiently long as to avoid false evictions of pages that are going to be accessed again soon, which could cause severe damage to the system performance.

In this sense, Figure 6.9 represents the average page dead time (i.e., the time in cycles spent from the last access to the page TLB entry to its eviction). As this time metric is extremely variable through the samples of the different considered applications, the data is displayed as a box-and-whiskers plot. Here the focus is put on the median, which on average is located close to 1 million cycles. This means that an early eviction done 1 million cycles after the last access to a page will do successfully evict all the greater samples, which on average symbolizes half of the page lives. Yet this represents at the most an upper limit to the timer used to an early eviction. In some applications(e.g., Barnes, Swaptions, etc.) this value will act in barely a 25% of the page lives, evidencing that lower values could perform better. In particular, the lower limit of a timer employed to early evict TLB entries should never be lower than the inter-access time of those entries to ensure no harming the system performance. Figure 6.10 shows the average number of cycles between accesses to the same page translation entry in a single TLB. The number of cycles to evict the page should at least be approximate to the limit of the third quartile to avoid affecting negatively the system (with at least 75% of the samples

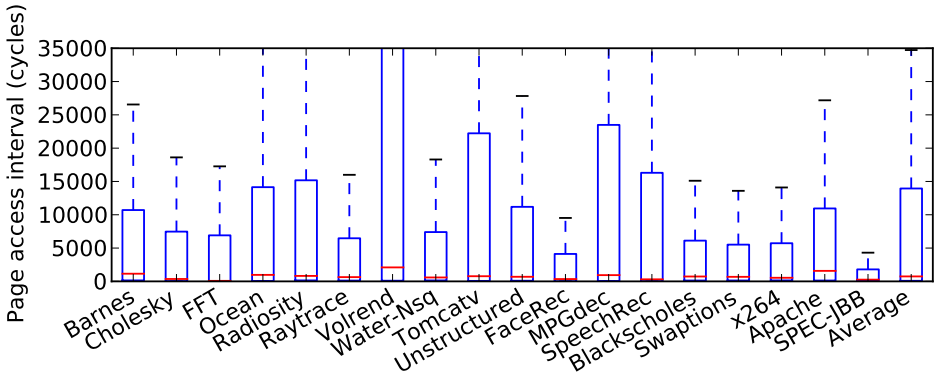


Figure 6.10: Average time (cycles) between TLB accesses

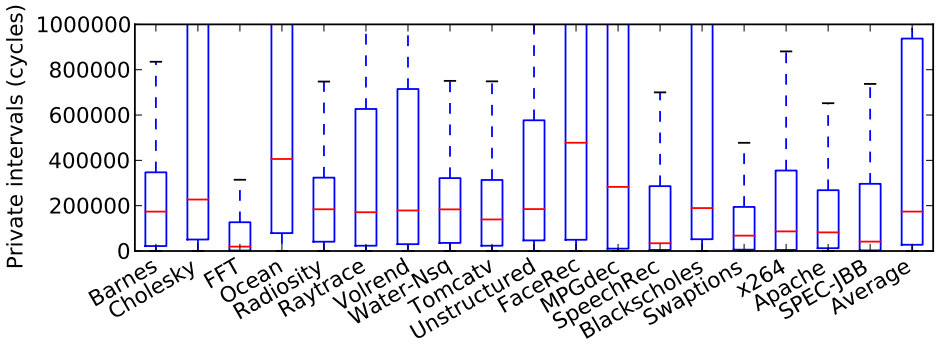
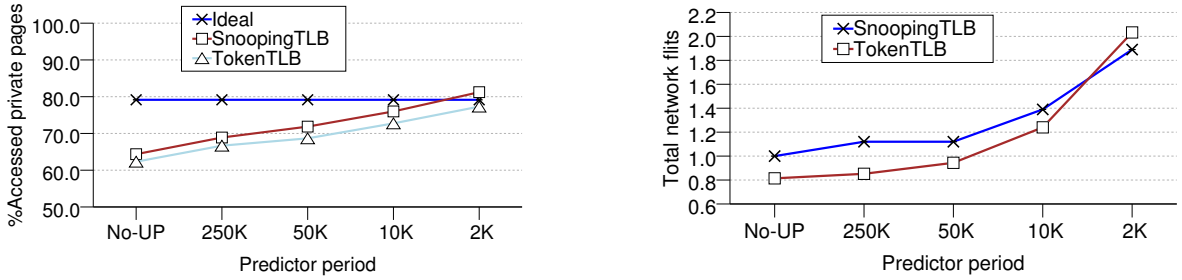


Figure 6.11: Cycles spent as private on a global live

below), meaning on average 10,000 cycles approximately. Although in some applications the limit is close to 25,000 cycles, turning out to be a safer lower value.

Lastly, as previously seen on Figure 6.2, there are intervals of a global page live where it becomes private, either to finally be evicted or to become shared again after some cycles. In these cases a notification mechanism able to track and update the sharing status of the page in the TLB may provide some benefit if those time periods are sufficiently long. Figure 6.11 shows how the samples representing these intervals are distributed. Note that a shared global page live will always start as private, but this period is negligible and is not represented on the graph. The longer the samples the more efficient and justifiable could be a notification mechanism to discover these private periods and update the page status accordingly. The median, on average, is placed around 200,000 cycles, although it evidences a high variability among the different applications.

So, we can conclude that a reasonable choice for a predictor timeout to approximate the start of a dead time of a page should never be lower than 10,000 cycles and up to a threshold of 1 million cycles. Since, as seen in Section 6.2, four predictor periods are required for a page to fall into disuse, we chose 2,000 cycles as the lowest period in the study (i.e., 8,000 cycles timeout), 10,000 cycles period (i.e., 40,000 cycles timeout), 50,000 cycles period (i.e., 200,000 cycles timeout), and finally 250,000 as the highest period (i.e., 1 million cycles timeout). On the other hand, the high variability of the private intervals of page generation times could discourage the use of a notification mechanism that relies on broadcast announcements, as it could be detrimental both to system performance and network consumption.



(a) Private page proportion

(b) Total flits injected

Figure 6.12: Comparative analysis: UP on top of SnoopingTLB or TokenTLB.

6.5 Experimental Results

Usage prediction for TLBs is to be implemented on top of a TLB-based classification scheme employing TLB-to-TLB transfers, and both SnoopingTLB and TokenTLB are possible choices. Four different predictor periods are employed, following the results in Section 6.4: 250,000, 50,000, 10,000, and 2,000 cycles. Token Predictor Buffer (TPB –Section 5.2.4) is not employed in either case. System and network configuration are as described in Chapter 3. Firstly, we show the main implications of each of the two TLB-based classification alternatives and discuss their differences. Then, we study the influence of prediction techniques for TLBs, considering both the *Base* predictor and *Forced-Sharing* predictor, on performance and classification accuracy. Next, a predictor for shared last-level TLBs (i.e., *SUP*) is analyzed, and compared against UP in terms of prediction overheads and their behavior when varying the core count.

6.5.1 UP: SnoopingTLB versus TokenTLB

Usage prediction is based upon the assumption that the classification mechanism consults the system TLBs for retrieving the sharing status (i.e., after a TLB miss). Therefore, both *SnoopingTLB* and *TokenTLB* are good candidates for working in conjunction with UP. Figure 6.12 shows the main differences between both options.

On the one hand, Figure 6.12a shows the proportion of private pages detected both for *SnoopingTLB* and *TokenTLB*. Also, the figure includes the *Ideal* shown in Figure 6.3. As discussed on Chapter 5, *TokenTLB* unblocks page access when the miss is resolved (i.e., the translation is received from another TLB, alongside some tokens), even though there may be some tokens still in-flight. It slightly lessens the amount of private pages detected, compared to *SnoopingTLB*. Specifically, nearly 4% more pages are classified as private with *SnoopingTLB* for a predictor period of 2K cycles. In both cases, the amount of private pages detected increases inversely proportional with the predictor period. Furthermore, *SnoopingTLB* slightly surpasses the *Ideal* (2% more private pages), which could be detrimental for system performance, as the mechanisms is eagerly invalidating TLB entries, artificially increasing the private detection. *Ideal* is obtained by tightly eliminating the page dead time from every page local generation time in the TLB and determine the private status of the pages through time. Thus, increasing the private pages detected over *Ideal* evidences how some pages have been eagerly invalidated as a consequence of miss-predictions.

On the other hand, Figure 6.12b shows the total network flits issued with UP working on top of both *SnoopingTLB* and *TokenTLB*, both normalized to *SnoopingTLB* without usage prediction. Since *TokenTLB* constraints the TLB response traffic and avoids translation repli-

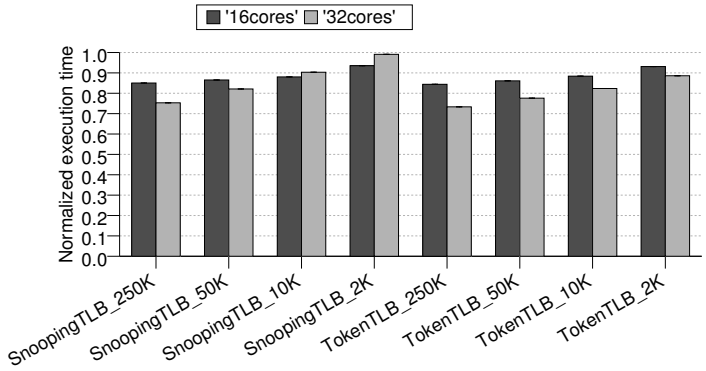


Figure 6.13: Normalized execution time when increasing core count.

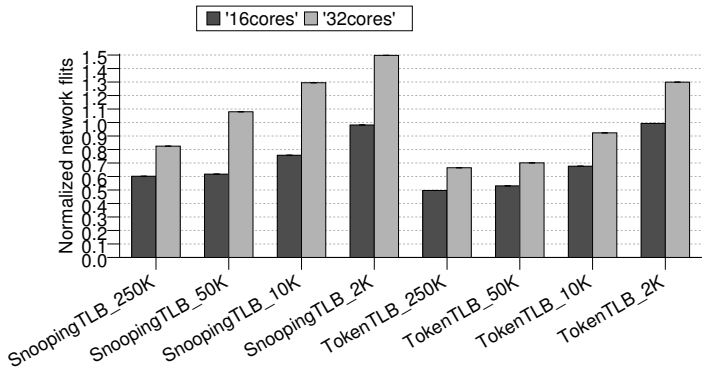
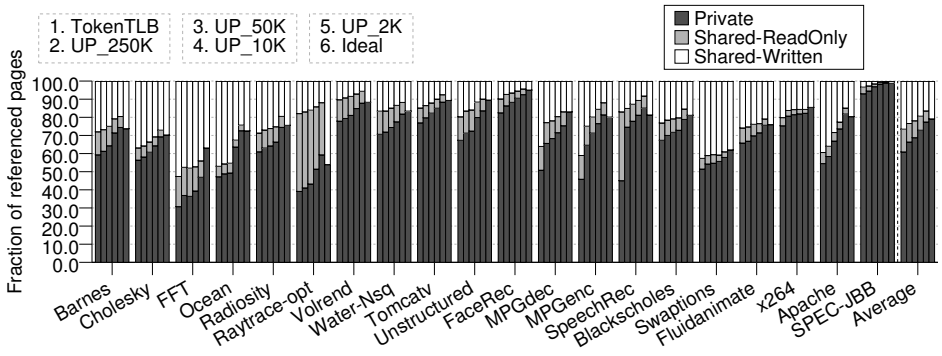


Figure 6.14: Network flits issued when increasing core count.

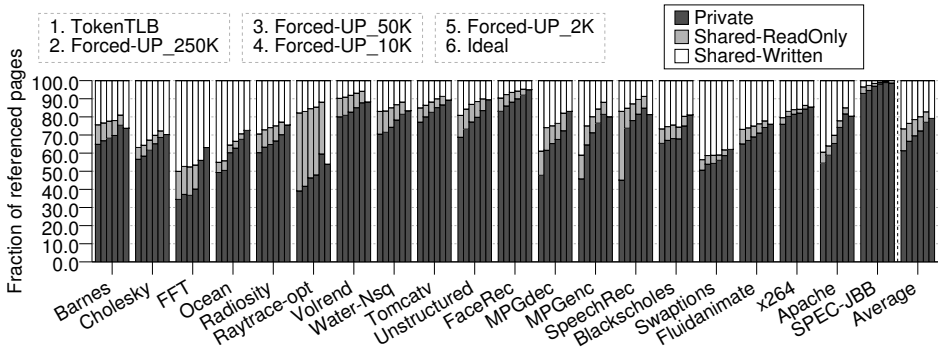
cation on broadcast responses, traffic is restrained compared to *SnoopingTLB*, up to a 26.8% with 250,000 cycles predictor period. However, since usage prediction entails translation invalidation of disused entries, *TokenTLB* rapidly increase network traffic due to non-silent TLB evictions (see Section 5.2.3), to the point of surpassing the traffic issued by *SnoopingTLB* with a predictor period of 2,000 cycles (14.3% more traffic with *TokenTLB*, doubling the traffic compared to not using usage prediction technique).

Scalability analysis. Next, we show how prediction approaches scale when applied to coherence deactivation. Due to the slowness of the simulation tools, this study is performed only with SPLASH 2 benchmarks and scientific applications. Results are normalized to a baseline without coherence deactivation.

Figure 6.13 shows how prediction on *TokenTLB* scales better compared to employing *SnoopingTLB*. Decreasing the predictor period on *SnoopingTLB* hurts execution time in a greater extent than *TokenTLB*, nearly neglecting the benefits of deactivating coherence for a predictor period of 2,000 cycles and a 32-core system. Conversely, *TokenTLB* performs a more accurate classification, with read-only detection and full-adaptivity, which reduces the overheads of prediction on TLBs, improving system performance nearly 11% over baseline for a 32-core system and a prediction period of 2,000 cycles.



(a) Base usage predictor for TLBs



(b) Forced-sharing usage predictor for TLBs

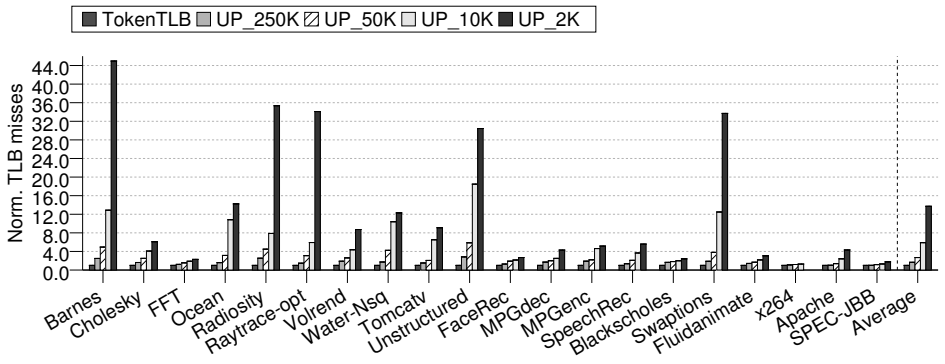
Figure 6.15: Private-shared and read-only page classification.

Furthermore, *TokenTLB* reduces traffic compared to *SnoopingTLB*, specially when increasing the number of cores in the system. *SnoopingTLB* collects responses and replicates translations on TLB transfers after every TLB miss. Consequently, prediction-induced TLB invalidations dramatically increase traffic consumption as the predictor period decreases. Figure 6.14 shows how a predictor period of 2,000 cycles increases the total traffic issued by 49.7% with *SnoopingTLB*. The traffic increase is partially tackled with *TokenTLB*, where the traffic increases by 30% for the same predictor period.

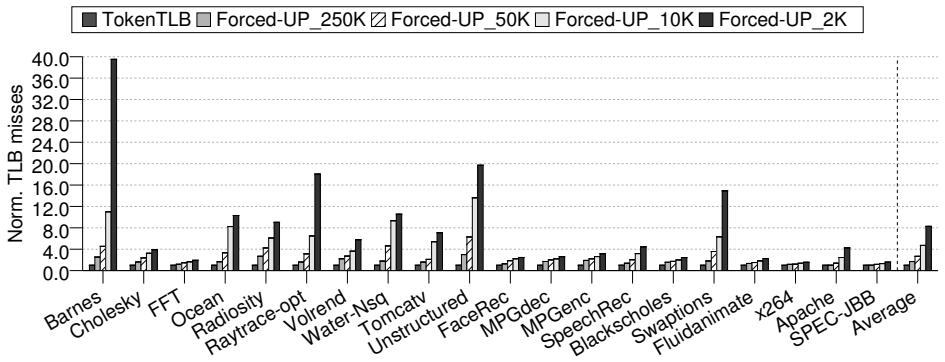
Conclusion. Despite seemingly detecting more private pages in conjunction with *SnoopingTLB*, prediction over *TokenTLB* provides full-adaptivity and read-only detection (see Chapter 5), ultimately benefiting execution time and traffic consumption when applied to data optimizations as coherence deactivation. In what follows, prediction-based classification analysis assumes TokenTLB as its base classification scheme.

6.5.2 Prediction Overheads and Forced-Sharing

This section analyzes how inaccurate TLB usage predictions entail an increase in the amount of TLB misses, and thus, there is a trade-off in the amount of private data detected and system performance. Also, it is shown how forced-sharing allows us to alleviate the problem. UP is applied over a token-based page classification approach (i.e., TokenTLB).



(a) Base usage predictor for TLBs

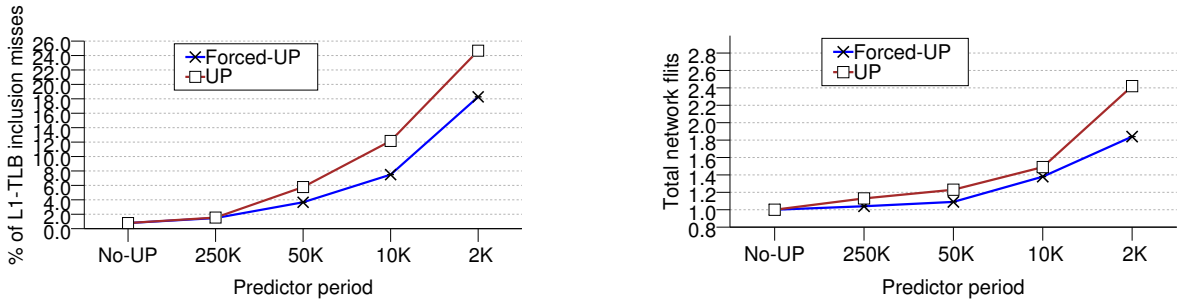


(b) Forced-sharing usage predictor for TLBs

Figure 6.16: Proportion of prediction-induced TLB misses.

Usage prediction increases the amount of private data detected by invalidating disused sharers. Figure 6.15 shows the pages considered as Private, Shared-ReadOnly or Shared-Written with and without prediction mechanism. Particularly, we show numbers for the idealized page live time introduced in Section 6.2.1 (*Ideal*), our base proposal without employing prediction (*TokenTLB*), and prediction-based classification (*UP*). If a page has been considered as shared at least once during the execution of the application, it will be plotted as shared in the graph. First, employing *Base* (Figure 6.15a) increases private page detection from 66.4% to up to 77.4% depending on the prediction period. Classification gets close to the *Ideal*, which represents 79.2% of private pages. Conversely, when using the *Forced-sharing* (Figure 6.15b) approach, private pages detected is slightly decreased, getting up to 77% private pages with the lowest predictor period. Note that thread migration is rare in the evaluated applications due to their short execution times.

Figure 6.16 shows the number of TLB misses normalized to *TokenTLB* (prior to applying a usage prediction). As can be seen, lower predictor periods with *Base usage predictor* (Figure 6.16a) lead to an increase in the number of TLB misses due to the invalidation of disused TLB entries, up to nearly 13 times more misses in the TLB on average with *UP_2K*, which will definitely, negatively impact system performance. Conversely, when using the *Forced-sharing* modification (Figure 6.16b), this trend is much less harmful, halving the number of misses in the TLB compared to UP, but still generating 7.27 times more misses than *TokenTLB* on average with *Forced-UP_2K*. The negative impact of the usage prediction is partly mitigated by the fast TLB miss resolution mechanism, since those extra TLB misses will probably find



(a) Percentage of L1 misses due to the TLB-L1 inclusion policy

(b) Total flits injected

Figure 6.17: Base UP versus Forced-sharing UP.

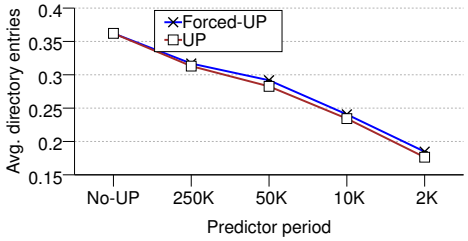


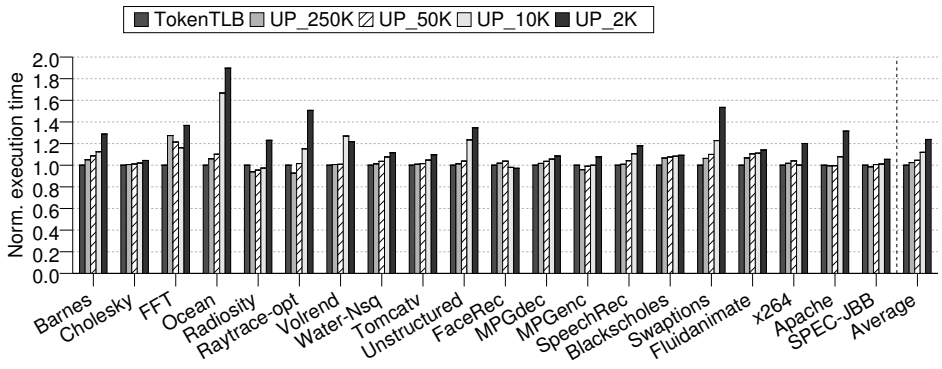
Figure 6.18: Average directory entries per cycle for Base- and Forced-UP.

the address translation in the TLB of the core that caused the invalidation of the disused translation entry in the TLB.

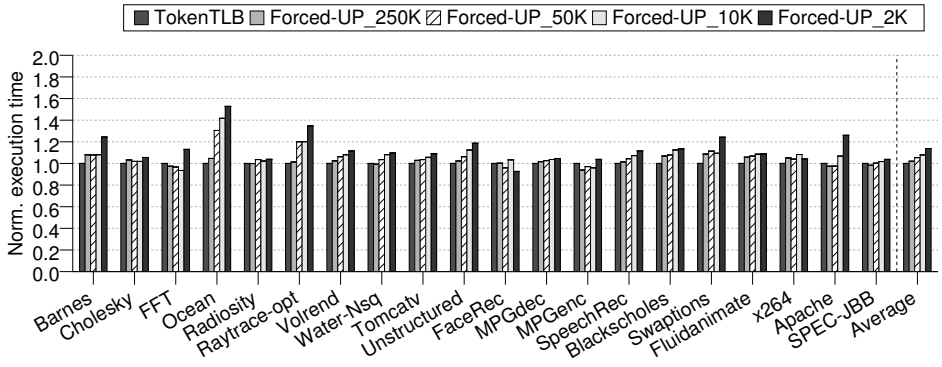
Increasing the amount of TLB misses entails an increase in the misses in the L1 cache due to the TLB-L1 cache inclusion policy. Whenever the predictor invalidates a TLB entry during its dead time, there will not feasibly be many blocks present on the L1 cache, and thus, they will not be likely revisited. However, on premature invalidations, the L1 cache misses, negatively affecting the system performance. Figure 6.17a shows that with *UP* nearly a 24.7% of the L1 misses are due to the inclusion policy when employing a 2K predictor period, on average. *Forced-UP* partially reduces the increase, however misses induced by the inclusion policy still represent 18.3% of all L1 misses with a 2K predictor period.

Furthermore, usage prediction also increases traffic. TLB transfers upon TLB misses and coherence issues after cache misses increase network consumption as predictor becomes more aggressive, due to miss-predicted invalidations. Figure 6.17b shows network flits issued normalized to TokenTLB without prediction (*No-UP*). Network traffic with *UP* gets well over twice the traffic of *No-UP* with 2K predictor period. Again, *Forced-UP* alleviates the traffic increase by avoiding some miss-predicted invalidations, reducing the traffic by 24% over *UP*, but still increasing the traffic by 84% over *No-UP* with a 2K predictor period.

All reported results are obtained with coherence deactivation. The main goal of deactivating coherence is to alleviate directory pressure by avoiding track of non-coherent blocks. Coherence deactivation is employed as a data optimization in order to observe the benefits of improving private detection. Figure 6.18 shows the average directory entries per cycle for *UP* and *Forced-UP*, normalized to a baseline without coherence deactivation. Directory usage is lowered as the predictor period decreases, to just 17.6% directory occupancy with *UP_2K*, 18.6% less directory entries than *No-UP* classification. *Forced-UP* slightly decreases directory usage reduction, by



(a) Base usage predictor for TLBs



(b) Forced-sharing usage predictor for TLBs

Figure 6.19: Execution time normalized to baseline.

roughly 1% on average. Nonetheless, gains with prediction-based classification are notable with both approaches when deactivating coherence maintenance.

Finally, Figure 6.19 shows the execution time of prediction-based classification approaches normalized to TokenTLB, a TLB-based classification approach without usage prediction. In some cases, specially with greater predictor periods (which induce less miss-predicted invalidations), as in Raytrace-opt or MPGenc, the benefit obtained from increasing the private detection when applied to coherence deactivation counterbalances the prediction overheads and execution time is improved. However, on average, both *UP* and *Forced-UP* damage to some extent the execution time compared to *TokenTLB*. However, specially with lower prediction periods, the damage with *forced-sharing* strategy is constrained, as ping-pong invalidations are avoided. Specifically, where *UP_2K* decreases average execution time by 23.8%, *Forced-UP_2K* hurts execution time by 13.8%. However, as seen in Chapter 5, most Coverage misses are avoided with just *TokenTLB*, thus although Figure 6.18 shows how UP notably reduces directory usage to a greater extent, directory ceases to be a performance bottleneck. However, in scenarios with smaller directories, or frequent thread migration or larger TLBs, UP may introduce greater improvements in terms of execution time. Furthermore, alternative or additional private-based or read-only-based optimization could be applied to benefit from the accuracy of a prediction-based approach.

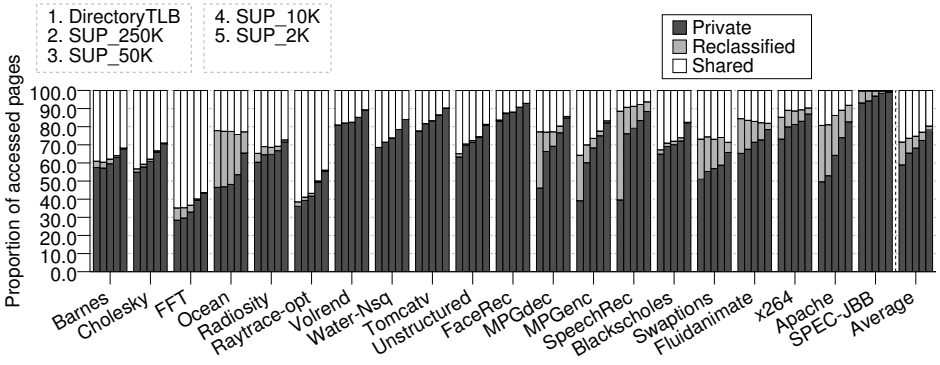


Figure 6.20: Private/shared page classification with distributed shared last-level TLB.

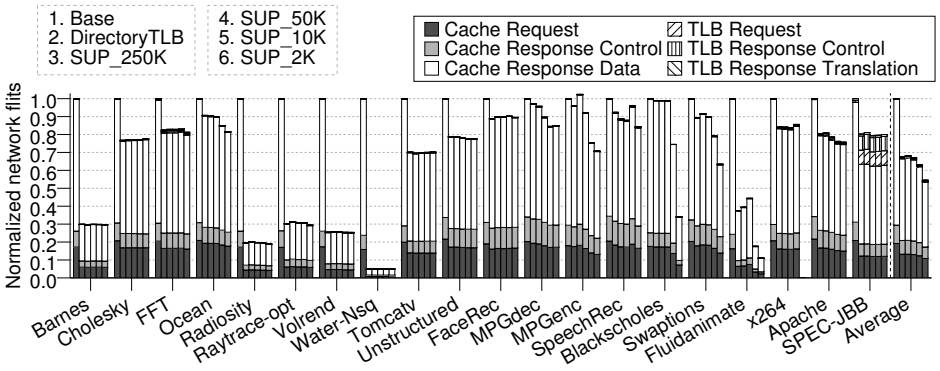


Figure 6.21: Total flits injected with SUP applied to coherence deactivation.

6.5.3 SUP: Prediction with Shared Last-level TLBs

This section evaluates a TLB-based classification for distributed shared L2 TLBs using the shared TLB usage predictor presented in section 6.3. First, we show the effect of SUP to the classification characterization. Next, we apply SUP to coherence deactivation as our case study.

Classification accuracy. Figure 6.20 depicts how pages are classified into private and shared for different classification mechanisms in a scenario with a distributed shared L2 TLB. The evaluation compares *DirectoryTLB*, as detailed in Chapter 4, Section 4.4, and different predictor periods for our shared TLB usage predictor (*SUP*). *DirectoryTLB* does not discern read-only pages, but it performs an adaptive classification, and *Reclassified* pages are depicted in the figure, symbolizing the amount of shared pages that are classified back to private at least once during the application's execution time. On the one hand, *DirectoryTLB* classifies 59.4% of pages as *Private*, and 14.6% as *Reclassified*, showing how precise its classification is at page level. On the other hand, *SUP* increases the proportion of *Private* pages up to a 78.1%, reducing *Reclassified* pages to merely a 2% of all accessed pages for a 2,000 cycles period.

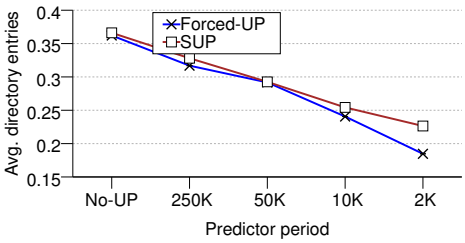


Figure 6.22: Average directory entries stored per cycle with SUP.

Case Study: Coherence deactivation. Figure 6.21 shows the total normalized network usage and its classification into cache or TLB messages when the classification is applied to coherence deactivation. *Base* is a baseline system with the same overall configuration, including a distributed shared second level TLB, but without data classification nor coherence deactivation. Classification mechanisms for shared TLB structures reduce the network traffic since they avoid the costly TLB-to-TLB broadcast transfers. For *DirectoryTLB*, the TLB traffic represents only 1.8% of the total. Furthermore, *SUP* prevents the TLB traffic increase attributed to lower predictor periods, since a shared L2 TLB acts as a filter for miss-predicted invalidations. Even if *SUP* does prematurely invalidate a TLB entry, it only requires a message to the L2 TLB home tile. Consequently, *SUP* only increases TLB traffic to as much as 2.3% for the lowest considered timeout, while the cache traffic is halved. As a consequence, *SUP* reduces total traffic issued by 15.7% compared to *DirectoryTLB* with *SUP_2K*. All in all, *SUP* improves classification accuracy while significantly reducing traffic overhead, which represents a far more scalable approach compared to either a predictor for purely-private TLB structures or *DirectoryTLB* without usage prediction mechanisms.

Figure 6.22 shows the average number of required directory entries for *Forced-UP* and *SUP*, each normalized to a baseline with the same TLB structure (with private and shared last-level TLBs, respectively) and system configuration, but without coherence deactivation. Particularly, *SUP* reduces directory storage similar to the reduction in *Forced-UP* with a purely-private TLB structure. However, on lower predictor periods, directory storage requirements are reduced to 22.6% with *SUP_2K*, while *Forced-UP* further reduces directory usage to just 18.5%. Note that *SUP* relies on the shared L2 TLB as a filter for short-term reclassifications. Our proposal for shared TLB structures initiates a reclassification process only when the home L2 TLB tile is accessed and found in a *Present No Sharers* state (see Section 6.3.1). Even so, a reclassification process may still fail to transition to private again if a core reaccesses a disused page. Conversely, a TLB predictor for purely private TLB structures is more dynamic and forceful, even with forced-sharing strategy, which favors short-term reclassifications albeit possibly hurting system performance.

Finally, Figure 6.23 shows the execution time of different applications employing our TLB-based classification for shared TLB structures under coherence deactivation, normalized to a baseline without coherence deactivation. *DirectoryTLB* reduces execution time by 6.8% compared to baseline. Note that baseline already leverages inter-core sharing patterns with a shared last-level TLB, and therefore the performance gains are only those obtained from coherence deactivation. Additionally, *SUP* further contributes to better system performance, reducing execution time up to 8.2%, even using a 2,000 cycles predictor period. Improving execution time over *DirectoryTLB* proves how prediction overheads (e.g., miss-predicted TLB invalidations) are almost completely avoided with *SUP*.

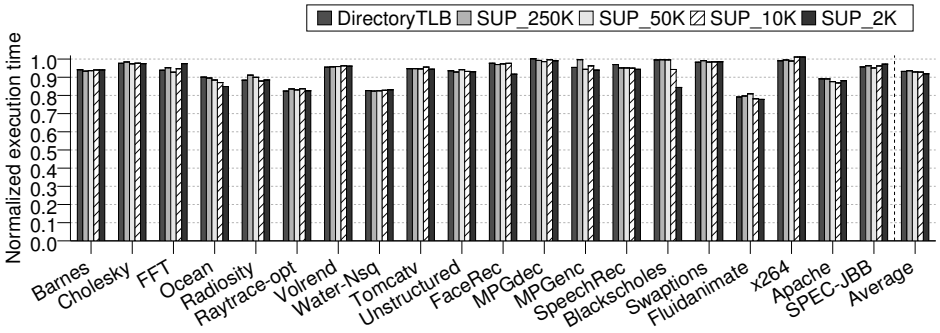


Figure 6.23: Execution time under coherence deactivation normalized to a shared TLB baseline with SUP.

6.6 Conclusions

This chapter evidences one of the main problems of TLB-based classification: the dependence of the classification accuracy with the size of the TLB structure. To avoid this counter-back, we introduce a new family of prediction-based approaches, which allow TLB-based classification to be independent from TLB size by invalidating entries that are predicted not be used in the near future, and granting a more accurate private detection (closer to the ideal). However, miss-predictions come with a cost, limiting the performance gains and discouraging the use of predictors. Forced-sharing strategy alleviates the problem but fails in making prediction completely appealing.

Furthermore, we propose SUP, a predictor that avoids most miss-predictions by leveraging a shared last-level TLB structure, which ultimately reduces traffic and improves execution time over UP, resulting in a more scalable choice. Thus, SUP makes prediction-based approaches attractive, showing the benefits obtained with a more precise private detection. However, SUP does not perform a full-adaptive classification, and not discern read-only pages, which limits its capability to perform an accurate classification. Furthermore, SUP is limited to systems implementing a shared TLB level, which is not a common design choice nowadays.

Chapter 7

Cooperative TLB Page-Usage Prediction Mechanism

This chapter shows that it is possible to remove prediction overheads on purely-private TLB structures by using a novel and elegant token-based usage prediction scheme, while maintaining a high private data detection rate. To this end, we introduce Cooperative Usage Predictor *(CUP), a mechanism that exploits TLB cooperation (i.e., TLBs willingly working together for a common purpose or benefit) in order to perform a system-wide page usage prediction.*

7.1 Introduction

Usage prediction for TLBs (UP) is a promising extension for TLB-based classification, helping to achieve a more accurate classification and making characterization independent from TLB size. However, prediction always comes with a cost (as seen in Chapter 6).

The key observation is that invalidations caused by a Usage Predictor do not necessarily lead to higher detection of private pages, causing a major increase in the TLB miss rate. In fact, many TLB entry invalidations just reduce the number of sharers without the certainty of improving the amount of private data. Hence, UP is applied blindly, without considering how the classification status will evolve, despite being supposedly at its service. Moreover, although the *forced-sharing* UP strategy effectively reduces predictor-induced damage to system performance, the solution is again somewhat dependent on TLB size. With smaller TLBs, invalidated TLB entries are evicted sooner and forced-sharing would not be triggered. With larger TLBs, many pages would be erroneously considered as prematurely invalidated, since translations remain longer in the TLBs, and prediction would be frequently ignored. Besides, further opportunities to reclassify as private may be lost if no other TLB misses occur during a favorable time period.

To illustrate this, Figure 7.1 shows the amount of TLB misses for different predictor periods, using the basic usage predictor for TLBs (*Base-UP*), and the modified *forced-sharing* predictor (*Forced-UP*), both normalized to a TLB-based classification mechanism without prediction. TLB misses are classified into *3C* misses (compulsory, capacity, and conflict), prediction-

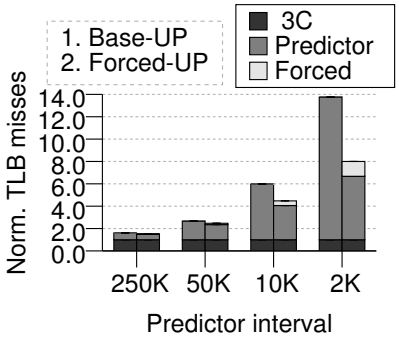


Figure 7.1: TLB misses considering its cause.

induced (*Predictor*) misses, and *Forced* misses (i.e., predictor-induced TLB misses triggering the forced-sharing strategy). As can be seen, *Forced-UP* avoids some prediction-induced TLB misses, specially using the lowest predictor period considered, for which 19% of all TLB misses trigger a *Forced* miss. Nevertheless, the proportion of TLB misses is still increased by more than 6.5 times due to *Predictor* misses, showing how inaccurate the TLB usage predictor is.

These overheads discourage the use of prediction mechanisms for private TLB structures, despite improving the private data detection rate.

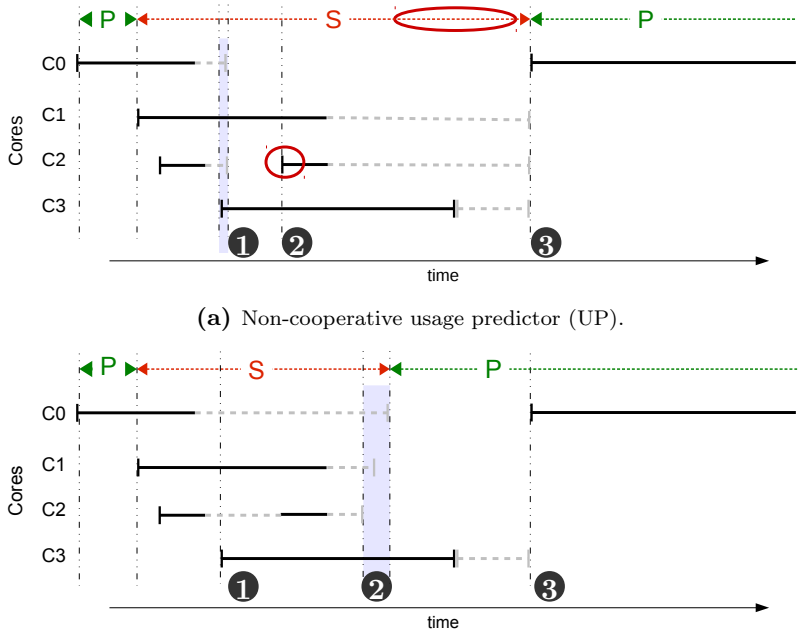
In this chapter we show that it is possible to remove prediction overheads by using a novel and elegant token-based usage prediction scheme for private TLB structures, while maintaining a high private data detection rate. To this end, we propose *Cooperative Usage Predictor* (CUP), a mechanism that exploits TLB cooperation (i.e., TLBs willingly working together for a common purpose or benefit) in order to perform a system-wide page usage prediction.

7.1.1 Potential Benefits

To understand the main benefits of CUP, Figure 7.2 depicts two examples for the global generation time of a given page in the TLBs of a 4-core CMP. Let us remember that the global generation time is defined as the time elapsed from the first page allocation in a TLB until the eviction of the page from the last TLB in the system. Every black line represents a page live time in a single TLB, and the grey dotted lines represent their disuse periods. The upper colored line shows how the page is classified using the pertinent usage predictor classification approach. Also, different time periods have been highlighted.

Figure 7.2a represents the behavior for a non-cooperative usage predictor approach (as seen in Chapter 6). C_3's TLB misses in period ❶ while the page is predicted not to be used in C_0 and C_2. In response to the TLB miss request, the disused entries are invalidated, even though the page classification remains as shared, since C_1 is still accessing the page. Consequently, C_2 incurs in an extra induced TLB miss afterwards (time instant ❷), evidencing an inaccurate disuse prediction. Yet, reclassification to private is only achieved in period ❸, when a miss occurs in C_0's TLB, by invalidating the disused entry in C_3's TLB.

Figure 7.2b shows a similar example for our cooperative usage predictor. In this case, translations are invalidated only when a reclassification opportunity is revealed, thus avoiding the unnecessary invalidations and the extra TLB miss in period ❶. Then, the cooperative usage predictor unveils an opportunity to reclassify as private in period ❷ beforehand (i.e., without



(a) Non-cooperative usage predictor (UP).

(b) Cooperative usage predictor (CUP).

Figure 7.2: Prediction-based classification examples.

TLB miss reliance), which results successful. Finally, the disused entry is invalidated in period ❸ to help the page to remain as private, just as with a non-cooperative predictor.

7.2 Cooperative Usage Predictor (CUP)

Cooperative Usage Predictor (CUP) is a mechanism designed to work in conjunction with a TLB-based classification approach. The twofold goal of our mechanism is to improve prediction accuracy while avoiding pointless TLB invalidations. To do so, TLB entries gather system-wide page usage information from other cores. This is accomplished through TLB cooperation, by sending notifications as soon as a given page is predicted not to be in use in the near future. Then, only if a shared page is hinted to be currently in use in a single core, disused TLB entries are invalidated through a reclassification process. Nonetheless, system-wide translation invalidation is only performed when it can positively contribute to the classification characterization.

7.2.1 Double Token Set: Implementation Details

We implement our Cooperative Usage Predictor on top of a token-counting TLB-based page classification mechanism (see Chapter 5). Then, CUP can be easily implemented by just adding an extra set of tokens associated with every page. This way, CUP employs two sets of tokens with different interpretations. First, a set of tokens, namely classification tokens (or $tokensC$), performs a page classification based on the token count. Similarly, a second set of tokens, called usage tokens (or $tokensU$), offers a hint for current system-wide page usage in order to reveal reclassification opportunities. In other words, CUP associates each page with a ($\#tokensC$,$\#tokensU$) pair, N tokens per set for a N-core system.

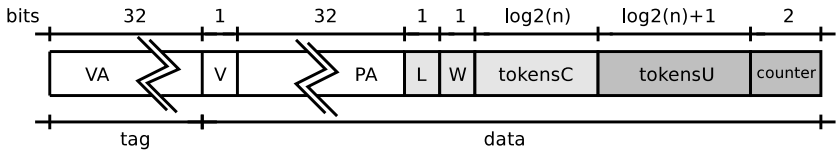


Figure 7.3: TLB entry format for the cooperative usage predictor (extra fields in grey).

Figure 7.3 shows the extra fields that CUP requires in each TLB entry in darker grey. Specifically, our cooperative usage predictor requires an extra $\lceil log_2(N+1) \rceil$-bit field to track from 0 to N usage tokens and keep a system-wide record based on its page sharers usage, and an extra 2-bit counter field in order to determine whether a page is currently being accessed or not (field is saturated) in any given TLB. The counter field is also required in the predictors seen in the previous chapter (Section 6.2), and is increased after a fixed period and reset after every access to the corresponding entry. Therefore, a page falls into disuse after the occurrence of four predictor periods without being accessed. All in all, our double token set strategy requires $3 + \lceil log_2(N+1) \rceil * 2$ total bits. Since the TLB entry data field often contains some unused bits [12] that are reserved, hardware overhead may be avoided by taking advantage of them. In any case, the hardware overhead represents only 13 extra bits per entry for a 16-core CMP, or 25 bits for a 1024-core CMP. As a consequence, the area overhead for our strategy represents ∼15% or ∼25% of the L2 TLB area for 16- and 1024-core CMPs, respectively, according to CACTI3.

CUP basic semantics

The presence of *tokensU* stored in a TLB entry signifies that the core is a potential page sharer. When a translation entry falls into disuse, *tokensU* are transferred to some core that is currently accessing the related page. Nonetheless, a memory request can continue to freely access any valid page translation entry (i.e., with at least 1 *tokenC*) without *tokensU*. Tokens are exchanged (and *tokensU* may be reacquired) through TLB miss responses or non-silent evictions. As in the single token set classification strategy (see Chapter 5), in CUP only TLBs that are holding 2 or more classification tokens are allowed to answer a TLB request (i.e., *token owners*). In this case, usage tokens are not required for the reply. However, if a page-owning TLB is also holding 2 or more *tokensU*, it responds with one *tokenC* and one *tokenU* (1,1) pair to the requesting TLB alongside the page translation, and keeps the remaining tokens.

Furthermore, upon the reception of *non-silent evictions*, token acquisition strategy is straightforwardly adopted for both token sets. Thus, a given TLB must be holding a valid page translation entry in order to store the tokens from an evicting TLB. Additionally, when the eviction message also contains any number of *tokensU*, a TLB entry has to be currently in use (i.e., either holding at least 1 *tokenU* or with an unsaturated prediction counter) in order to receive the incoming tokens.

Consequently, when a given TLB recovers all its *tokensU*, it is very likely the only core currently accessing the page. Thus, if the TLB entry is not in possession of all page's *tokensC* (i.e., page is shared at the moment), a reclassification process starts to try to acquire the remaining *tokensU* so the page may transition to private.

Time-based obsolescence for TLB entries

CUP averts immediate invalidation for disused translations, since TLB misses are no longer considered as reclassification opportunities. In other words, CUP shelves translation invalidation until TLBs hint for a reclassification opportunity. Hence, TLB cooperation is carried out by announcing the disuse condition right after the TLB entry counter fields saturate.

Principle #1: keeping the page as private as long as possible. While a page is private, only one TLB is holding its translation, and thus the disused condition does not need to be revealed. A private TLB entry holds all (N, N) tokens, and the disuse condition is discovered upon the reception of a network TLB request, just as in non-cooperative usage prediction. When a TLB miss is received and the local translation entry is not in use, all (N, N) tokens are sent to the requester alongside the page translation, invalidating the responder TLB entry and favoring the page to remain private.

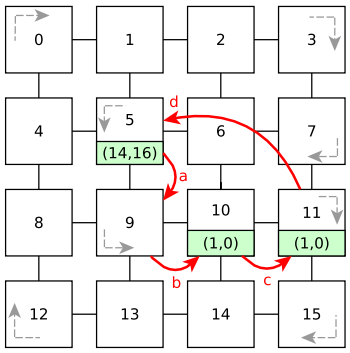
Note that the *forced-sharing* strategy is orthogonal to a cooperative TLB scheme, and can be easily adopted by it. Accordingly, if a *forced-sharing* request is received and the page is private, prediction is overridden (i.e., it does not take into account the counter field) and a (1,1) token pair is sent back to the requester, promoting the page to shared.

Principle #2: cooperating to detect shared-to-private opportunities precisely. Whenever a predictor counter field of a shared page (i.e., $\#tokensC < N$) saturates, the associated TLB entry preserves only a single classification token, giving a total of $(\#tokensC - 1, \#tokensU)$ tokens away, that is, only one *tokenC* is kept. Tokens are never destroyed, but optimistically sent to the network looking for a new holder. The process of giving usage tokens away is called *disuse announcement*, and is sent following the token path depicted in Figure 5.4.

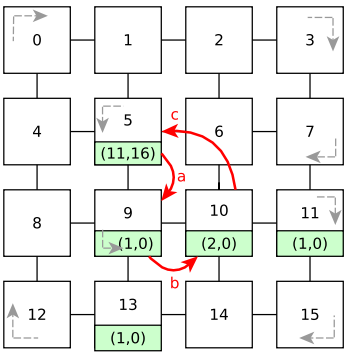
A given TLB must be holding a valid, in-use page translation entry (i.e., with at least one *tokenC* and the 2-bit predictor counter not saturated) in order to acquire the tokens contained by disuse announcements. Unlike eviction messages, acknowledging the token acquisition from a disuse announcement is not required, since the classification is not immediately updated after the announcement message is sent. That is, a valid page that does not hold any *tokensU* is not blocked and can continue being accessed. Naturally, as tokens are given away blindly looking for a new holder, a disuse announcement may traverse the network back to the initial requester if all the TLB page entries of the rest fall into disuse at the same time. In that situation, the source TLB retrieves its own tokens from the network regardless the usage status of the page entry. Thereupon, disuse announcement is turned off, preserving all remaining tokens. Consequently, a shared, disused entry may have stored some *tokensU* that failed to find a new holder. Thus, after receiving the next TLB miss request, the TLB simply responds by sending $(\#tokensC - 1, \#tokensU)$ tokens to the requester TLB.

Cruise-missile reclassification

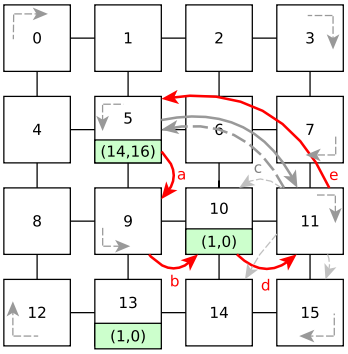
Eventually, a given TLB entry for a shared page will collect all N *tokensU*, suggesting that it is the only TLB currently accessing the page while other TLBs still retain some *tokensC*. As a consequence, a reclassification process is initiated in order to attempt to retrieve all *tokensC*, which would entail the page transition to private. To do so, we employ an exploration technique, namely *cruise-missile reclassification* (CMR), that shares some similarities with *cruise-missile-invalidates* [14]. The cruise-missile reclassification process sends a single message across the virtual ring (the same path followed by evictions and disuse announcements). The CMR message keeps a $(\#tokensC, \#tokensU)$ pair, namely *token pool*, which is initialized to $(0, N - \#tokensC)$. The first component stores the *tokensC* as they are being recovered,



(a) Case #1: Cores 10 and 11 are in disuse when receiving the reclassification. Page is reclassified to private.



(b) Case #2: Core 10 is being used although it gave its *tokensU* away. Reclassification is interrupted.



(c) Case #3: A TLB misses while a reclassification tryout is ongoing. Reclassification is interrupted.

Figure 7.4: Different outcomes for cruise-missile reclassifications initiated by the TLB in core 5. The grey dashed arrows depict the route followed by CMR messages across the virtual ring.

whereas the second represents the maximum number of *tokensC* to be recovered, which remain spread in other TLBs. Usage tokens in the CMR token pool also allow a reaccessed translation entry to reacquire *tokensU* from the CMR message. A CMR message traverses the virtual ring until *tokensC* are fully recovered, and other cores' translation entries are invalidated.

It is important to highlight that memory accesses are not blocked in the initiator core for a page involved in a reclassification process. Classification evolves naturally when a CMR message returns to the initiator. This way, the negative impact on system performance for our cruise-missile strategy due to the path length is avoided. CMR simply delays reclassification, causing some temporarily miss-classified shared accesses to otherwise possibly private data.

Lastly, reclassification rarely needs to completely traverse the ring, and some exploration optimizations may be applied, resulting in three possible outcomes depicted in Figure 7.4 for a 16-core CMP:

Case #1: successful reclassification. On every hop of a CMR message, if a page translation is present, valid, and disused, the translation entry is invalidated, the blocks for that page in the L1 cache are flushed, and all of its *tokensC* are stored in the CMR token pool. Then, as long as $\#tokensC < \#tokensU$ in the token pool, CMR is forwarded to the next TLB in the virtual ring. Conversely, when $\#tokensC = \#tokensU$, reclassification process succeeds (i.e., reclassification recovered all missing *tokensC*) and tokens are straightforwardly sent back to the initiator.

The initial C_5's page sharing status in Figure 7.4a is (14,16) tokens. Thus, a reclassification starts with (0,2) tokens in the CMR message (C_5's TLB keeps (14,14) tokens). CMR message misses on C_9 and is forwarded to the next node in the virtual ring (transitions (a) and (b)). Next, TLB in C_{10} invalidates its disused page translation, storing the single remaining *tokenC* in the CMR token pool. Then, CMR is forwarded to the next node (transition (c)) with (1,2) tokens. TLB entry in C_{11} is disused again, and the last *tokenC* is stored in the CMR. Finally, since the token pool contains the same amount of classification and usage tokens (i.e., (2,2) tokens), CMR is sent back to the initiator (i.e., C_5 in transition (d)). The result is that the page is successfully reclassified to private with (16,16) tokens.

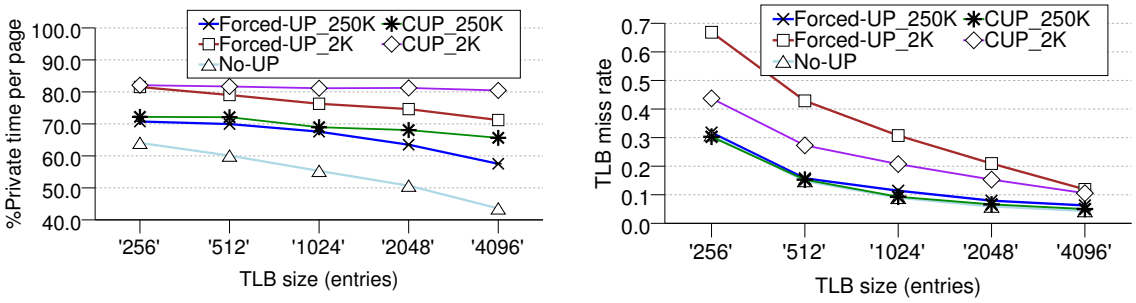
Case #2: page restoration from the CMR token pool. *TokensU* in the CMR token pool allow translation *restoration* (i.e., the TLB counts again as a page sharer). In this sense, when a TLB entry receives a CMR message and has a recently-accessed, valid page translation, it reclaims as many *tokensU* from the token pool as *tokensC* it is currently holding, which *restores* the entry and aborts reclassification. CMR is sent straightforwardly back to the initiator, since reclassification cannot be fulfilled.

Initially, C_5's TLB entry holds (11,16) tokens in Figure 7.4b. Hence, CMR token pool contains (0,5) tokens at the beginning of the reclassification. Since C_9's TLB entry remains disused, the translation is invalidated, and its single classification token is stored in the token pool. Next, (1,5) tokens are forwarded within the CMR message to the next TLB in the virtual ring (transition (b)). However, TLB entry in C_{10} contains 2 *tokensC* and has been recently accessed. Therefore, the translation is *restored* by taking 2 *tokensU* from the CMR token pool. CMR is sent back to the initiator with (1,3) tokens (transition (c)). Finally, the TLB in C_5 annotates the tokens received, collecting a total of (12,14) tokens. As a consequence, the reclassification to private fails. Note how TLBs in other cores (e.g., 11 or 13) are not consulted by the CMR message although they may be in disuse. Restoring the entry in C_{10} disallows reclassification, allowing other cores to retain their tokens.

Case #3: racing TLB miss during reclassification attempt. Finally, reclassification to private is canceled when a CMR message bumps into a TLB which either has recently obtained tokens (e.g., after resolving a TLB miss for the same page), or is currently involved in a TLB miss. Since a reclassification starts with all N *tokensU* stored in the initiator TLB entry, the presence of *tokensU* in a TLB in the route of a CMR message implies that tokens were recently obtained from the initiator TLB, and thus missing *tokensC* will not be able to be fully recovered. As a special case scenario, if a CMR message traverses a TLB which afterwards suffers a miss for the same page, reclassification might return to the initiator TLB either as in case #1 or case #2. Nevertheless, reclassification still fails since some *tokensC* were delivered to the TLB that missed.

Figure 7.4c shows how TLB in C_{11} misses for a page that is involved in a reclassification process. The broadcast TLB request message (dashed arrows) reaches the *page-owning* TLB in C_5, which kept (14,14) tokens after initiating the CMR. In response, C_5's TLB sends (1,1) tokens alongside the page translation to the TLB in C_{11} (transition (c)) to resolve the TLB miss. Then, after having traversed C_9 and C_{10}, the CMR message reaches C_{11} in transition (d), which is currently holding (1,1) tokens. Hence, reclassification is canceled and the CMR is sent back to the initiator with (1,2) tokens (the *tokenC* was recovered from C_{10}). Reclassification fails with (14,15) tokens in C_5's TLB.

Cruise-missile against broadcast reclassification. To sum up the main properties of our cruise-missile reclassification, we compare it against an alternative solution for reclassification based on broadcasts. First, CMR is far more efficient in terms of traffic, as it avoids the require-



(a) Private time proportion.

(b) TLB miss rate.

Figure 7.5: TLB size and usage prediction analysis.

ment for multiple response messages, while allowing many cases where system exploration ends beforehand. Second, contrary to a broadcast solution, CMR early conclusion avoids system-wide invalidation of disused pages after every reclassification tryout. Finally, even though broadcast reclassification may parallelize invalidations and responses, which allows faster transition to private, cruise-missile reclassification does not incur in additional damage to system performance, as accessing the memory hierarchy is permitted during the whole process.

7.3 Experimental Results

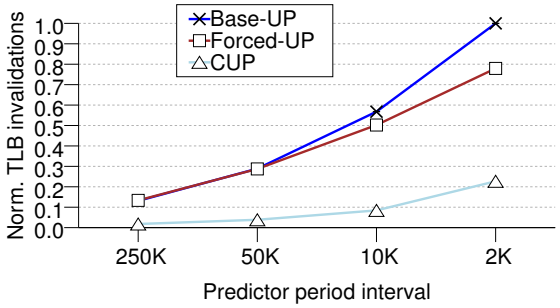
In this section we compare CUP to the usage predictor (UP) with different prediction periods, ranging from 250K to 2K cycles. System configuration is as detailed in Section 3, with private two-level TLBs. First, we present some detailed results to analyze how CUP operates. As an example of the potential benefits of TLB cooperation, we then show how CUP performs compared to *forced-sharing* UP when applied to coherence deactivation.

7.3.1 CUP Impact on TLB Classification

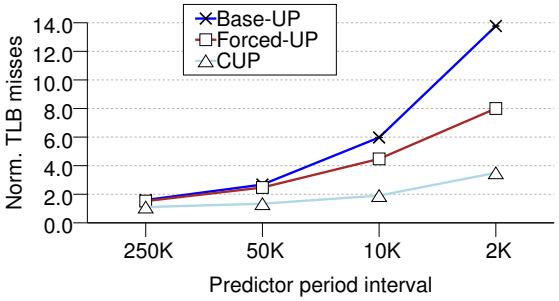
TLB size influence. The main purpose of employing a usage predictor is to make classification accuracy for TLB-based classification mechanisms unaffected by TLB size. To illustrate this, Figure 7.5 shows how classification varies on TLB-based mechanisms with different TLB sizes (associativity is kept invariable).

On the one hand, the average percentage of cycles per page spent as private is shown in Figure 7.5a. As expected, *No-UP* gradually diminishes the private time proportion per page as the TLB size increases. Similarly, although usage prediction increases the time spent as private, *Forced-UP* is still affected by TLB size variations, progressively losing accuracy with larger TLBs. Conversely, *CUP* remains invariable, keeping pages 22% more time as private on average with *CUP_250K*, and up to 36.8% with *CUP_2K*, both compared to *No-UP* with the largest TLB analyzed. Also, *CUP_250K* spends 8.1% more cycles as private compared to *UP_250K*.

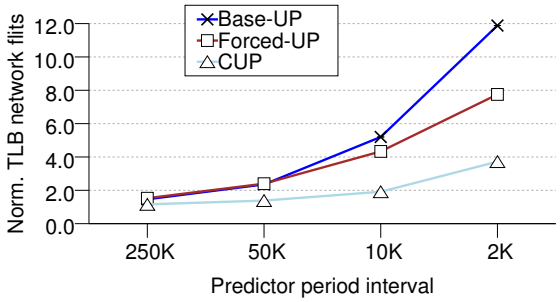
On the other hand, Figure 7.5b shows how TLB miss rate evolves for different TLB sizes. As can be expected, applying usage prediction techniques comes with a cost, since it entails increasing the TLB miss rate as the predictor period is reduced. Moreover, the larger the TLB size, the lower the TLB miss rate. Surprisingly, TLB miss rate reduction evolves faster with *Forced-UP*, since many positive feedback TLB invalidations are naturally and gradually prevented



(a) Predictor-induced TLB invalidations normalized to *Base-UP_2K*.



(b) Number of TLB misses normalized to non-prediction.



(c) TLB network flits injected normalized to non-prediction.

Figure 7.6: Miss-prediction overhead analysis: UP against CUP.

only by increasing TLB size. Unfortunately, reducing the TLB miss rate potentially reduces private page detection for *Forced-UP*. Per contra, classification accuracy loss can be prevented through TLB cooperation, as shown in Figure 7.5a, since reclassification opportunities are unveiled without TLB miss reliance.

Prediction overheads analysis. CUP aspiration and one of its main contributions is to smartly invalidate TLB entries only when a page may effectively transition to private, dramatically reducing the amount of predictor-induced invalidations and their consequences (e.g., miss-predicted invalidations), as seen in Figure 7.6a (normalized to *Base-UP* with 2K period). *Forced-UP* lessens the amount of predictor-induced invalidations by detecting positive feedback situations, as much as 22% reduction for the 2K predictor period compared to *Base-UP*. Unlike previous predictors, CUP nearly flattens the proportion of extra invalidations for all configurations. However, the most aggressive predictor period considered in the study still induces some additional invalidations, which degrades prediction accuracy to favor slightly better classification accuracy. Yet 4 out of 5 invalidations are still avoided compared to *Base-UP*.

Therefore, eliminating most miss-predicted invalidations in the TLB implies reducing the amount of predictor-induced TLB misses, as shown in Figure 7.6b. TLB misses in the figure are normalized to a non-predicting token-counting classification. Specifically, *CUP* halves the amount of TLB misses compared to *Forced-UP*, particularly on smaller predictor periods.

Finally, the network TLB traffic issued to support prediction-based classification is dramatically increased as a consequence of predictor-induced TLB misses, since the mechanism tries to obtain tokens through network exploration. However, Figure 7.6c shows how the total TLB traffic increase is tackled by TLB cooperation, halving it compared to *Forced-UP*, and amply offsetting the extra TLB traffic issued by CMR messages.

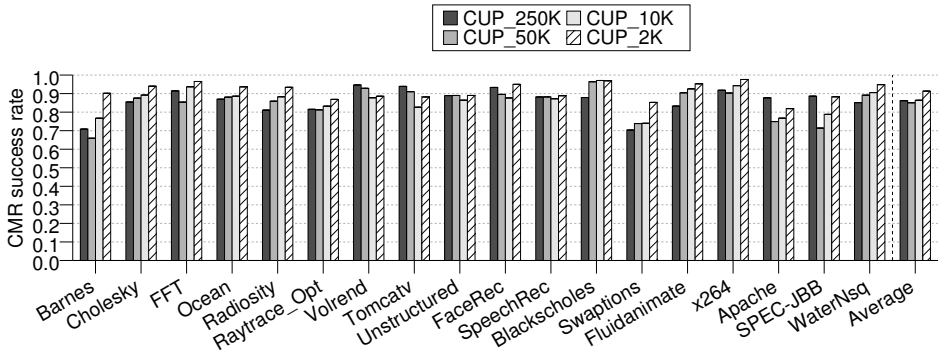


Figure 7.7: CMR tryouts that successfully transition the page to private.

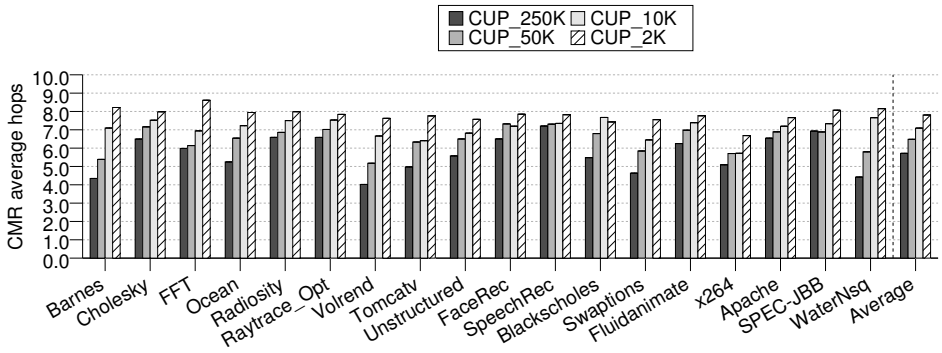


Figure 7.8: Average number of steps for CMR messages.

Cruise-Missile Reclassification analysis. System-wide usage prediction on CUP should be as accurate as possible to avoid premature invalidations. In this sense, Figure 7.7 shows the proportion of reclassification tryouts that successfully end up transitioning to private. Specifically, for any predictor period in the evaluation, the success rate for page reclassification to private through CMR messages is greater than 85%, proving the accuracy of our cooperative predictor.

A cruise-missile message progresses node by node through the network until reclassification either succeeds or is aborted. Hence, CMR messages are not required to visit all system nodes in the common case. Figure 7.8 shows the number of average hops that a CMR message has to perform to finish a reclassification process. Regardless of the predictor period employed, a CMR message requires less than 8 hops in average in order to conclude a reclassification process. Thus, just half the maximum number of hops are taken for the 16-core CMP system considered in the evaluation. In the case of a 250,000 cycles predictor period, a CMR tryout is completed after just 5.7 hops in average.

Reclassification frequency is indicative of prediction aggressiveness. Unlike in a non-cooperative predictor, where each and every TLB miss is seen as an opportunity for reclassification, CUP smartly invalidates TLB entries by sending a CMR message when a reclassification opportunity arises. Therefore, for comparison purposes, Figure 7.9 depicts the amount of CMR tryouts per TLB miss of a *forced-sharing* UP. Particularly, CUP reclassification rates for 250K, 50K, 10K, and 2K predictor periods are 0.03, 0.045, 0.065, and 0.09 tryouts per TLB miss, respectively.

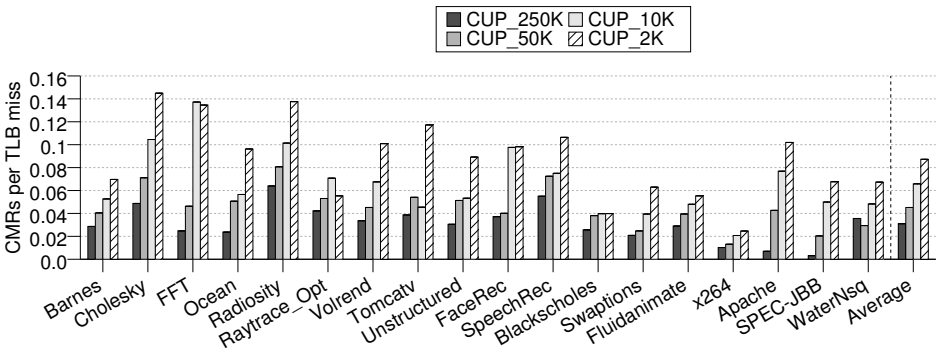


Figure 7.9: CMR messages issued per TLB miss.

Conclusion. CUP improves the time spent as private per page by 22% over a non-predicting classification, effectively making classification characterization independent from TLB size. Also, CUP constrains TLB traffic to a third, compared to a non-cooperative predictor, as it avoids 4 out of 5 aimlessly induced TLB invalidations. Efficient CMR messages are sent exclusively when a reclassification opportunity is detected, with over 85% success rate in private reclassification, and less than 8 nodes visited per tryout.

7.3.2 Coherence Deactivation

We chose coherence deactivation as our study case to test the benefits of the proposed data classification scheme.

Coherence deactivation avoids some L1 cache *Coverage* misses induced by directory evictions. However, a TLB-based classification mechanism induces some extra cache misses as a consequence of conflicted TLB entries and TLB-cache inclusion policy. Additionally, employing a usage predictor for TLBs may increase the amount of blocks forcefully flushed from the cache due to TLB invalidations and recoveries, and ultimately neglect its potential benefits. Figure 7.10 depicts the average L1 cache misses proportion caused due to the TLB-cache inclusion policy. Expressly, it shows how *CUP* induces just up to an extra 5% of L1 cache misses for the smallest predictor period, whereas *Forced-UP* gets up to 18.4% extra misses for the same period value, due to frequent, brute-force, system-wide invalidations (and consequent cache flushes) after every TLB miss.

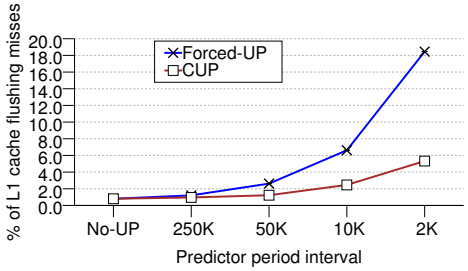


Figure 7.10: Proportion of L1 cache flushing misses.

Figure 7.11 depicts execution time for token-counting classification mechanisms, with and without usage prediction, all normalized to a baseline without coherence deactivation nor

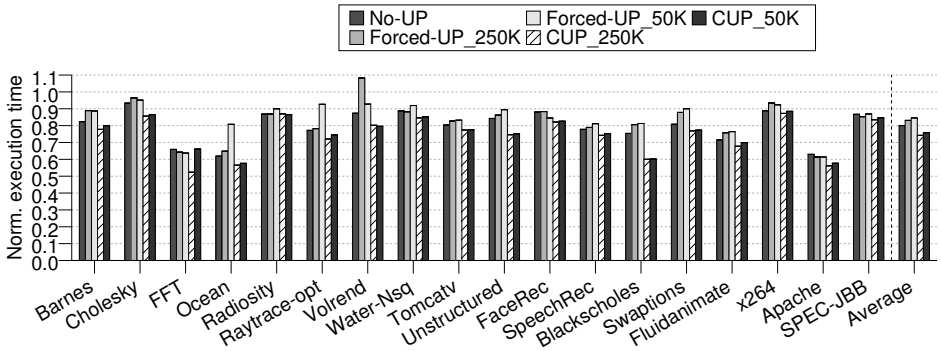


Figure 7.11: Execution time normalized to baseline without coherence deactivation.

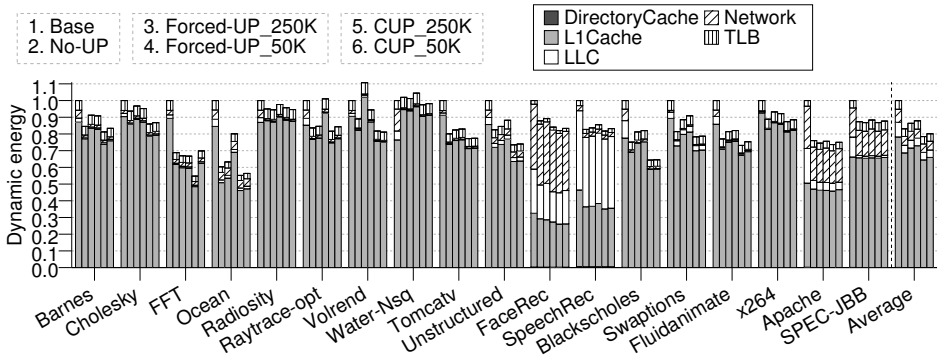


Figure 7.12: Normalized dynamic energy consumption in the cache hierarchy.

TLB-to-TLB transfers. Note that 10K and 2K predictor periods are not displayed for the sake of clarity. Although they reduce directory usage to a greater extent, inasmuch as TLB and cache miss rates are increased by 3.6% and 2.0%, respectively, its application may be discouraged. Particularly, deactivating coherence improves execution time by 20% with *No-UP*. Usage predictors introduced in the previous chapters (e.g., *Forced-UP*) do not suffice to overcome its own overheads over the benefits obtained from a better classification when applied to coherence deactivation. In fact, blindly invalidating translation entries gradually damages system performance compared to *No-UP*, by 3.1% and 4.7% for *Forced-UP_250K* and *Forced-UP_50K*, respectively. On the contrary, CUP squeezes coherence deactivation to its maximum while mostly eliminating prediction overheads, smartly invalidating translation entries through precise system-wide predictions. Altogether, CUP positively impacts execution time, not only avoiding performance penalization, but also improving execution time with *CUP_250K* by 5.8% compared to *No-UP*, 8.8% over *Forced-UP_250K*, and up to 25.8% over baseline.

Furthermore, improving classification characterization and reducing prediction overheads has also a direct impact in the cache hierarchy energy consumption. Figure 7.12 shows how consumption is reduced by 16.9% with *No-UP* compared to *Base*, since coherence deactivation contributes to reduce the L1 cache miss rate. Nonetheless, *Forced-UP* increases the consumption over *No-UP*, by 3.3% and 4.8% for 250K and 50K predictor periods, respectively, due to the burden of extra induced TLB and cache misses. Conversely, *CUP_250K* reduces dynamic consumption by 4.7% over *No-UP*, and 8.0% with respect to *Forced-UP*, which shows the goodness of the proposed cooperative prediction mechanism.

7.4 Discussion

This section discuses some design choices and architectural alternatives, and how they may affect to our prediction-based classification approach.

7.4.1 Token Dependence

Although this chapter proposes a double token set strategy to achieve cooperative TLBs, it is just a design choice. The key remark is that TLB entry invalidation must service a better classification, otherwise it must be averted to prevent performance shrinkage.

Thus, our proposed double token set implementation is independent from the key remark for TLB cooperation. Usage tokens emulate a distributed system-wide count for potential sharers, which could be centralized, using either a directory-like structure or a shared last level TLB structure (e.g., SUP). Also, CUP may operate on top of any TLB-based classification approach, eschewing the need for classification tokens (e.g., SnoopingTLB). However, analysis in Chapter 6 proved that TokenTLB grants full-adaptivity, representing a more accurate and scalable classification solution. Conclusively, double token set strategy offers an elegant general basis for an accurate and efficient classification scheme valid for any TLB organization, size, and/or associativity.

7.4.2 Virtual Caches

As discussed in Chapter 4, TLB-based classification is completely applicable to virtually indexed, virtually tagged (VIVT) caches (also known as virtual caches). However, as memory accesses hit on the caches without accessing the TLB, usage prediction might not be applicable (neither for UP nor for CUP). However, the problem could be solved by moving the predictor to the cache (as in Cache Decay [40]) and periodically probe the presence of blocks from the TLB. Similarly, the number of blocks in the cache per page stored in the TLB can be tracked, and explicit notification may be sent after a block is cached or falls into disuse. When the blocks from a page in the cache have all been evicted or have fallen into disuse, the TLB entry can be invalidated.

7.5 Conclusions

In this chapter Cooperative Usage Predictor (CUP) has been proposed, a prediction mechanism designed to reduce the generation time of page translations in TLBs according to its usage in order to service a more accurate private classification. CUP is completely independent from TLB size, eagerly unveils private pages, and smartly invalidates TLB entries only when an opportunity for reclassification to private is detected. To do that, we introduce a double token set strategy. One of the sets is used for page classification and the other set represents the page usage throughout the system. Employing *disuse announcements*, where usage tokens are released, the opportunity for reclassification is promptly discovered. Then, our proposed cruise-missile approach allows an efficient reclassification to private at a minimum cost in terms of performance and traffic. As a consequence, CUP enables optimizations for private accesses with minimum overhead. All in all, unlike the approaches in Chapter 6, CUP manages to make usage prediction appealing, promoting classification accuracy while effectively improving system performance for TLB configuration employed in commodity systems.

Chapter 8

Conclusions

In this chapter we summarize the contributions and conclusions of the work carried out, the publications and the future work.

8.1 Contributions and Conclusions

Classifying memory accesses into private or shared data has become a fundamental approach to achieve efficiency and scalability in multi- and many-core systems. Since most memory accesses in both sequential and parallel applications are either private (accessed only by one core) or read-only (not written) data, devoting the full cost of coherence to every memory access results in sub-optimal performance and limits the scalability and efficiency of the multiprocessor.

Ideally, the classification performed by any mechanism should be as accurate as possible, and with low-overhead in terms of traffic, performance, and area. However, data classification usually requires a large amount of extra storage in order to track the data sharing status. Furthermore, some mechanisms cannot dynamically reclassify data from shared to private, thus missing the detection of temporarily-private data and limiting the potential performance benefits of data classification. Finally, other classification schemes are not applicable to all data optimizations or applications, involving some strict requirements to be fulfilled in order to become a suitable choice.

The aim of this dissertation has been the design of a novel family of TLB-based classification approaches able to cope with all the desirable properties for data classification, achieving an adaptive classification that accounts for temporarily-private pages and tolerating thread migration, with low storage requirements and fairly good scalability. A TLB-based classification mechanism is based on querying other TLBs' about the presence of page translations on every TLB miss. This way, a TLB knows whether the page is shared or not when the miss is resolved.

A part of this thesis is dedicated to describe and thoroughly analyze the new classification strategies proposed, improving the classification accuracy and achieving new goals and properties at the same pace as the dissertation has been progressing. The main contributions of this dissertation are:

- Snooping TLB-based classification (SnoopingTLB –Chapter 4). This mechanism represents the starting point of this dissertation. SnoopingTLB offers a runtime, adaptive classification based on inquiring the other cores' TLBs in the system (through TLB-to-TLB requests) on every TLB miss. The other TLBs reply to the requester indicating whether or not they are caching the page translation.

- TLB-based classification with shared last-level TLB (DirectoryTLB –Chapter 4). This mechanism leverages a shared last-level TLB in a NUCA-like structure in order to track the current page sharers and resolve TLB misses in the upper private TLB level, similar to an in-cache directory for coherence maintenance.

- Token-based classification (TokenTLB –Chapter 5). This mechanism is based on token counting in order to determine the current classification status. Tokens are stored in TLB entries and exchanged through TLB-to-TLB communication, allowing the classification to naturally evolve to and from shared.

- Usage Predictor for TLBs (UP –Chapter 6). This is the basic predictor from which this thesis started. Relying on the presence of page translation in system TLBs for classification leads to a strong TLB size dependence for classification accuracy. A usage prediction mechanism for TLBs is essential to avoid this dependence. When a TLB entry is predicted not to be used and is probed with a TLB-to-TLB request, the entry is invalidated, greatly increasing the opportunity for private data detection.

- Forced-Sharing prediction strategy (Chapter 6). Lower predictor periods induce miss-predicted TLB invalidation, causing more TLB misses, which in turn induces further TLB invalidations. This positive feedback is avoided by the forced-sharing strategy, where prediction is overriden when a premature invalidation of a TLB entry is detected.

- Shared Usage Predictor (SUP –Chapter 6). A shared TLB level acts as a filter for miss-predicted invalidations, since TLB entries are only invalidated by probing the upper level TLBs when the shared TLB detects a reclassification opportunity.

- Cooperative Usage Predictor (CUP –Chapter 7). This mechanism exploits TLB cooperation (i.e., TLBs willingly working together for a common purpose or benefit) in order to perform a system-wide page usage prediction in private TLB structure. To do that, a dual token set strategy is employed, for classification and page usage along system, respectively.

The family of TLB-based mechanisms proposed for classification share some similarities with cache coherence strategies, and hence their names. However, since translation entries are not commonly modified directly in the TLB, the classification approaches require simpler design considerations than their coherence counterparts. Furthermore, on read misses in the TLB, TLB-based classification accelerates the translation process by detecting inter-core sharing patterns. We put a lot of effort in this thesis in order to measure the positive impact on system performance, comparing our results against previous classification approaches. Also, we tried to measure its overheads, specially in terms of area and network consumption. Some important conclusions can be extracted from the evaluation of the proposed mechanisms.

The *first conclusion* is that TLB-based classification, unlike other classification approaches, significantly improves system performance regardless of the classification-based data optimization employed (e.g., Coherence deactivation, VIPS, Reactive-NUCA, etc.), as it accelerates the page translation process. This improvement relies on the application's sharing degree and the size of the TLB structure. Specifically, system performance is improved up to 26.4% with

64-set TLBs, according to our experiments. Consequently, inasmuch as other possible overheads (e.g., network traffic, additional cache or TLB misses, etc.) are minimized, TLB-based classification could be an attractive alternative for any classification-based data optimization over other classification approaches.

Concerning the classification overheads, we have showed how the amount of traffic attributable to the TLB classification protocol is not as harmful as it might seem, since TLB miss rate is usually very low. TLB traffic hardly represents up to 10% of all network traffic issued in some applications and configurations, or 5.2% on average with SnoopingTLB.

However, relying entirely on broadcasts on larger systems might become too expensive. In order to reduce traffic pressure we proposed DirectoryTLB and TokenTLB. On the one hand, DirectoryTLB leverages a shared last-level TLB, which is in charge of storing the sharing information. On TLB misses, the NUCA-like home L2 TLB slice is accessed. This way, broadcasts are completely avoided, while inter-core sharing patterns are still exploited to accelerate the page translation process. Therefore, DirectoryTLB represents by far the most scalable approach in this thesis, while getting similar figures for private data detected to those of SnoopingTLB. Nonetheless, a shared last-level TLB is not a common design choice nowadays.

On the other hand, TokenTLB is conceived for a more common private TLB structure, halving the traffic attributable to the TLB by restricting TLB responses and translation replication upon TLB misses. Furthermore, TokenTLB greatly boosts the precision of the classification approach, which hints for the next conclusion.

The *second conclusion* is that adaptivity (i.e., the mechanism property to identify shared-to-private transitions) is a decisive factor for the accuracy of the classification mechanism, and hence, for the system performance and the benefits obtained from any classification-based data optimization. We claim that an OS-based mechanism is non-adaptive. While this is true, when a page table entry is evicted, classification information is lost and it might be brought back again as private, possibly resulting in a shared-to-private reclassification. However, an OS-based classification approach does not have any control whatsoever over these transitions. They just occasionally take place mainly due to competition on the physical space in main memory, which is not a frequent case.

Conversely, SnoopingTLB is an adaptive mechanism, capable of reclassifying the page as private from one global page generation to the next. Although technically, SnoopingTLB does not have control over reclassification, as opportunities for reclassification require the page to be evicted from a TLB and then check the classification status again afterwards. We still claim that SnoopingTLB is an adaptive mechanism, since the generation time of a page in a TLB is far more dynamic than a page table entry in main memory, and reclassifications happen more often. As a consequence, SnoopingTLB detects 61.8% of all accessed pages as private (nearly 18% more than an OS-based mechanism), and is able to reclassify 13.4% of all shared pages to private at least once.

In this thesis, the mechanism that really highlights the importance of adaptivity above other classification schemes is TokenTLB. Relying on token exchange, TokenTLB naturally reclassifies a page from shared to private when a TLB entry recovers all of the system page tokens. Thus, page classification naturally evolves while the page translation remains stored in the TLB. We refer to this property as *full-adaptivity*, and TokenTLB is able to classify nearly 41% more blocks as private (which depends on the classification of the page when the access misses in the L1 cache) than SnoopingTLB, consequently improving execution time and dynamic energy consumption by 20% and 21.9%, respectively, over our baseline system when we devote the classification scheme to coherence deactivation.

Finally, the *third conclusion* is that a usage predictor for TLBs is vital in order to increase the precision of a TLB-based classification approach with larger or deeper TLBs, since it allows a classification accuracy less dependent on the size or associativity of the TLB structure. A predictor invalidates those TLB entries that are not being used when a TLB request is received, thus creating new opportunities for private classification. This way, the classification is determined by the page access pattern (i.e., page usage) and the disuse timeout period (i.e., the expire time from the last access for a page to fall into disuse) instead of depending on the page generation time.

A Usage Predictor (UP) improves the amount of private data detected from 63.4% with TokenTLB to up to 77.4% with UP. However, a poor prediction accuracy with UP may entail some major overheads. A miss-predicted disuse period may lead to an invalidation of a page that is actually being accessed, which will induce a predictor-induced TLB miss afterwards. Furthermore, this additional misses may induce further premature invalidations, which ultimately produce more predictor-induced misses in the TLB structure. These positive feedback invalidations cause a deterioration in system performance, and dramatically increase the TLB traffic produced when the classification mechanism tries to retrieve the sharing status after every TLB miss.

Some mechanisms have been proposed in order to restrain or completely avoid these positive feedback invalidations. Forced-sharing strategy is a slight modification of the state machine. When a TLB misses on a page that has been recently invalidated (the translation entry is still present), a special miss request is sent, overriding any disuse prediction, and avoiding further invalidations. This strategy avoids the predictor attempt to bring the page classification back to private, and hence its name. Although with Forced-sharing UP the traffic increase is reduced by 24%, and performance loss is partially prevented, these overheads still represent a major issue and might prevent the appliance of prediction techniques for TLB-based classification schemes.

Notwithstanding, the two main proposed alternatives for UP are SUP and CUP. SUP is a predictor strategy for systems with a shared last-level TLB (DirectoryTLB). The main feature of SUP is that, unlike UP, disuse periods are announced to the shared TLB instead of waiting for another TLB to miss and invalidate the page entry. Moreover, invalidation occurs only when the shared TLB detects a transition to private, which induces a reclassification process. This way, the shared TLB acts as a filter for premature invalidations. SUP completely avoids prediction overheads, improving the execution time by 2% over DirectoryTLB (8.2% over baseline) when classification is applied to coherence deactivation.

However, SUP relies again in a shared last-level TLB. Following a similar strategy in a purely-private TLB structure, we proposed CUP, a Cooperative Usage Predictor. CUP inherits TokenTLB full-adaptivity and read-only detection, turning out to be the mechanism with the better classification accuracy in this thesis, while avoiding most drawbacks. CUP halves the amount of TLB misses and network traffic issued compared to UP with Forced-sharing strategy. This way, CUP improves execution time by 5.8% over TokenTLB and 8.8% over UP with Forced-sharing when applied to coherence deactivation. Consequently, CUP proved how prediction might become invaluable for future, deeper, private TLB structures by achieving high accuracy standards on both classification and prediction.

In summary, in this thesis we have proposed a new family of TLB-based classification protocols, representing a new attractive alternative for classification-based data optimizations by tackling all desirable properties for classification. Furthermore, TLB-based classification approaches do not involve additional hardware resources. They only require some additional fields in the TLB, thus incurring in low area overhead. Our techniques achieve a precise, runtime, adaptive,

page-level classification that is based on page-access patterns and access predictions, and can detect read-only data regions, while removing most of its overheads. Since scaling to larger systems might increase end-to-end latency or network consumption (ultimately making TLB-based classification unattractive), the classification schemes in this thesis aim for small- and medium-scale CMPs.

8.2 Scientific Publications

Next, we list the articles published in relation to the work presented in this thesis.

Publications on National Conferences:

- **"Temporal-Aware TLB-Based Private Page Classification in CMPs"**. A. Esteve, A. Ros, M.E. Gómez, and A. Robles. In procceding of the XXVI Jornadas de paralelismo, pages 14-23, Córdoba (Spain), September 2015, Publisher: Pedro del Río Obejo (Copisterías Don Folio, S.L.). ISBN: 978-84-16017-52-2.

- **"Mecanismo de clasificación de páginas basado en el paso de tokens entre TLBs"**. A. Esteve, A. Ros, A. Robles, and M.E. Gómez. In procceding of the XXVII Jornadas de paralelismo. Jornadas SARTECO 2016, pages 471-480, Salamanca (Spain), September 2016, Publisher: Ediciones Universidad de Salamanca. ISBN: 978-84-9012-626-4.

- **"CUP: Un Predictor de Uso de Páginas basado en Tokens"**. A. Esteve, A. Ros, A. Robles, and M.E. Gómez. Accepted in the XXVIII Jornadas de Paralelismo (JP2017), Málaga (Spain), September 2017.

Publications on International Conferences:

- **"TokenTLB: A Token-Based Page Classification Approach"**. A. Esteve, A. Ros, M.E. Gómez, A. Robles, and J. Duato. In procceding of the 2016 International Conference on Supercomputing (ICS), Article No. 26, Istambul (Turkey), June 2016, Publisher: ACM Digital Library. ISBN: 978-1-4503-4361-9.

Publications on International Journals:

- **"Efficient TLB-Based Detection of Private Pages in Chip Multiprocessors"**. A. Esteve, A. Ros, A. Robles, M.E. Gómez, and J. Duato. IEEE Transactions on Parallel and Distributed Systems (TPDS), pages: 748-761, March 2015, Publisher: IEEE. ISSN: 1045-9219.

- **"TLB-Based Temporality-Aware Classification in CMPs with Multilevel TLBs"**. A. Esteve, A. Ros, A. Robles, M.E. Gómez, and J. Duato. IEEE Transactions on Parallel and Distributed Systems (TPDS), pages: 2401-2413, August 2017, Publisher: IEEE. ISSN: 1045-9219.

- **"TokenTLB+CUP: A Token-Based Page Classification with Cooperative Usage Prediction"**. A. Esteve, A. Ros, A. Robles, and M.E. Gómez. Submitted to IEEE Transactions on Parallel and Distributed Systems (TPDS), February 2017.

8.3 Future Work

There are several research directions that can be taken out of this dissertation. This is a brief list of possible efforts for the future:

- Extend TLB-based classification for alternate cache designs, as hierarchical caches, where cores are grouped in clusters in order to reduce network congestion by localizing traffic among different hierarchical levels. Furthermore, prediction-based classification needs to be reworked to make it compatible with VIVT caches.

- Study the potential benefits, and recognize the main challenges of implementing the proposed TLB-classification schemes in future heterogeneous systems, where on-chip GPUs and CPUs may share virtual space. This might lead to study environments with frequent thread migration, which would help to validate the properties of our classification scheme against other runtime classification proposals, and compare our solution against other techniques dealing with passive data sharing (e.g., PSCR).

- Employ an adaptive predictor period per TLB entry in order to accurately adjust the predictor period to the actual page inter-access period. Even though many predictor periods have been explored in this dissertation, a given period can be valuable for some applications, but detrimental for others. Even for the same application, it may differ for different pages or application phases.

- Analyze the effect of TLB-based classification on different data optimizations. Different performance and power consumption results would arise.

- Propose and analyze designs for different classification granularities, i.e., larger pages, and, specially, sub-page granularities. Finer granularity can be considered another desirable property for classification approaches.

- Study the effect of a static-dynamic classification, where read-only data is managed by a compiler-based classification mechanism and privacy is detected by our TLB-based classification. This way, the overheads of read-only detection at runtime can be avoided.

Bibliography

[1] Advanced Micro Devices. `http://www.amd.com`. [Online; accessed Jan-2016].

[2] ARM holdings plc. `http://www.arm.com`. [Online; accessed Jan-2016].

[3] Intel Corporation. `http://www.intel.com`. [Online; accessed Jan-2016].

[4] Sun Microsistems. `http://www.sun.com`. [Online; accessed Nov-2015].

[5] *ARM Architecture Reference Manual ARMv7-A and ARMv7-R*, 2012.

[6] *ARM Architecture Reference Manual ARMv8-A*, 2015.

[7] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner prediction for accelerating cache-to-cache transfer misses in cc-NUMA multiprocessors. In *ACM/IEEE Conf. on Supercomputing (SC)*, pages 1–12, Nov. 2002.

[8] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42, Apr. 2009.

[9] N. Agarwal, L.-S. Peh, and N. K. Jha. In-Network Snoop Ordering (INSO): Snoopy coherence on unordered interconnects. In *15th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 67–78, Feb. 2009.

[10] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *5th Workshop On Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 30–38, Feb. 2002.

[11] M. Alisafaee. Spatiotemporal coherence tracking. In *45th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 341–350, Dec. 2012.

[12] AMD. AMD64 architecture programmer's manual volume 2: System programming. `https://goo.gl/8E97El`. [Online; accessed 6-Oct-2016].

[13] T. W. Barr, A. L. Cox, and S. Rixner. Spectlb: A mechanism for speculative address translation. In *38th Int'l Symp. on Computer Architecture (ISCA)*, pages 307–318, June 2011.

[14] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *27th Int'l Symp. on Computer Architecture (ISCA)*, pages 12–14, June 2000.

[15] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared last-level tlbs for chip multiprocessors. In *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 62–73, Feb. 2011.

[16] A. Bhattacharjee and M. Martonosi. Inter-core cooperative tlb for chip multiprocessors. In *15th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 359–370, Mar. 2010.

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, Oct. 2008.

[18] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[19] J. F. Cantin, J. E. Smith, M. H. Lipasti, A. Moshovos, and B. Falsafi. Coarse-grain coherence tracking: RegionScout and region coherence arrays. *IEEE Micro*, 26(1):70–79, Jan. 2006.

[20] J. B. Chen, A. Borg, and N. P. Jouppi. A simulation based study of tlb performance. In *19th Int'l Symp. on Computer Architecture (ISCA)*, pages 114–123, May 1992.

[21] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *39th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 455–465, Dec. 2006.

[22] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *38th Int'l Symp. on Computer Architecture (ISCA)*, pages 93–103, June 2011.

[23] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by avoiding the tracking of non-coherent memory blocks. *IEEE Transactions on Computers (TC)*, 62(3):482–495, Mar. 2013.

[24] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras. The effects of granularity and adaptivity on private/shared classification for coherence. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(3):26:1–26:21, Aug. 2015.

[25] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras. An efficient, self-contained, on-chip, directory: DIR_1-SISD. In *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 317–330, Oct. 2015.

[26] L. Fang, P. Liu, Q. Hu, M. C. Huang, and G. Jiang. Building expressive, area-efficient coherence directories. In *22nd Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 299–308, Sept. 2013.

[27] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: A scalable directory for many-core systems. In *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 169–180, Feb. 2011.

[28] P. Foglia, R. Giorgi, and C. A. Prete. Reducing coherence overhead and boosting performance of high-end smp multiprocessors running a dss workload. *J. Parallel Distrib. Comput.*, 65(3):289–306, Mar. 2005.

[29] A. García-Guirado, R. F. Pascual, and J. M. García. Icci: In-cache coherence information. *IEEE Trans. Computers*, 64:995–1014, 2015.

[30] R. Giorgi and C. A. Prete. Pscr: A coherence protocol for eliminating passive sharing in shared-bus shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):742–763, July 1999.

[31] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *36th Int'l Symp. on Computer Architecture (ISCA)*, pages 184–195, June 2009.

[32] E. Herrero, J. González, and R. Canal. Elastic cooperative caching: An autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 419–428, 2010.

[33] H. Hossain, S. Dwarkadas, and M. C. Huang. POPS: Coherence protocol optimization for both private and shared data. In *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–55, Oct. 2011.

[34] Intel Xeon Phi Coprocessor. `http://software.intel.com/en-us/mic-developer`, Apr. 2013.

[35] A. Jimborean, J. Waern, P. Ekemark, S. Kaxiras, and A. Ros. Automatic detection of extended data-race-free regions. In *International Symposium on Code Generationand Optimization (CGO)*, pages 14–26, Austin, TX (USA), Feb. 2017. IEEE Computer Society.

[36] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam. Reducing data tlb power via compiler-directed address generation. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 26(2):312–324, Feb. 2007.

[37] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Design, Automation, and Test in Europe (DATE*, pages 423–428, Apr. 2009.

[38] Kalray Inc. MPPA2-256. `https://goo.gl/jJ7wlf`, 2012.

[39] M. Kambadur, K. Tang, and M. A. Kim. Harmony: Collection and analysis of parallel block vectors. In *39th Int'l Symp. on Computer Architecture (ISCA)*, pages 452–463, June 2012.

[40] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *28th Int'l Symp. on Computer Architecture (ISCA)*, pages 240–251, June 2001.

[41] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 211–222, Oct. 2002.

[42] C. Kim, D. Burger, and S. W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. *IEEE Micro*, 23(6):99–107, Nov. 2003.

[43] D. Kim, J. Ahn, J. Kim, and J. Huh. Subspace snooping: Filtering snoops with operating system support. In *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, Sept. 2010.

[44] D. Kim, H. Kim, and J. Huh. Virtual snooping: Filtering snoops in virtualized multi-cores. In *43rd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 459–470, Dec. 2010.

[45] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *Int'l Symp. on Workload Characterization (IISWC)*, pages 34–45, Oct. 2005.

[46] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones. Compiler-assisted data distribution for chip multiprocessors. In *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 501–512, Sept. 2010.

[47] Y. Li, R. Melhem, and A. K. Jones. PS-TLB: Leveraging page classification information for fast, scalable and efficient translation for future cmps. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):28:1–25:21, Jan. 2013.

[48] Y. Li, R. G. Melhem, and A. K. Jones. Practically private: Enabling high performance cmps through compiler-assisted data classification. In *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 231–240, Sept. 2012.

[49] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 176–185, Feb. 2004.

[50] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[51] M. M. Martin. *Token Coherence*. PhD thesis, University of Wisconsin-Madison, Dec. 2003.

[52] M. M. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *30th Int'l Symp. on Computer Architecture (ISCA)*, pages 206–217, June 2003.

[53] M. M. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *30th Int'l Symp. on Computer Architecture (ISCA)*, pages 182–193, June 2003.

[54] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, Sept. 2005.

[55] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. Martin, and D. A. Wood. Improving multiple-CMP systems using token coherence. In *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 328–339, Feb. 2005.

[56] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *18th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*.

[57] A. Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. In *32nd Int'l Symp. on Computer Architecture (ISCA)*, pages 234–245, June 2005.

[58] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos. Prediction-based superpage-friendly tlb designs. In *21th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 210–222, Feb. 2015.

[59] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos. Prediction-based superpage-friendly tlb designs. In *21th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2015.

[60] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. Increasing tlb reach by exploiting clustering in page translations. In *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 558–567, Feb. 2014.

[61] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. Colt: Coalesced large-reach tlbs. In *45th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 258–269, Dec. 2012.

[62] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. SWEL: Hardware cache coherence protocols to map shared data onto shared caches. In *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 465–476, Sept. 2010.

[63] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy. UNified instruction/translation/data (UNITD) coherence: One protocol to rule them all. In *16th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 1–12, Feb. 2010.

[64] A. Ros, M. E. Acacio, and J. M. García. A novel lightweight directory architecture for scalable shared-memory multiprocessors. In *11th Int'l Euro-Par Conference*, pages 582–591, Aug. 2005.

[65] A. Ros, M. E. Acacio, and J. M. García. DiCo-CMP: Efficient cache coherency in tiled CMP architectures. In *22nd Int'l Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–11, Apr. 2008.

[66] A. Ros, M. E. Acacio, and J. M. García. Dealing with traffic-area trade-off in direct coherence protocols for many-core cmps. In *8th Int'l Conf. on Advanced Parallel Processing Technologies (APPT)*, pages 11–27, Aug. 2009.

[67] A. Ros, B. Cuesta, R. Fernández-Pascual, M. E. Gómez, M. E. Acacio, A. Robles, J. M. García, and J. Duato. EMC^2: Extending magny-cours coherence for large-scale servers. In *17th Int'l Conf. on High Performance Computing (HiPC)*, pages 1–10, Dec. 2010.

[68] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato. Temporal-aware mechanism to detect private data in chip multiprocessors. In *42nd Int'l Conf. on Parallel Processing (ICPP)*, pages 562–571, Oct. 2013.

[69] A. Ros and A. Jimborean. A dual-consistency cache coherence protocol. In *29th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, pages 1119–1128, May 2015.

[70] A. Ros and A. Jimborean. A hybrid static-dynamic classification for dual-consistency cache coherence. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, PP(99), Feb. 2016.

[71] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 241–252, Sept. 2012.

[72] A. Ros and S. Kaxiras. Callback: Efficient synchronization without invalidation with a directory just for spin-waiting. In *42nd Int'l Symp. on Computer Architecture (ISCA)*, pages 427–438, June 2015.

[73] A. Ros and S. Kaxiras. Racer: Tso consistency via race detection. In *49th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 1–13, Oct. 2016.

[74] D. Sanchez and C. Kozyrakis. SCD: A scalable coherence directory with flexible sharer set encoding. In *18th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 129–140, Feb. 2012.

[75] A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-based tlb preloading. *SIGARCH Comput. Archit. News*, 28(2):117–127, May 2000.

[76] A. Sembrant, E. Hagersten, and D. Black-Schaffer. The direct-to-data (d2d) cache: Navigating the cache hierarchy with a single lookup. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 133–144, 2014.

[77] A. Sembrant, E. Hagersten, and D. Black-Schaffer. A split cache hierarchy for enabling data-oriented optimizations. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2017.

[78] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *10th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 3–14, Sept. 2001.

[79] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-end sequential consistency. In *39th Int'l Symp. on Computer Architecture (ISCA)*, pages 524–535, June 2012.

[80] I. Singh, A. Shriraman, and W. W. L. Fung. Cache coherence for gpu architectures. In *19th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 578–590, Feb. 2013.

[81] S. Srikantaiah and M. Kandemir. Synergistic tlbs for high performance address translation in chip multiprocessors. In *43rd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 313–324, Dec. 2010.

[82] M. Talluri and M. D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *6th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 171–182, Oct. 1994.

[83] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs, Apr. 2008.

[84] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. DiDi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 340–349, Oct. 2011.

[85] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[86] D. A. Wood, M. D. Hill, and R. Kessler. A model for estimating trace-sample miss ratios. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 79–89, 1991.

[87] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang. Efficient timestamp-based cache coherence protocol for many-core architectures. In *Int'l Conf. on Supercomputing (ICS)*, ICS '16, pages 19:1–19:13, 2016.

[88] J. Zebchuk, B. Falsafi, and A. Moshovos. Multi-grain coherence directories. In *46th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 359–370, Dec. 2013.

[89] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A tagless coherence directory. In *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 423–434, Dec. 2009.