SOFTWARE-ORIENTED DATA ACCESS CHARACTERIZATION FOR CHIP MULTIPROCESSOR ARCHITECTURE OPTIMIZATIONS

by

Yong Li

B.S. Telecommunication Engineering, Chongqing University, 2005

M.S. Computer Engineering, University of Pittsburgh, 2010

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2013

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Yong Li

It was defended on

Oct. 29th 2013

and approved by

SOFTWARE-ORIENTED DATA ACCESS CHARACTERIZATION FOR CHIP MULTIPROCESSOR ARCHITECTURE OPTIMIZATIONS

Yong Li, PhD

University of Pittsburgh, 2013

The integration of an increasing amount of on-chip hardware in Chip-Multiprocessors (CMPs) poses a challenge of efficiently utilizing the on-chip resources to maximize performance. Prior research proposals largely rely on additional hardware support to achieve desirable tradeoffs. However, these purely hardware-oriented mechanisms typically result in more generic but less efficient approaches. A new trend is designing adaptive systems by exploiting and leveraging application-level information. In this work a wide range of applications are analyzed and remarkable data access behaviors/patterns are recognized to be useful for architectural and system optimizations. In particular, this dissertation work introduces software-based techniques that can be used to extract data access characteristics for cross-layer optimizations on performance and scalability. The collected information is utilized to guide cache data placement, network configuration, coherence operations, address translation, memory configuration, etc. In particular, an approach is proposed to classify data blocks into different categories to optimize an on-chip coherent cache organization. For applications with compile-time deterministic data access localities, a compiler technique is proposed to determine data partitions that guide the last level cache data placement and communication patterns for network configuration. A page-level data classification is also demonstrated to improve address translation performance. The successful utilization of data access characteristics on traditional CMP architectures demonstrates that the proposed approach is promising and generic and can be potentially applied to future CMP architectures with emerging technologies such as the Spin-transfer torque RAM (STT-RAM).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

Among many people who helped me with this work, I first thank my advisor, Dr. Alex Jones, for his relentless support throughout the entire duration of my graduate research, which forms the foundation of this dissertation. It was him who invited me to his excellent research group in which I initiated my first research project and have been actively participated during my PhD program. His instructive advice helped me to build my research experiences from ground up and follow the right direction since then. His strong enthusiasm motivates me to concentrate on my high performance computing research. Without his help, I could have never done this work.

Second, I would like to thank Dr. Rami Melhem, who has co-advised my research work for over five years of my graduate study. His encouragement at the early stage of my work made me feel warm and helped me through the hard times. It was from his words I gained the confidence to pursue a PhD degree. His patient guidances and directions not only helped me to conquer the difficulties I have experienced in my research work but also equipped me with valuable capabilities necessary for conducting research. From him I have learned many useful techniques including presentation/reasoning skills, academic paper writing, research idea formulating, etc.

I also thank Professor Yiran Chen, Professor Hai Li and Professor Zhihong Mao for being on my program committee and giving me constructive advice on this dissertation. I highly appreciate their time spent on reviewing the dissertation.

Thanks are also given to Ahmed Abousamra, my research group mate, for his suggestions and collaborative work. It was him who helped me to get a quick start on conducting research experiments.

Finally, I owe a thank you to my wife, Ruoxin Zhang, for her constant care and support during my entire PhD program. Her contribution to the family has enabled me to concentrate on my research work.

1.0 INTRODUCTION

The chip-multiprocessor (CMP) paradigm is now a prevalent platform to harness the increasing amount of on-chip transistors and offer massively parallel computing power that can be leveraged by a set of diverse applications. With the scaling of computing power available on a single die, effectively using resources is becoming increasingly important for designing high-performance, low-power and cost-efficient CMPs. To better apply these resources for today's diverging application requirements, new demands arise for "smarter" computer architectures that are more efficient and adaptive to application characteristics and deliver higher performance per watt. This dissertation work explores extracting application-level information that can be used by modern computer architecture components (e.g., caches, networks-on-chip and translation lookaside buffers (TLBs)) for optimized performance, scalability and efficiency. This chapter opens by introducing several challenges in building scalable and efficient modern CMP systems. Limitations of prior approaches and a brief introduction of the proposed solutions are also discussed in this chapter.

1.1 CMP CHALLENGES

The increasing amount of computing capabilities integrated on chip requires both architectural innovation and consideration of the impact from technology scaling. While the performance continues to increase, the system suffers from many scaling induced issues including prolonged remote data access latency, more expensive communication overheads, increased leakage power, etc. The following two subsections discuss specific issues of architecture and technology scaling that motivate many CMP system optimizations including the proposed work.

1.1.1 Architecture Scaling

Architectural resources in modern parallel computing platforms such as cache memories and interconnects are constantly evolving at a rapid pace to drive the computing capabilities and satisfy the increasing performance and scalability requirements. Unlike traditional computing processors in which a small number of cores are tightly integrated on a bus, current and future computers tend to have a larger number of integrated cores with local as well as shared resources connected using a distributed network-on-chip (NoC).

Organizing multiple processing cores in a distributed manner increases the inter-core communication bandwidth and mitigates the contention issue when handling data communication for a large number of cores. However, a well know issue associated with the distributed NoC is the multi-hop communication latency for accessing remote data. On the widely used non-uniform cache architecture (S-NUCA) [55] in which data blocks are interleaved across multiple cores, data communication latency due to remote accesses is significant. The increase in core count also complicates cache coherence operations typically required in multi-core systems to maintain data consistency. As a result, more storage resources and coherence messages are required to track and maintain the coherence states among multiple cores. In addition to the aforementioned issues, architecture scaling also challenges many other aspects of CMP designs, such as the virtual to physical address translation mechanism.

1.1.2 Technology Challenges

Technology scaling is another driving force of modern CMP computing platforms. In particular, the shrinking transistor size and decreasing gate voltage of smaller technology process enable the designing of more powerful and energy-efficient CMP systems. The downside of this scaling trend is a variety of challenges (e.g., increased leakage power and reliability) that must be addressed before the technology process can become an overall competitive solution. One particular scaling issue this dissertation targets is the reduced sensing margin and consequently degraded read performance due to the process scaling in Spin-transfer torque RAM (STT-RAM), which has been been actively studied as a low leakage alternative for conventional memory technologies such as SRAM and DRAM. As the technology scales, the supply voltage, transistor size, and transistor gate voltage of an STT-RAM cell decrease. Additionally, process variation [60] at smaller technology nodes begins to show a significant impact on device operations and result in a distribution of various electronic properties. As a result, data read performance suffers from larger sense amplifier delays for detecting increasingly small sense margins which is further exacerbated by the process variation.

1.2 PROPOSED SOLUTIONS

To address the issues presented in Section 1.1, this dissertation proposes various cross-layer optimizations to mitigate the coherence penalty, data access latency, communication overheads and improve address translation efficiency in scalable architectures. The proposed approaches leverage application-level information including data access classification, memory access patterns and data reuse behaviors in multi-threaded programs to optimize different architectural components such as caches, interconnect and TLBs.

In order to achieve a desirable utilization of the data access behaviors of a certain application in different scenarios, it is necessary to consider the information from either the compiler or OS, depending on the optimization target. Compiler techniques can be applied in many cases to extract fine-grained data access information and provide a method to examine an application's "future" characteristics *a priori*. OS-based mechanisms are good complements to a compiler approach and can efficiently retrieve page-level characterization when fine-grained information is not necessary. To gain an understanding of the behavior of today's parallel applications, various multi-threaded benchmarks from the SPLASH [5], PARSEC [15] and RODINIA [21] benchmark suites are studied. Many of them exhibit regular data access patterns that can be statically extracted by a compiler or dynamically captured by an OS and used for cross-layer optimizations. This dissertation proposes techniques for characterizing data accesses in multi-threaded applications. The data access characterizations are used for a variety of optimizations including data placement among NUCA [55] caches, data-classification-aware caching, communication pattern prediction, circuit scheduling in network-on-chip (NoC), address translation acceleration and emerging memory configuration.

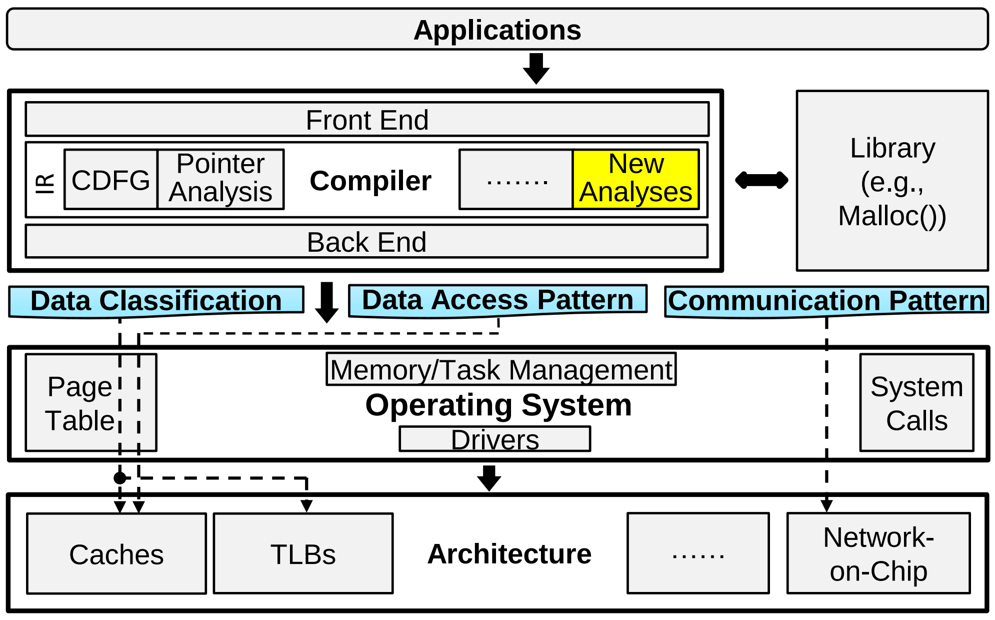


Figure 1: Overview of compiler- and OS-oriented architecture optimizations

Figure 1 illustrates a high-level overview of the proposed system with four layers: application, compiler, operating system (OS) and architecture. In the system the input applications are first processed by the compiler's front end and transformed into an intermediate representation (IR), on which a series of program analyses are performed. Many traditional compiler analyses are leveraged, and new compiler analyses are developed to extract useful application information that can be used by various system components at the architecture level. The compiler interacts with a customized library that can assist in passing the information to the OS or the underlying architecture. Finally, the extracted program characteristics are passed to the architecture at runtime. One example from this dissertation is data classification information discovered from the compiler analyses is first instrumented with a memory allocation routine (e.g., *malloc()*) and then passed through the page table to caches and TLBs for architecture optimizations. A second example is that data access pattern and communication pattern information can be passed to on-chip caches and NoCs through the OS page table [69] in a similar fashion to guide the cache data distribution and NoC configuration. Further, the instruction set architecture (ISA) can be extended with new instructions dedicated for architecture configuration, which is also discussed in this work.

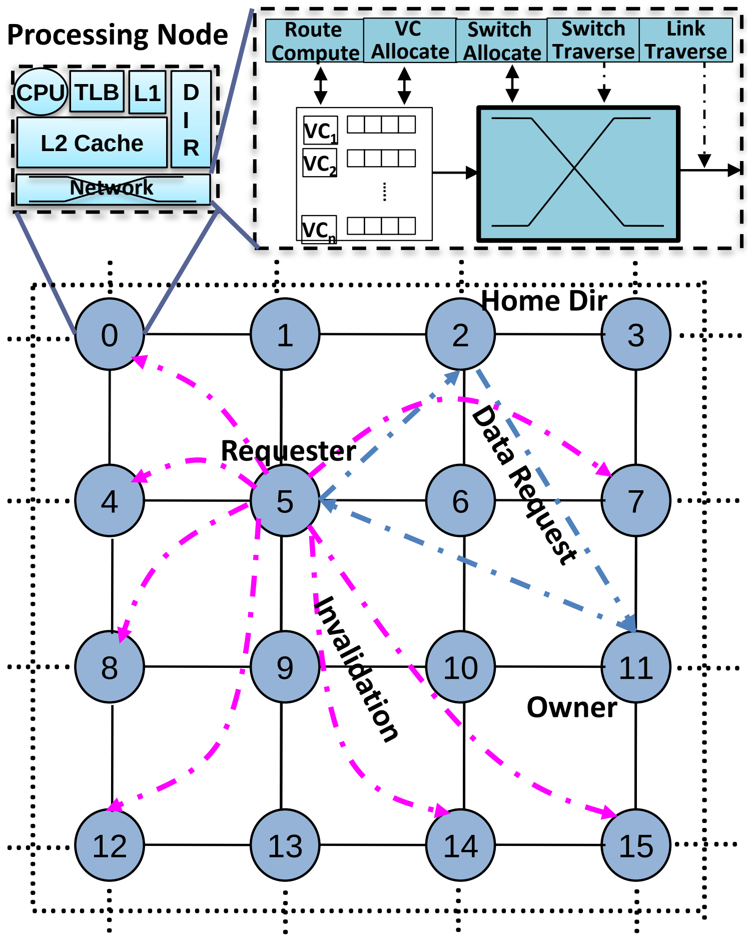


Figure 2: Scalable chip multiprocessor architecture

Figure 2 shows an example CMP architecture discussed as a baseline throughout the dissertation. The architecture contains several processing nodes and each node is composed of a processing core, a cache hierarchy (L1 and L2 caches), a TLB, a cache coherence directory (DIR) and a network switch. The network switch is pipelined in stages including route computation, virtual channel (VC [59]) allocation, switch allocation, switch traversal and link traversal. The cache coherence is maintained by a directory-based coherence protocol such as the MESI protocol [30]. In such an organization, a *requester* (e.g., node 5 in Figure 2) that accesses a datum not in the local cache may need to consult a non-local *home* directory (node 2), which provides the state information of the datum and forwards the request to the *owner* (node 11). The datum is finally returned to the requester from the *owner* and this results in multi-way communications over the NoC. If the requester issues a write on a data item, multiple invalidation messages must be sent out to all the nodes that cache that data item. All of these operations can lead to expensive multi-hop communications, long access latencies, complicated coherence operations and high power consumption. The situation deteriorates as the architecture scales to larger core counts.

To mitigate the problems raised by architecture scaling, many runtime approaches have been proposed [112, 61, 20, 114, 94] to achieve fast memory accesses, reduced coherence overhead and communication latency. These schemes typically rely solely on hardware and are not aware of distinct application behaviors, resulting in more generic but often less efficient solutions in many scenarios. For example, Sun *et al.* [94] proposed a hybrid SRAM/STT-RAM cache in which hardware counters are used to keep track of runtime write access patterns and guide data migrations between the two types of memory. Recently, systems that adapt to certain application data access behaviors have been proposed [116, 29, 42, 48, 53] to further enhance performance and scalability by efficiently managing the on-chip components including caches, interconnect and TLBs. For example, Hardavellas *et al.* [42] proposed an OS based page-level data classification mechanism to guide the data placement in multiple last level cache (LLC) banks so that both private and shared data pages are placed as if in their favored LLC organizations (i.e., private and distributed shared, respectively).

The aforementioned mechanisms improve system performance and scalability by leveraging application runtime information. However, these mechanisms can mislead the hardware in certain scenarios and result in configuration thrashing due to a lack of knowledge about the overall data access characteristics. In other words, runtime based approaches rely on "local" application behavior typically extracted from a small time window rather than more reliable "global" information. Consider the data classification mechanism [42] that classifies a page as either private or shared based on the number of cores that have accessed the page. In such a runtime scheme a single access from a second core to a page results in the page being classified as shared, even if all the subsequent accesses are all private. This mechanism provides a simple and efficient solution when only coarse-grained classification information is needed (e.g., address translation discussed in Chapter 5). For situations where finer grained data access information is required, the OS-page level mechanism can lead to inaccurate data classification. Tracking write access behaviors using hardware counters for data migration [94] can also result in problematic configuration and migration decisions. For example, a transient write access behavior can cause an unnecessary data migration if the detected behavior no longer persists after the migration.

An inherent drawback of the above mentioned approaches is that the runtime detected application characteristics may not be persistent to benefit an architecture configuration, resource scheduling, or a data distribution decision. A significant distinction that sets this dissertation work apart from the prior runtime based approaches is that in the enclosed solutions, relatively stable program behaviors are detected by compiler techniques and used by customized architecture designs that are aware of the detected information. By leveraging the compiler's capability in analyzing program characteristics over a larger execution window (e.g., global data flow information, intra- and inter-procedure analyses, etc.), the proposed solutions avoid temporally misleading program behaviors and manage CMP resources more efficiently.

1.4 CONTRIBUTIONS

This dissertation work shows that how a variety of data characterizations including access patterns, data classification, sharing characteristics and reuse behavior can be leveraged for different architecture optimization goals. The proposed approaches are demonstrated on a cluster of novel designs and systems that are aware of certain application data access characteristics, particularly from the compiler or OS, to enhance various aspects of CMP architectures:

- **Compiler-assisted Data Classification for Cache and TLB Optimization**: This dissertation introduces a fairly simple but generic compiler analysis that can classify data into different categories including private and shared style memory accesses. In addition, to address several issues raised by runtime data classification mechanisms, a data classification termed *practically private* is introduced. The concept of practically private implies a situation where the compiler cannot prove that a data block is only accessed by one processor but speculatively determines that private style access is highly probable. Practically private classification also covers scenarios where most elements of a data block are private or a data block is mostly accessed privately. Based on the data classification two architecture optimizations are proposed:

  - By designing a cache coherence protocol that is aware of the data classification discovered by the compiler, the system achieves fast data accesses, efficient utilization of the on-chip LLC capacity, reduced communication latency and coherence overhead.
  - The private-shared data classification is also used in the designing of a novel TLB architecture to provide translation sharing while keeping translation latency low. The new TLB design is based on traditional private TLBs but in addition provides a shared buffer to accommodate shared translations dictated by a page-level data classification mechanism. Since translation data is cached in TLBs at the OS page granularity, both the compiler and a simple and efficient OS-based approach are considered to provide the classification support for the proposed TLB design. The proposed TLB is demonstrated to outperform state-of-the-art translation solutions.

- **Compiler-assisted Data Partitioning and Communication Pattern Analyses:** Compiler techniques are proposed to determine multi-threaded memory access patterns and data partitioning in parallel programs that exhibit compile-time deterministic data access locality. Based on the detected data partitioning, each block is assigned an owner, which is the thread/core that accesses the block most frequently. Given the multi-threaded memory access patterns and data partitioning the compiler further determines communication pattern in an application. The data ownership and communication pattern can be used for efficient CMP architecture design:
  - Ownership information is instrumented with memory allocation and passed to an architecture with an ownership-aware NUCA cache organization. Data blocks are distributed across NUCA banks based on the ownership information for improving locality and reducing remote data accesses.
  - The communication patterns are used to guide circuit establishment to improve the circuit utilization and reduce circuit scheduling overhead in a hybrid packet/circuit-switching NoC [1].

- **Compiler-guided Configurable STT-RAM L1 Cache:** Finally, this dissertation work demonstrates that data access characteristics are important in designing next generation computer architectures leveraging emerging memory technologies. In particular a configurable STT-RAM cache architecture designed to operate in two modes is studied. One mode is slower in servicing read accesses but has high density and low energy advantages. The other mode offers faster read accesses at the expenses of high dynamic write power and reduced memory density. Compiler techniques are demonstrated to identify consecutive data accesses to help the mode configuration to achieve optimized tradeoff between read speed, density and write power.

The evaluation demonstrates that the compiler-assisted data classification mechanism reduces an average of 46% coherence traffic and achieves around 10% speedup over the shared caching scheme for a set of tested parallel applications. The access-pattern-guided data placement scheme achieves a 20% speedup over traditional shared cache and leveraging communication patterns for network configuration provides an additional 5.1% performance gain. By leveraging data classification in address translation, the TLB can achieve a nearly 50% reduction in off-chip misses and 45% improvement in translation latency. Finally, the compiler-assisted configurable STT-RAM cache brings 5% performance gain over SRAM and 10% performance improvement with less than 2% dynamic power increase over STT-RAM designs without read optimizations at 22nm technology.

1.5 OVERVIEW

The rest of the dissertation is organized as follows: Chapter 2 describes the background and related research efforts. Chapter 3 introduces the compiler-assisted data classification mechanism. Chapter 4 presents an data-classification-aware coherence cache design to reduce remote data accesses and coherence overhead. Chapter 5 presents an advanced TLB design leveraging the data classification information to improve TLB sharing and reduce translation misses. Chapter 6 elaborates the proposed compiler techniques for analyzing data partitioning and communication patterns. An architecture with customized cache and NoC designs to utilize the data partitioning and communication pattern information is detailed in Chapter 7. In Chapter 8, the configurable architecture based on STT-RAM that leverages compiler-extracted consecutive reuse information is discussed. Finally, Chapter 9 summarizes the dissertation work.

2.0 RELATED WORK

The proposed approaches in this dissertation are cross-layer and leverage system-level software such as compilers for optimizing a variety of architectural components including caches, NoCs and TLBs. This chapter is dedicated to introducing a general background and prior arts related to the work presented in this dissertation. More specific and detailed background relevant to each optimized component will be addressed in the subsequent chapters.

2.1 COMPILER OPTIMIZATIONS

Compiler analyses and optimizations have been proven to be critical to improve code efficiency, reduce resource utilization, and expose optimization opportunities. Some of the proposed data characterization approaches are dependent on a series of conventional compiler analyses including control and data flow analysis, symbolic analysis, etc.

A control flow graph (CFG) [3] is a directed graph built on top of the intermediate code representation abstracting the control flow behavior of a function that is being compiled. In a CFG, each vertex represents a basic block [1] and each edge represents a possible transfer of control flow from one basic block to another. Since the CFG logically represents the relationship among different components of a program, it forms the basis for a large number of compiler analyses and optimizations such as pointer analysis, reaching definition, liveness analysis, dead code elimination, loop transformation, constant propagation, branch elimination, instruction scheduling, etc.

For data flow analyses, each basic block is further represented as a data flow graph (DFG) [49]. A DFG, which is also a directed graph, carries the data dependencies within the code between

[1]A basic block is a continuous sequence of code with only one entry point and only one exit

control points. In a DFG, each node represents an operator (e.g. addition, logical shift, etc.) or an operand (e.g. constant, variable, array element, etc.). Each directed edge indicates a data dependency that denotes the transfer of a value.

Based on a combination of basic compiler optimization approaches, one can perform various high-performance oriented compile-time optimizations including dependence analysis, data reuse analysis, data access analysis, loop transformation, auto-parallelization [37, 40, 39], as have been done by many researchers.

Some early attempts have been made to analyze data accesses in a nested loop and find a data or loop partitioning among parallel threads. Ramanujam and Sadayappan [77] used a matrix representation to formulate an optimized data partitioning applied to shared memory multiprocessors without caches. Ju and Dietz [51] attempt to determine a data layout (row or column major) for a memory block in a uniform memory access (UMA).

In the PARADIGM compiler [38], data usage in parallelizable Fortran 77 and HPF (high performance Fortran) code was analyzed and partitioned across machines with distributed memories. Optimized communication operations were generated for the parallelized code. Another similar effort has been made by Kremer *et al.* [54], in which an automatic data layout specification was produced using 0-1 integer programming.

In the work from Lam's group at Stanford [105], data dependence was studied using a special case of the integer programming approach. Leveraging features of well formatted programs, their analysis results in a polynomial time algorithm. For data that carries no dependence, they distributed loop iterations across multiple processors. In another attempt, Barua *et al.* [11] developed a heuristic method to handle data partitioning in a way that avoids NP-complete linear programming.

There are also a number of efforts made to represent memory access patterns within single-threaded programs with the intent of assisting compiler transformations. Tu and Padua [100] approximated memory accesses by representing each array access subscript using a triplet notation. Li and Yew proposed a representation named *atom image* [68] to capture the coefficients of indices and loop bounds in a program. A more precise notation is *convex region* [99, 28], which expresses the geometrical shape of array accesses. Paek introduced the concept of the *array access region* that uses span-stride pairs [73] with an abstract access form to represent memory accesses within a program phase such as a nested loop. Note that the array access region theory forms the foundation of the multi-threaded data access pattern analysis presented in this dissertation (see Chapter 6).

Based on array access regions, Paek and Padua [74] presented an advanced compiler framework to achieve automatic parallelization and communication generation for a machine that does not employ automated cache coherence. Chu *et al.* adopted a profiling based scheme [27] for determining affinity relationships between memory accesses and computation operations. Data accesses and computations are partitioned across data caches of each core to avoid memory stalls and improve computation parallelism. In another attempt from the same research group [26], a compiler-directed approach was proposed to cooperatively partition data objects and the associated computation across multiple clusters to achieve improved locality and reduced communication. Shao *et al.* proposed complier analyses to identify communication patterns in MPI-based parallel applications. The identified patterns are used to configure on-chip networks to avoid circuit establishment overheads and improve communication speed.

In another recent attempt [70], a polyhedral model is used to perform localization analysis based on the Farkas Lemma and Fourier-Motzkin algorithm. The goal of this analysis is to find a data layout transformation to promote locality of accesses which would otherwise be destroyed by finely interleaving data among the tiled banks. This work targets array accesses in sequential programs and restructures the array indexing such that a distributed shared cache policy is retained, but most of the addresses accessed by a particular thread are mapped to the tile on which that thread runs. It assumes a fixed way of partitioning the iteration space (hence the data space) among parallel threads and works in a similar way to that of some conventional paralleling compilers.

2.2 CMP ENHANCEMENTS

There is a large body of research approaches proposed to optimize different CMP components including coherent caches, NoCs and TLBs. Most of these approaches target one CMP component and are generic in that they are independent of other architectural features and characteristics of applications. This section introduces relevant research efforts at architecture level. In particular, Section 2.2.4 discusses related work leveraging program characteristics for CMP optimizations.

2.2.1 Coherent Caches

Ever since the concept of the static non-uniform cache access architecture (S-NUCA) [55] was proposed, extensive research has been conducted to mitigate the impact of poor data proximity associated with this scheme often by adopting different variations of a private caching [19]. However, private caches, due to the replication of shared data, do not utilize cache capacity as effectively as a distributed shared S-NUCA cache[2]. Several hybrid approaches have been proposed [111, 58] starting either with shared or private cache organizations to achieve an optimized design. Specifically, Zhang and Asanovic [114] proposed the victim replication cache scheme based on a shared L2 cache structure. In their work, the local L2 cache slice is used as a place to hold victim cache lines from local as well as remote L1s. Hammoud *et al.* designed a cache organization in which the unique copy of a data block is placed at the "center of gravity" of its requesters [41] to simplify coherence operation, save directory entries and reduce average access latency for multiple processing cores.

Chang and Sohi [20] designed cooperative caching based on private caches. They enlarged the effective cache capacity by evicting cache lines which have multiple copies prior to those with only a single copy. They also studied optimizations such as cache-to-cache transfer of clean data, replication-aware data replacement and global replacement of inactive data. Dybdahl [34] tried to combine the advantages of both shared and private caches by dividing cache banks into shared versus private partitions. The size of each partition changes dynamically depending on the miss rate of the corresponding bank. Other attempts have been made to mitigate the cache coherence overhead by designing more efficient directory structure [112] or adopting policies that simplify the coherence states such as the self-eviction policy [82].

Unlike the proposed techniques, which leverage software-extracted application characteristics, the above runtime cache optimizations rely on dedicated hardware to collect data access and sharing information. The collected information typically reflects only application behaviors in a small time window thus can result in frequent but inefficient cache reconfigurations.

[2] In this work "shared" is used as a short term to refer to distributed shared S-NUCA.

2.2.2 Network-on-chip

Interconnect communication is also becoming a major factor limiting CMP performance and scalability and has received significant research attention. Conventionally, interconnect is either packet or circuit switching. In traditional packet switching, packets must undergo several pipelined stages including decoding at input ports, buffering in virtual channels, computing routes, arbitrating/allocating switch resources and traversing the switch links. Communicating a message from a source to a destination typically involves multiple hops and thus the latency is high. By contrast, in circuit-switching networks circuits are established between source and destination nodes to achieve direct communication. However, the circuit establishment overhead is high and circuits are sparse resources that should be utilized carefully.

Recently, there have been several attempts to create configurable and hybrid networks to leverage the benefits of both packet and circuit switching techniques. For example, Peh's group at Princeton has developed the concept of Express Virtual Channels (EVCs) [62, 59]. EVCs provides a flow control mechanism that allows data packets to bypass arbitration and routing stages in a pipelined switch. With predefined virtual express paths (fast paths), packets can skip virtually the entire router pipeline at intermediate nodes along their paths and thus the communication delay approaches that of a dedicated wire interconnect. Jerger *et al.* [47] proposed a hybrid circuit switching, a technique that removes the circuit establishment overhead by intermingling packet-switched flits with circuit-switched flits. A prediction-based coherence protocol is also designed to leverage the existence of circuits and promote circuit reuse between sharer cores. Abousamra *et al.* [1] use a runtime system to determine the most beneficial fast paths to establish based on runtime collected traffic statistics. The fast paths, once established, are pinned for a predetermined period of time to promote locality and avoid expensive overheads due to frequency circuit establishments.

Many other strategies for reducing communication latency for traversing the network rely on methods for reducing the global hop count [57, 8, 18, 31]. For example, Kim *et al.* [57] proposed an flattened bufferfly network topology using high-radix routers to reduce the diameter of the network and hence the communication latency. They also exploited two dimensional network layout and channel bypass to further optimize on-chip communication. In another attempt [31], a hybrid topology that combines the bus and low-radix mesh was proposed to improve communication efficiency. The bus is used for local communication among a few number of nodes while the mesh topology is used for global communication. Variants of the mesh-based network [8, 18] have also been demonstrated to have benefits over traditional mesh networks. These approaches adapt traditional mesh network based on traffic patterns to improve link utilization.

2.2.3 Address Translation and TLBs

The TLB is another important CMP component on the critical path of memory accesses which has recently received considerable attention. Traditional TLBs are designed as private to each processing core in CMP systems to avoid long access latency. However, providing sharing for TLBs offers potential benefits of reduced TLB misses, enlarged TLB capacities and optimized TLB operations (e.g., TLB flush).

In particular, Bhattacharjee *et al.* recently presented a study of TLB sharing characterization [13] that motivated a number of research efforts including Synergistic TLBs [93] (detailed in Section 5.2.2). Based on the study of TLB sharing behavior, Bhattacharjee *et al.* proposed a TLB prefetching scheme [14] to reduce TLB misses. Using a technique called *leader-follower*, when a tile misses in the local TLB it becomes the "leader." The fetched TLB entry is sent to the prefetch buffer of other "following" cores that frequently utilize the leader's pages. A second technique, *distance-based cross-core*, matches the historical distance between TLB misses and predicts/prefetches TLB entries based on pattern matching. The system records the two distances between three successive TLB misses in a *distance table*. When two misses in any core match the first distance, the page matching the second distance is prefetched into the prefetch buffer.

A more detailed discussion of TLB related work and their comparison with the proposed TLB optimization can be found in Section 5.2.2.

2.2.4 Application-aware Optimizations

A number of mechanisms have been recently proposed to utilize application data access characteristics to efficiently optimize various hardware components of CMPs such as caches, interconnects, coherence directories, etc. As these approaches are highly relevant to the proposed work, a summary of these approaches is provided below.

Cache: In reactive NUCA (R-NUCA) [42], data accesses are classified as private, shared, and read-only at the page granularity. Data is assumed to be private until a second core accesses the data (signaled by a TLB miss). Data lines from private pages are cached locally to improve access latency while lines from shared pages are cached using S-NUCA style [56, 55] (i.e., distributed shared data placement) to improve capacity.

Another relevant effort is Jin and Cho's software oriented shared (SOS) cache management [48]. They classify data accesses to a range of memory locations (returned by a memory allocation function such as `malloc()`) into several categories such as Even Partition, Scattered, Dominant Owner, Small-Entity, and Shared. Applications are profiled and memory accesses are matched to one of the above categories. Hints are provided to the memory allocation functions based on the matched access patterns. Pages within the memory ranges are assigned to cache tiles indicated by these hints.

Cuesta et al. [29] recently presented an efficient cache coherence directory based on a runtime data classification scheme similar to the one used in R-NUCA. The proposed scheme saves more than 50% coherence directory entries by distinguishing shared data blocks from private ones and maintaining directory entries for only the shared data blocks. When the classification of a data page experiences a transition from private to shared, as detected by the OS, a coherence recovery process is invoked to recover the coherence states.

R-NUCA and SOS are both techniques that leverage execution history and profiling to detect the data access pattern. In addition, the two techniques extract information at the OS page granularity. Consequently, they have a higher probability of one data access pattern being polluted by another compared to a technique that utilizes application information at a finer granularity such as a cache line or an element of a data-structure.

Interconnect: Several attempts have been made to understand the communication characteristics of parallel programs [7, 43, 35, 89, 86, 103, 9, 79]. Most of these attempts, however, reveal only coarse properties of communication behaviors such as "point-to-point" and "collective". They do not provide accurate descriptions of communication patterns. The work in [86] uses a compiler to discover the communication behavior for MPI-based (message passing interface) programs and uses a matrix to describe the communication among multiple processors. The compiler is extended to include a phase partitioning algorithm and a scheduling methodology to configure a hybrid electronic packet-switched and optical circuit-switched interconnect [88]. However, these efforts only focus on explicit communications in MPI-based programs, omitting the implicit communication patterns implied by a shared-memory cache system.

2.3 EMERGING MEMORIES IN CMPS

STT-RAM has been proposed for use in CMP cache hierarchies as a potential replacement for SRAM, particularly for LLC. STT-RAM caches can leverage both non-volatility for reduced leakage power and increased density and capacity over SRAM. Previous conventional wisdom for STT-RAM is that writes are slower and require more power than their conventional SRAM counterparts, although recent research efforts [115, 60, 72, 110] demonstrate that read performance becomes a new bottleneck as technology scales down to 45nm and below (detailed in Chapter 8). This section mainly discusses several techniques proposed at different levels (i.e., device, architecture and compiler levels) to mitigate the STT-RAM write challenges.

According to Smullen *et al.*, the excessive long write delay can be significantly reduced by relaxing the non-volatility [91] or reducing data retention time [96] with dynamic data refresh support to retain data. The write energy can be also saved by adopting early write termination [117], which avoids unnecessary writes in STT-RAM cells.

There are similar techniques proposed for combating write related penalties in phase-change memory (PCM). Qureshi *et al.* [76] recently proposes PreSET, a scheme aimed to improve read-/write performance by leveraging the asymmetry in writing different logic values. Their earlier effort [75] attempts to alleviate the penalty of pending reads caused by long write delays using write cancellation and write pausing.

STT-RAM optimizations have also been substantially studied at architecture level. Guo *et al*. [36] use STT-RAM to re-design a number of non-write-intensive micro-architectural components. They also adopt a subbank write buffering policy with read-write bypassing to increase write throughput and hide the high write latency. Wu *et al.* [108] proposed a region-based hybrid cache architecture (RHCA) and a level-based hybrid cache architecture (LHCA) by combining disparate memory technologies including SRAM, STT-RAM and phase change memory (PCM). Dedicated hardware units are used to collect data write intensity information and distribute data blocks to appropriate types of memories to reduce access latency and power consumption. Rasquinha *et al.* [78] proposed new promotion and insertion policies that operate differently for read versus write operations. The high write energy of STT-RAM was addressed by adopting a new replacement policy that increases the residency of dirty lines at higher cache levels (e.g., L1) at the expense of a higher data miss rate.

Li *et al.* [65] proposed a compiler-assisted technique to improve the performance and energy efficiency for embedded systems with STT-SRAM hybrid caches by reducing the migration overhead. In particular, the work identifies migration-intensive memory blocks through compiler analysis and give those blocks higher priority to be placed in the SRAM component to avoid frequent migration and long write latency on STT-RAM.

Chen *et al.* [23] developed a compiler pass that provides data placement hints to reduce STT-RAM write frequency on a customized hardware that can correct the compiler hints based on runtime cache behavior. In their solution a compiler technique is proposed to leverages the concept of memory reuse distance. However, the reuse distance concept used in their work is similar to the conventional reuse analysis and not aware of the read-write interleaving patterns. Thus, it cannot identify optimization opportunities brought by consecutive reads.

3.0 COMPILER ANALYSES FOR DATA CLASSIFICATION

For multi-threaded applications running on CMPs, an important and commonly used approach for characterizing accesses to a data block is to examine how many cores/threads access that data block, based on which the block can be classified as private (accessed by one core) or shared (accessed by more than one core). This chapter presents how private versus shared data accesses can be detected using lightweight compiler analyses. To further reduce the compilation complexity and avoid the downside of a runtime data classification mechanism such as data classification pollution (see Section 3.2.1), a new data classification termed *practically private* is introduced. Practically private indicates that the compiler cannot easily prove that the memory is accessed privately (e.g., exclusively by one thread/core) but speculatively determines that private access is highly probable and any sharing is minimal. Due to the speculative nature of the practically private concept, the complexity of the compiler analysis required to capture data sharing information is greatly reduced, resulting in a simple and effective approach.

The practically private data classification promotes access proximity for many data blocks used by parallel applications that would be treated as shared in many run time schemes. Moreover, the proposed classification is helpful in designing a more efficient coherence protocol that distinguishes practically private versus shared data, which have remarkably distinct sharing and coherence behaviors. This chapter demonstrates that practically private data is ubiquitous across a variety of applications and a high percentage of practically private data is dominated by local accesses. The following two chapters demonstrate that the detected application data classification information can be used to optimize coherent caches and TLBs.

3.1 BASIC ANALYZING APPROACHES

The proposed compilation methodology requires the use of several known compiler techniques that are well understood. This section discusses a few of these optimization passes and their applications in order to provide a background for the compiler methodology described in the following sections. In particular, the existing libraries available in the parallel compiler infrastructures SUIF [105] such as the dependence test, data flow framework [98], aliasing analysis, etc., are helpful in the development of the target compiler-based analysis and data classification.

In programs that utilize dynamically allocated objects, the starting addresses of memory blocks returned by `malloc()` are usually assigned directly to pointers. Other pointers can point to the same memory block through pointer assignments. To address the memory disambiguation problem, one necessary compilation technique is pointer analysis, which can be used to track memory access information of dynamically allocated memory. As such, the following definitions are introduced:

Definition 1. *A* pure pointer *is a pointer that has been directly assigned the return value of a heap memory allocation routine such as* `malloc()`*.*

Definition 2. *A* derived pointer *is a pointer that is derived from a pure pointer either directly or indirectly based on a pointer assignment.*

Definition 3. *A* reference list *for an allocated memory block is a collection of pointers that can be used to refer to that block.*

To accomplish this analysis, a variant of Andersen's point-to analysis [4] was implemented. Instead of keeping a series of point-to pairs, a reference list is used to include all the pointers that may point to a particular memory block. By using the pointer analysis, array accesses can be resolved back to their pure pointers to simplify subsequent compiler analyses. A detailed example of the reference list based pointer analysis is shown in Section 3.3.

3.2 DATA CLASSIFICATION

A study of multi-threaded code from a variety of program domains such as scientific computing, multimedia, image processing and financial processing reveals that data structures can be used in quite different ways. It is also observed that the way data is usually used by multiple threads can be implied by information such as where in the virtual memory space the data is allocated and how the references of data are handled. For example, instructions and globally allocated data such as synchronization structures are typically shared by all threads. In contrast, stack and heap objects allocated within a thread usually have very few sharers. From the system performance point of view, the access characteristics of these data objects are so different that each of them should be treated using a customized design. This section describes the approach to classify and identify the data classification at compile time. Since a variety of multi-threaded benchmarks feature extensive usage of dynamic memory allocation for managing computed data, emphasis is given to analyzing data blocks allocated through memory allocators such as `malloc()`. The analysis approach can be extended to other memory allocation routines such as `new`.

3.2.1 Motivation: The Concept of Practically Private

Understanding the data access behavior in multi-threaded programs is essential to deliver high performance on CMPs. Parallel applications tend to exhibit flexible and diverse data access patterns. This poses a challenge for compiler to detect and describe these patterns. However, the study shows that there are several representative patterns/classifications that exist in a variety of parallel applications and they dominate the entire program execution. Consider the matrix multiplication example in Figure 3. The code in Figure 3(a) computes the product of matrix a and vector b and stores the result in x. Figure 3(b) shows a typical parallelized version of the same problem using POSIX threads, assuming the number of available threads is four ($PROCS = 4$). Both the source code and the illustration in Figure 3(b) imply a clear data classification: a and x are partitioned across the 4 threads and thus can be classified as private while b should be classified as shared since it is entirely shared by all threads. This example will be revisited in more detail in Section 3.3 to show how the data classification can be formally identified from source code.

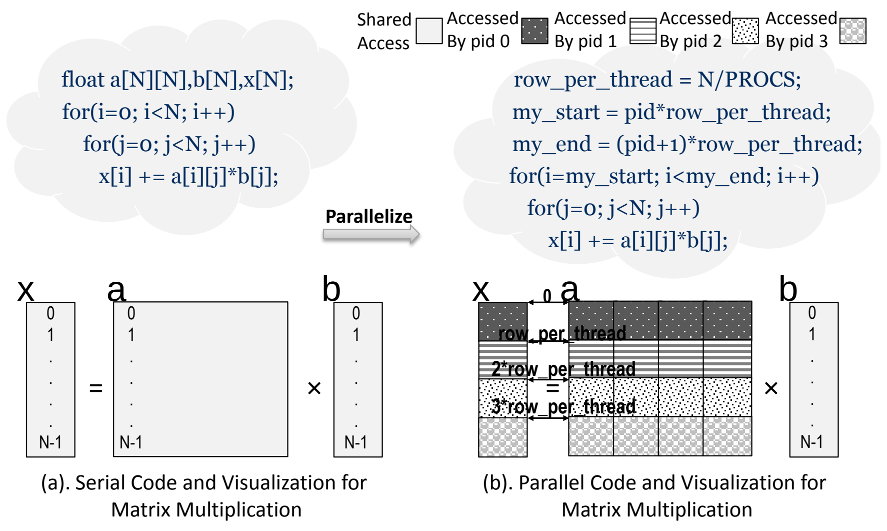


Figure 3: Serial and parallel computation for matrix multiplication

Generally, the classification of data must meet certain criteria to improve data access latency, save coherence directory entries or reduce network traffic. In particular:

- Different classifications within an application exhibit remarkable disparity in terms of locality, storage requirements, and access latency and as such must be treated differently.
- Classifications are practical for the compiler to identify.
- Classifications are representative for a wide spectrum of parallel applications.

A straightforward method is to classify data blocks into two distinct access categories: private versus shared, as shown in the above example. Private data is accessed by only one processor and thus is suitable to be placed locally to reduce access latency and promote locality. Coherence directory [113] size can also be reduced by eliminating entries for private data [29]. Conversely, shared data is accessed by more than one processor and should be optimized using different methods than private data. Shared data can be placed at a fixed location indexed by its address or at the "center of gravity" of its requesters [41] to reduce coherence traffic, save directory entries and simplify searching of data, especially when the data exhibits frequent/heavy sharing.

As a compiler-assisted data classification approach, the analyses must remain conservative when identifying private data to guarantee correctness. Unfortunately, identifying data privacy requires complicated and NP-complete compiler analyses (e.g., inter-procedural analysis, memory disambiguation, etc.), which drastically increase the compilation overhead and may still fail to guarantee data privacy in some complicated cases such as calling procedures by function pointers and accessing memory through pointer arithmetic. To avoid these complications and expensive analyses, the private-shared classification is extended with the third category, *practically private*. Leveraging practically private greatly reduces the burden of compiler analysis for ensuring data privacy and forms a data classification space the compiler can handle. The resulting three data categories are described as follows:

- **Private:** In multi-threaded applications, a data block (such as one returned by `malloc()`) is defined to be *private* if every element in it is accessed by only one thread in the parallel program segment. This is the case when multiple threads in the program partition a data block exclusively without overlap.

- **Practically Private:**

  For data that appears to be private but is not provably so from the compiler analysis, the data is classified as *practically* private. There are two scenarios that this classification covers. First, data that is *probably* private can practically be treated as private. Probably private data is frequently entirely private at runtime, as depicted in Figure 4(a), but safeguards are required to deal with cases when sharing occurs. Second, if the practically private data is not entirely private, typically it is still *mostly* private, as illustrated in Figure 4(b) and Figure 4(c). Figure 4(b) indicates mostly private data where each thread operates largely on exclusive data regions with shared boundaries (i.e., spatial perspective). Figure 4(c) illustrates mostly private (i.e., temporal perspective) where multiple threads operate exclusively on the data most of the time (e.g., data is shared only at the beginning or the end when threads are forked/joined). Typically, mostly private data has a low degree of sharing in terms of sharers and frequency of sharing, making it suitable for private treatment in practice.

- **Shared:** When a data block cannot be classified into the above two categories, the data block typically has a global scope and is likely to be accessed by multiple threads. Thus, the shared category is the default classification when a data block can be neither classified as private nor practically private.

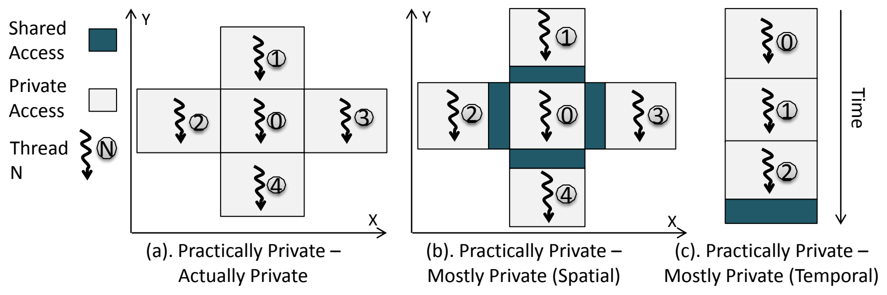

Shared Access
Private Access
Thread N
Y
X
(a). Practically Private – Actually Private
(b). Practically Private – Mostly Private (Spatial)
(c). Practically Private – Mostly Private (Temporal)
Time

Figure 4: Different scenarios for practically private data

From its definition, practically private data can be regarded as a relaxed version of the private data classification to ease the compiler analysis and reduce the classification detection overhead. Additionally, the concept of practically private addresses several issues raised by previously proposed data classifications [42, 29], which have a relatively rigid definition of private versus shared. First, the runtime classification scheme results in a "data pollution" problem that as little as one shared element in a page would lead to the whole page being classified as shared. This reduces the effectiveness of the data classification and can degrade performance for applications with boundary sharing (e.g., OCEAN, WATER, etc.). Another common case where the runtime data classification does not perform well is when a data page is initialized by one thread but heavily accessed by another thread. The runtime scheme mis-classifies the page as shared while private accesses are predominant. The above issues can be addressed by leveraging the concept of practically private.

3.3 DATA CLASSIFICATION DETECTION

In order to detect data classification for dynamic memory allocations, the compiler uses a reference list (see Section 3.1) to keep track of all the pointers that may point to a particular memory block. Initially, a reference list is created at each call site of `malloc()` and the return address is added into the reference list. Reference list updates utilize data flow analysis that traverses the CDFG (control and data flow graph) of the analyzed program as follows: Let s be a statement node in the CDFG, $succ(s)$ be the list of the immediate successor nodes of s, $pred(s)$ be the list of the immediate predecessor nodes of s, $In(s)$ be the reference list state before executing s and $Out(s)$

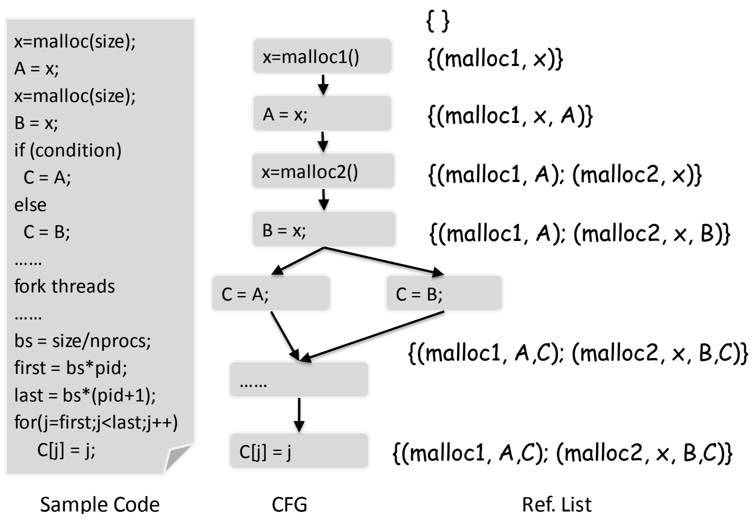
be the state after executing s. Each statement s in the program has two effects on $In(s)$ and $Out(s)$: *Gen* and *Kill*. *Gen* generates a new reference list or adds a pointer in an existing reference list, depending on the format of s. *Kill* removes a reference list or a pointer within it. For example, the statement $a = malloc()$ has the *Gen* function of creating a reference list with a in it and the *Kill* function of removing a from its current reference list. Likewise, the statement $b = a$ generates b for the reference list that contains a and kills b from its current reference list. The data flow equation for updating the reference list can be derived based on the above notions:

$$Out(s) = Gen(s) \bigcup (In(s) - Kill(s)) \tag{3.1}$$

$$In(s) = \bigcup_{s\prime \epsilon pred(s)} Out(s\prime) \tag{3.2}$$

To illustrate the operation of this analysis, consider the example shown in Figure 5. The sample code on the left allocates data using `malloc()` and accesses the allocated data after multiple threads are forked. The analysis begins by constructing a CFG. As the CFG is being traversed, `malloc()` call sites and pointer assignments are detected. Using the *Gen*/*Kill* functions and data flow equations, reference lists are created and updated, as shown on the right hand side in Figure 5. Initially, the reference list set is empty. At the first `malloc()` call site (labeled as $malloc1()$), x is added into the reference list as a pure pointer. The next assignment $A = x$ adds A into the reference list that contains x. When x is reassigned by a second `malloc()` x is removed from the reference list with A and a new reference list is created to which B is added with the succeeding assignment statement. When the conditional branch is encountered, C is added into both reference lists.

Note that some pointer assignments may involve pointer arithmetic (e.g., $C = A + a$), which complicates the memory ambiguity and consequently the data classification detection. Essentially, pointer arithmetic results in scenarios where pointers can point to arbitrary locations within a data block allocated by *malloc()*. Each of these pointers, when dereferenced and used in array accesses, indicates a particular classification. Consequently, multiple pointers that point to different locations in a data block may have a distinct classification, thus presenting a challenge on correctly classifying the data block. However, a study of various representative parallel applications from the SPLASH and PARSEC benchmark suites shows that multiple pointers with potentially different offsets pointing to a block rarely raise a data classification conflicts. This is due to the consistent private or shared data access pattern typically inherent in parallel applications. In other words, during the lifetime of an allocated memory block its sharing behavior typically does not exhibit a drastic change. Therefore, to simplify the data classification process and reduce compilation overheads without affecting the classification effectiveness, the analysis only considers the base pointers when pointer arithmetic is encountered.



```
x=malloc(size);
A = x;
x=malloc(size);
B = x;
if (condition)
  C = A;
else
  C = B;
……
fork threads
……
bs = size/nprocs;
first = bs*pid;
last = bs*(pid+1);
for(j=first;j<last;j++)
    C[j] = j;
```

x=malloc1()
A = x;
x=malloc2()
B = x;
C = A;
C = B;
……
C[j] = j
{ }
{(malloc1, x)}
{(malloc1, x, A)}
{(malloc1, A); (malloc2, x)}
{(malloc1, A); (malloc2, x, B)}
{(malloc1, A,C); (malloc2, x, B,C)}
{(malloc1, A,C); (malloc2, x, B,C)}
Sample Code
CFG
Ref. List

Figure 5: Pointer analysis example for data classification

3.3.1 Thread-Identifying Variables

A study of multi-threaded benchmarks shows that a large portion of data-parallel applications tend to exhibit regularity in their data access patterns. In the benchmarks considered in this dissertation, 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 defined as *Thread-Identifying (TI) Variables* [66]. TI variables are thread-local variables which have unique values for different threads of execution. Typically, these variables are used to determine which memory blocks and in some cases which portion of a memory block a thread will access. The compiler must identify which variables in the program are TI variables in order to determine how the pointers in the reference lists are dereferenced and used by each of the threads.

One of the commonly used methods for specifying TI variables is to pass different values to parallel threads as function arguments. As shown in Figure 6, the fourth argument of `pthread_create()` passes the addresses from `my_arg[0]` to `my_arg[num_threads]` from within a `for` loop. The passed addresses serve as local variables in the forked function `SlaveProcedure` where each instance has a local variable `my` with a unique value. Thus, `my` is a TI variable that can be detected by the compiler. Another common way to specify TI variables in multi-threaded applications is illustrated in Figure 7. Multiple threads try to access and modify a global variable in a critical region under the protection of a mutex lock. This type of code is much more difficult for a compiler to analyze. In particular, certain modifications of a global variable is not obvious for compiler to detect. For example, the value change of a shared variable in the critical region might be incurred by a procedure that requires non-trivial analyses. In addition, a modified variable in the critical region is not necessarily a TI variable if the modification is guarded by certain conditions. Thus, if the program uses this or any other methods to specify TI variables, the user is required to input a directive to specify the TI variables. For example in Figure 7, `#pragma TIV pid` specifies pid as a TI variable.

```
for(i=0; i<num_threads; i++) {
  my_arg[i] = i;
  pthread_create(&p_threads[i],&attr,
      SlaveProcedure,(void*)&my_arg[i]);
}
void SlaveProcedure(void *my)...
```

Figure 6: Detecting TI variable passed as parameters

```
#pragma TIV pid
pthread_mutex_lock(&(idlock));
 pid = Globalid;
 Globalid++;
pthread_mutex_unlock(&(idlock));
```

Figure 7: Detecting TI variable by directives

A special type of TI variable is *TI pointer*. TI pointers have the same property as TI variables. In other words, multiple instances of a TI pointer may point to different memory addresses in different threads. TI pointers can be identified using the TI variable detection approach described above. In particular, a global pointer can be claimed as a TI pointer if its value is modified in a region guarded by a mutex lock.

The classification of a data block is strongly implied by pointer dereference features. One such important features for the compiler to discover is pointer dereference (array access) with

TI variables. Note that variables derived from other TI variables, either through calculation or assignment, are also TI variables. Once the basic TI variables are identified, all other TI variables can be detected by performing a data flow analysis similar to the reaching definition problem [3]. Given the notions of the TI variable and the earlier introduced reference list, the data classifying rules are introduced as follows:

- **Determine Private Data:** Data can be assured to be private if all the pointers in the associated reference list are of local scope (i.e., pointers are never passed to global pointers or other threads).
- **Determine Practically Private Data:** Data on the stack is classified into this category. Heap data blocks are also identified as practically private if at least one pointer in the reference list is of global scope (i.e., the pointer is global or passed to other threads) and that pointer is dereferenced with a TI variable[1]. Typically, this indicates that these data blocks are probably private. In many embarrassingly parallel applications, probably private data is actually private when the global data blocks are exclusively partitioned among multiple threads by TI variables. Even if the practically private data blocks are not actually private, it is likely that they are mostly private, as the cases in many particle interaction simulation programs where small amount of data is shared among neighboring processing nodes (please refer to Figure 4).
- **Determine Shared Data:** Data that can not be identified as belonging to the above two categories is classified as shared.

As can be seen from the above rules, detecting private data is conservative, since any pointer in the reference list that implies shared or practically private classification will overwrite the ones that imply private classification. Determining practically private, however, is optimistic. The data classifying rules are designed to maximally expose optimization opportunities while guaranteeing correctness. This is also revealed from Algorithm 1, as will be described in Section 3.3.4.

Given the data classification methodology described so far, the data-parallel matrix multiplication example shown in Figure 3 (b) is revisited. Without loss of generality, assume that all the matrices are globally declared and as such cannot be identified as private. First, note that the variable *pid* is a TI variable. Since both *my_start* and *my_end* are expressions of *pid*, they are also

[1]This condition is slightly modified to determine practically private data for dynamic parallel programs, as detailed in Section 3.3.2

identified as TI variables using forward data flow analysis. Variables *my_start* and *my_end* further serve as loop bounds, making the index *i* a TI variable. Since the pointers *a* and *x* are accessed with the index *i*, matrices *a* and *x* are thus classified as practically private based on the data classification rules introduced above. Similarly, *b* is classified as shared. In this example, it is shown that the data classification scheme does not identify matrices *a* and *x* as private data. This conservatism is to guarantee correctness in the presence of complicated pointer usage, which makes global data difficult to be assured to be private. Furthermore, it is expected that by treating private data as practically private the performance will not be significantly degraded[2].

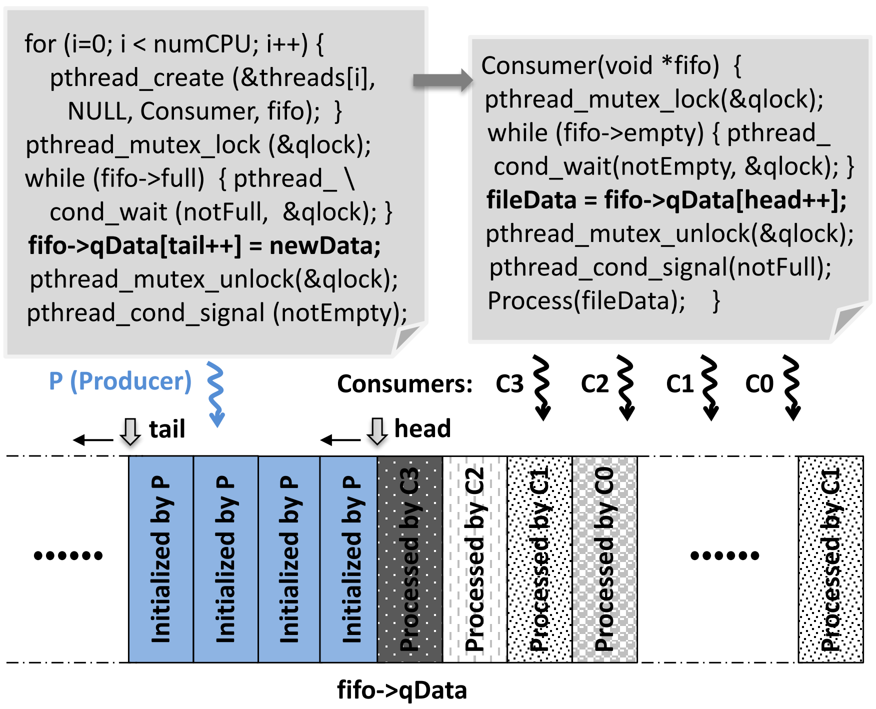
3.3.2 Programs with Dynamic Parallelism

Unlike data-parallel programs (e.g., parallel matrix multiplication) which statically partition workloads using thread IDs, applications with dynamic parallelism operate on their input by coordinating multiple threads based on certain scheduling policies. For example, the PARSEC benchmark X264 processes input video frames in a pipeline with the number of stages equal to the number of encoder threads. The threads communicate with each other to resolve inter-frame dependences and assign new frames to idle threads. Another example is the file compressor PBZIP, which utilizes a producer-consumer parallel model to handle input files dynamically. Although these applications represent a different paradigm than the data parallel model, studying them reveals that the concept of practically private still applies and the techniques introduced in Section 3.3.1 can be used to identify practically private data classification in these programs.

Compared to static data-parallel applications, which typically utilize thread IDs or other similar scalar variables to partition data, dynamic parallel programs often use TI pointers. Based on the notion of TI pointers described in Section 3.3.1, the classification rule can be extended for determining practically private data induced by TI pointers:

- **Determine Practically Private Data:** A heap data block is classified as practically private if at least one pointer in the reference list is of global scope and **that pointer is a TI pointer** or is dereferenced with a TI variable.

[2]In a directory based private cache organization, the only difference between private and practically private data is that private data allows a memory request to go directly to main memory upon a local last-level cache miss thus bypassing the coherence directory query.



```
for (i=0; i < numCPU; i++) {
   pthread_create (&threads[i],
     NULL, Consumer, fifo);  }
pthread_mutex_lock (&qlock);
while (fifo->full)  { pthread_ \
   cond_wait (notFull,  &qlock); }
fifo->qData[tail++] = newData;
pthread_mutex_unlock(&qlock);
pthread_cond_signal (notEmpty);
```

```
Consumer(void *fifo)  {
 pthread_mutex_lock(&qlock);
 while (fifo->empty) { pthread_
  cond_wait(notEmpty, &qlock); }
 fileData = fifo->qData[head++];
 pthread_mutex_unlock(&qlock);
 pthread_cond_signal(notFull);
 Process(fileData);    }
```

Figure 8: Practically private data in a program applying the producer-consumer parallel model

Figure 8 depicts a typical program with producer and consumer threads sharing a data queue. The producer examines the queue and puts initialized data at the tail of the queue as long as the queue is not full. Several consumer threads dynamically operate on the initialized data blocks from the head of the queue based on the current progress and the work remaining. Due to the head and tail updates in the atomic region (code between `pthread_mutex_lock` and `pthread_mutex_unlock`), memory blocks pointed to by *fifo->qData* are identified as practically private. In reality, the data queue in Figure 8 is only initialized by the producer thread and mostly accessed by one of the consumer threads, leading to the temporal private accesses illustrated in Figure 4(c). Although for simplicity only the example of one producer is illustrated, the analysis applies to the case where multiple producers exist. In that scenario, multiple producers individually generate and initialize work for a group of consumer threads. The temporal privacy, and as such the practically private property, of a data block still remains.

Figure 9 shows a pipelined parallel program with three stages and two queues (*q1* and *q2*), each of which is shared between two adjacent stages. Each queue stores data from the prior stage and

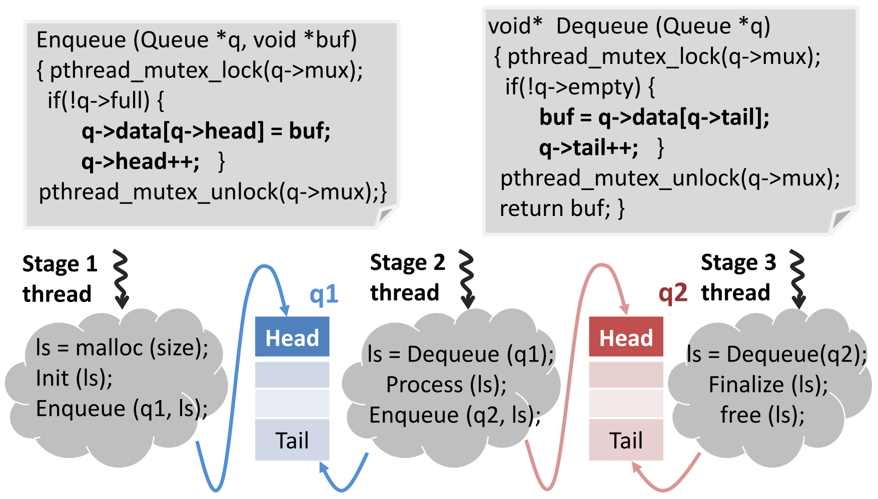


Figure 9: Practically private data in a program applying the pipeline parallel model

provides workload for the next. Accesses to the queues from threads in different stages are protect by mutex locks to avoid race conditions. The thread in stage 1 invokes the *Enqueue()* procedure to load data into $q1$ if the queue is not full. Once the data is loaded, the thread will never again access the data. On the other hand, the thread in stage 2 retrieves the data previously loaded in stage 1 and accesses the data exclusively (no other threads request the data at the same time). In this example, the data in the queues is shared by exactly two threads and the data accesses exhibit highly temporal privacy. The TI pointer based approach can detect this scenario by recognizing the TI pointers (e.g., $q\text{->}\ data$) and assert a practically private access pattern for the queue.

3.3.3 Data Classification for Other Parallel Programming Models

The concept of data classification including practically private applies to many parallel programming models beyond simple threads. For example, with a specifically extended method to identify TI variables, the data classification approach introduced in Section 3.3.1 can be utilized to detect practically private data in OpenMP programs.

In OpenMP programs, one requirement of a TI variable is that it has to be thread-private (either declared as default private or explicitly specified by the OpenMP clauses `PRIVATE`, `FIRSTPRIVATE` or `LASTPRIVATE`). A thread-private variable becomes a TI variable if multiple instances of the variable derive different values through mechanisms such as `OMP_GET_THREAD_NUM()` and OpenMP locks (`OMP_SET_LOCK()` and `OMP_UNSET_LOCK()`). Note that one special type of TI variable in OpenMP is the index variable of a loop following the directive `#pragma omp parallel for`. Once TI variables are detected, techniques introduced in Section 3.3.1 and Section 3.3.2 can be used to recognize practically private data. The other two categories in the data classification, namely private and shared, can be identified using the same approach introduced in Section 3.3.1.

It is worth mentioning that the `PRIVATE` clause in the OpenMP specification does not imply a private data classification defined in this chapter. This seeming incongruity stems from the semantic meaning of the OpenMP clause `PRIVATE`, which only dictates whether multiple instances of a variable should be created, one for each thread. Additionally, data scope attribute clauses in OpenMP are restricted to scalar variables, typically pointers, not arrays or objects. In this sense, specifying the scope of a pointer does not affect how many threads can access the data accessible through the pointer and thus its data classification. Therefore, the data classification approach introduced above is still necessary and valuable for OpenMP programs.

3.3.4 Data Classification Algorithm

The SUIF [106] infrastructure was used to implement the necessary compiler analyses for the proposed data classification. SUIF provides the Sharlit [98] framework to facilitate the implementation of data flow analyses. Using Sharlit the compiler first constructs a reduced flow graph for the source program based on an extension of Tarjan's fast path algorithm [97]. It then uses an iterator to traverse the reduced graph, calling user specified flow functions (e.g., the *Kill* and *Gen* described in Section 3.3) and applying meet rules (e.g., Eqs. (3.1) and (3.2)) at path joins, until solutions (e.g., reference lists and TI variables) are found. After the iterative process is complete, reference lists are attached to the nodes where pointers are dereferenced. In addition, every variable is identified either as a TI or non-TI variable. The flow graph is then traversed in another pass during which an action routine that checks the data classification rules at the array access points, such as $C[j]$ in Figure 5, is called at every node. The algorithm for the action routine is summarized in Algorithm 1, assuming $G = (V, E, r)$ is a directed graph with nodes V, edges E, and an entry node r. $RL(A)$ represents the reference list elements that are associated with pointer A.

Algorithm 1: Data classification algorithm on graph $G(V, E, r)$

```
begin
    for each node $b_i \in V$ do
        if $b_i$ is an array access or pointer dereference then
            $A \longleftarrow$ array base $o \longleftarrow$ array offset $L \longleftarrow \{malloc()\ m : m \in RL(A)\}$
            for each element $e_i \in L$ do
                if $e_i$ has been classified as shared then
                    if A is global && o is TIV then
                        re-classify $e_i$ as practically private
                    else
                        do nothing (stay shared)
                else if $e_i$ has been classified as private then
                    if A is global && o is not TIV then
                        re-classify $e_i$ as shared
                    else if A is global && o is TIV then
                        re-classify $e_i$ as practically private
                    else /*A is local*/
                        do nothing (stay private)
                else /*$e_i$ has not been classified yet*/
                    if A is global && o is not TIV then
                        classify $e_i$ as shared
                    else if A is global && o is TIV then
                        classify $e_i$ as practically private
                    else /*A is local*/
                        classify $e_i$ as private
        else
            continue;
    mark unclassified memory allocations as shared
```

3.4 EVALUATION

In this section, the accuracy of the compiler-based data classification approach is studied by comparing its results with classification from run-time profiling. The full system simulator Wind River Simics [71] is used to collect data classification information. The tested benchmarks are selected from the SPLASH 2 [5], PARSEC 2 [15] and RODINIA [21] benchmark suites, as detailed in Table 1. The data classification accuracy and effectiveness for all the benchmarks are evaluated to demonstrate that the proposed technique is generic and can handle a wide range of parallel applications with different parallel models (i.e., data-parallel, pipelined and producer-consumer) and frameworks (i.e., Pthreads and OpenMP).

Table 1: Benchmarks

Benchmark Suit	Benchmark	Input Workload	Application Domain	Parallel Model	Parallel Framework
SPLASH 2	BARNES	524288 particles;	Particle Simulation	Data-parallel	Pthreads
	OCEAN	1026x1026 matrix;	Movement Simulation	Data-parallel	Pthreads
	RADIX	10485760 radix;	Integer Sorting	Data-parallel	Pthreads
	FFT	2^26 even integers;	Mathematical Transform	Data-parallel	Pthreads
	CHOLESKY	input file *tk*23.*O*;	Factorization	Data-parallel	Pthreads
	RAYTRACE	input file *teapot*;	Rendering	Data-parallel	Pthreads
	WATER-SPATIAL	3000 molecules;	Molecule Simulation	Data-parallel	Pthreads
PARSEC 2	BLACKSCHOLES	20000 options;	Financial Analysis	Data-parallel	Pthreads
	STREAMCLUSTER	1024 data points;	Data Mining	Data-parallel	Pthreads
	SWAPTIONS	64 swaptions;	Financial Analysis	Data-parallel	Pthreads
	DEDUP	496K input;	Enterprise Storage	Pipelined	Pthreads
	X264	640×360 8 frames;	Media Processing	Pipelined	Pthreads
	PBZIP	$35K$ bytes;	File Processing	Producer-consumer	Pthreads
RODINIA-2.1	HOTSPOT	temp_1024 input	Physics Simulation	Data-parallel	OpenMP
	LEUKOCYTE	testfile.avi input, 3 frames	Medical Imaging	Data-parallel	OpenMP
	LUD	2048.data input	Linear Algebra	Data-parallel	OpenMP
	NW	8192 max_rows/cols, 10 penalty	Bioinformatics	Data-parallel	OpenMP
	SRAD	2048 rows and cols	Image Processing	Data-parallel	OpenMP
	BFS	graph1MW_6.txt input	Graph Algorithms	Data-parallel	OpenMP
	HEARTWALL	test.avi, 5 frames	Medical Imaging	Data-parallel	OpenMP

3.4.1 Compiler-based Data Classification

To demonstrate the effectiveness of the data classification methodology, Figure 10 shows the percentage of data accesses in each category during the application execution. Except for FFT, which is dominated by shared accesses, a significant amount of the accesses are practically private. For benchmarks such as BLACKSCHOLES, LUD, NW and HEARTWALL, practically private dominates the data accesses. On average, more than 60% of the data accesses are practically private. From the above results, it can be concluded that practically private is commonly present in parallel applications and has a potentially large impact on system scalability, efficiency and performance.

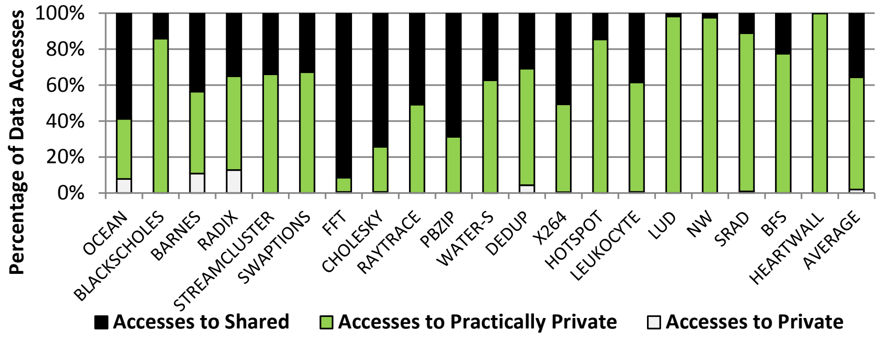


Figure 10: Percentage of accesses classified by the compiler as shared, practically private, and private

Figure 11 and Figure 12 show the actual sharing behavior at run-time for the data that is identified as practically private. Figure 11 reports the percentages of practically private data blocks with different numbers of sharers. For all the tested benchmarks, more than 50% of all the practically private data blocks are actually private. Most benchmarks exhibit predominantly private behavior across all the data blocks identified as practically private. On average, over 80% practically private data blocks are verified to be private. Figure 12 presents the percentages of data accesses to all practically private data blocks with one, two, three or more sharers. For many benchmarks the percentage of private accesses within practically private data is not as high as the corresponding percentage of private data blocks. For example, WATER-S exhibits over 70% private data blocks on which the accesses comprise only around 40% of the total accesses. This is because a data block with more sharers is likely to be heavily accessed, compared to a private block. However, private accesses from only one core still contribute an average of 77% of all the accesses on practically private data.

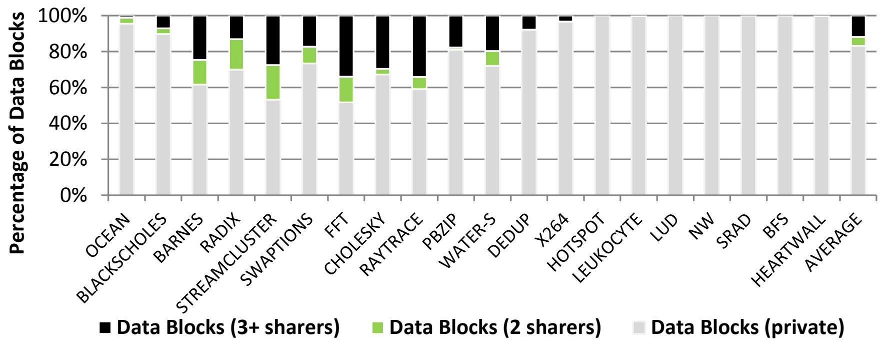


Figure 11: Percentages of data blocks classified as practically private that are accessed by one core (private), two cores, or three or more cores

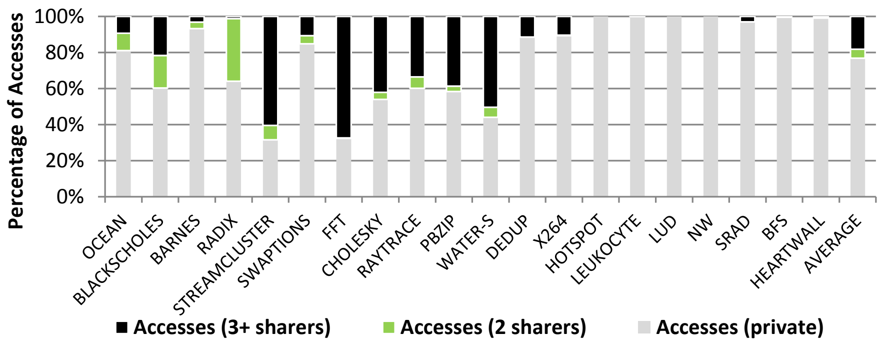


Figure 12: Percentages of accesses to the data blocks classified as practically private that are accessed by one core (private), two cores, or three or more cores

Figure 13 and Figure 14 report the runtime sharing behavior for the data classified as shared. On average, only 19% of the data blocks classified as shared turn out to be private while more than 80% of the compiler classified shared data has two or more sharers. Compared to the percentage of data blocks, the percentage of accesses further supports the classification accuracy. Figure 14 shows that only 5% of the data accesses occur on private blocks while the remainder occur on shared blocks. By comparing with the runtime sharing behavior of the practically private data it can be concluded that the shared data exhibit a predominantly shared access pattern, indicating an effective data classification by the compiler.

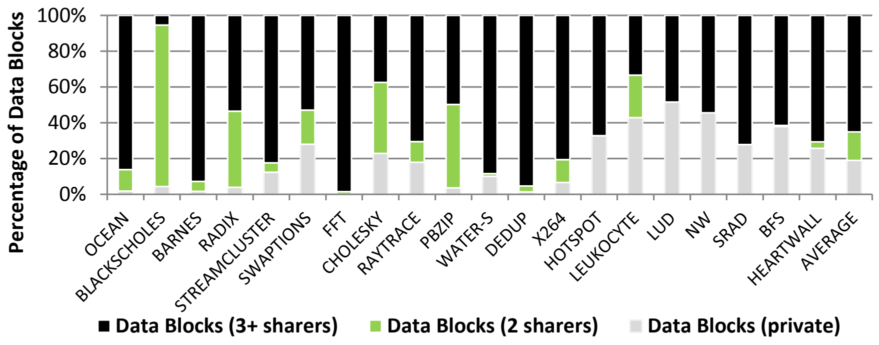


Figure 13: Percentages of data blocks classified as shared that are accessed by one core (private), two cores, or three or more cores

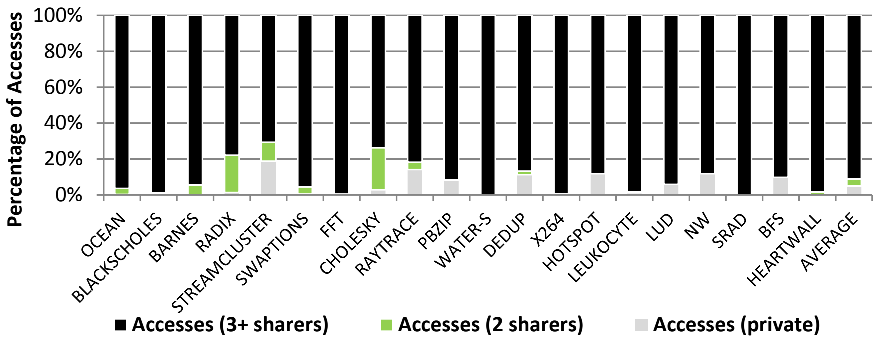


Figure 14: Percentages of accesses to the data blocks classified as shared that are accessed by one core (private), two cores, or three or more cores

4.0 DATA CLASSIFICATION AWARE CACHE ARCHITECTURE

The performance of CMPs is largely limited by the latency of data accesses, which is highly dependent on the organization of its memory caches connected using on-chip interconnect. As the number of cores in CMP systems increases, the latency of the interconnect is becoming an even greater bottleneck. As a result, elimination of remote data accesses and localization of communication have been demonstrated to be crucial to the performance improvement [1]. A number of architectural techniques have been proposed to achieve this goal [25, 34, 20, 114, 58]. In general, these techniques aim at a compromise between the two basic cache organizations, namely the distributed shared caches [56] and the per-core private cache architectures [19]. The goal is promoting data proximity while efficiently utilizing the entire cache capacity with minimal coherence overhead. This chapter presents a cross-layer CMP architecture that leverages the data classification technique proposed in Chapter 3 to optimize on-chip coherence caches.

4.1 CUSTOMIZED MEMORY ALLOCATOR

For the underlying architecture to be data-classification-aware, the memory allocator is modified to store and pass the compiler instrumented classification information to the runtime system. Existing memory allocators such as `malloc()` obtain the starting address of the heap from the OS for the requested data block and maintain a list to keep track of allocated as well as free memory blocks in virtual heap space. Each newly allocated block is filled with meta information within its header including the block size (S) and allocation status (A) of the block, as illustrated in Figure 15. The memory allocator also optimizes the allocated blocks by reducing fragmentation through mechanisms such as eight-byte alignment, splitting and block coalescing.

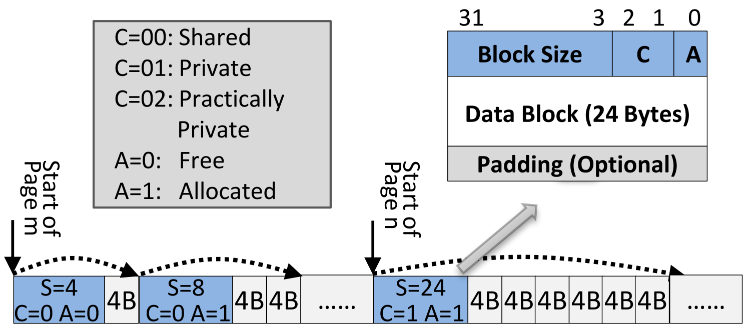


Figure 15: Data blocks maintained by the memory allocator

A similar mechanism can be used to provide support for efficient sharing of information between the compiler and architecture. To inform the hardware of the data classifications identified by the compiler, the prototype of `void *malloc(size_t size)` is extended to: `void *malloc(size_t size, classification_t cls)`. The `size` parameter is retained from the original version of `malloc()` and denotes the size of the requested data block. The second parameter `cls` is automatically filled by the compiler with the identified data classification. Upon an allocation, the invoked memory allocator adds the data classification information to the header of the newly allocated memory block, now including a 2-bit `C` field in addition to the original meta information. To facilitate runtime utilization of the classification information at a page granularity, the modified memory allocator aggregates blocks with the same classification into the same pages, as illustrated in Figure 15. In other words, data blocks with different classifications are not permitted to be allocated within the same page. Each page table entry (and the corresponding TLB entry) is also augmented with the data classification information. During the virtual-to-physical address translation, the classification information in the page table entry is retrieved with the physical address and can be utilized by the runtime system.

4.2 DATA CLASSIFICATION AWARE CACHING

This section describes how the data classification can be utilized by a CMP architecture with supporting cache policies. Figure 16 depicts how the identified data classification information can be utilized in a typical CMP microarchitecture in which each node has a processing core, caches, TLBs, etc. L1 instruction and data caches are local and private to each core. Physically, the L2 is tiled and distributed. Logically, it could be either private to its local node or shared among all nodes, depending on the data classification of the served L2 data block. To support caching both private and shared data, each cache block (in both L1 and L2) is augmented with an additional two-bit field, *class*, indicating its data classification. The *class* field of each cache block is filled with the classification information from the corresponding TLB entry during the address translation process. Whenever a cache block needs to be searched, placed or written back, the cache controller consults the *class* field. This creates an illusion that both private and shared cache blocks are respected in their favored cache organizations, private and shared caches, respectively.

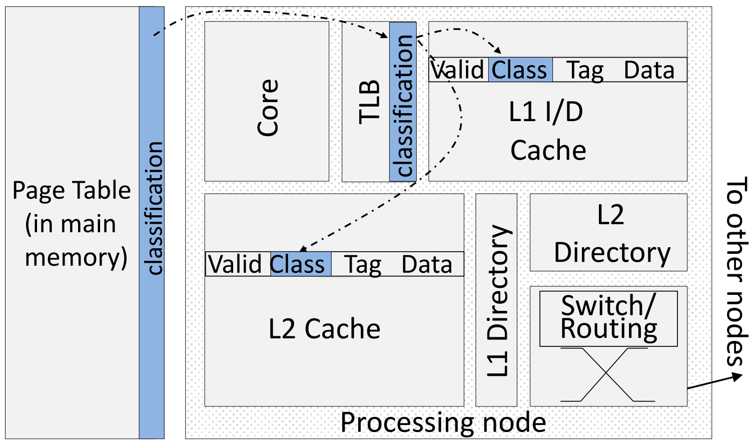


Figure 16: Architecture organization for data classification aware caching

As in a distributed shared cache, an in-cache directory [112] is used to maintain the private L1 coherence using the MESI protocol. Another on-chip sparse directory is provided to handle the L2 coherence for only the practically private data. This saves significant directory entries, compared to the traditional private cache in which each line requires a coherence directory entry.

In particular, the placement and search policy after an L1 miss is described as follows:

- **Shared:** Shared data is statically distributed across all the cache tiles as a function of its physical address. This keeps a unique copy at a fixed location to maximize the effective cache capacity, simplify data search, and avoid the coherence issue at the L2 level. Each shared cache line in L2 is associated with an entry in the in-cache directory to maintain the L1 coherence.

- **Practically Private:** Practically private data is likely to be accessed as private data. Thus, the local core would retrieve the data from the local cache tile. However, because it is not guaranteed to be private, it might be accessed by other cores. To ensure correctness while simultaneously promoting locality, this type of data should be placed within the local cache tile of the requester (e.g., first touch access). The MESI protocol can be adopted to maintain data coherence among potential sharers, as performed in a traditional private cache organization. For data that is shared by two or more cores this can reduce the effective cache capacity and increase coherence overhead. However, due to the implication of practically private (see Section 3.2), this type of sharing is infrequent and does not significantly harm the cache capacity and coherence.
- **Private:** Private data blocks are typically accessed by only one core and thus cache coherence actions can be saved for improved performance and efficiency. For example, on a L1 miss of a private access, the local L2 cache bank is directly checked for the requested data. Upon a L2 miss the data is directly obtained from main memory and filled into the cache as if it were a private L2 scheme without a coherence directory[1].

4.2.1 Classification Aware Coherence Protocol

Considering the interaction among cache blocks with distinct classifications, the placement, eviction and coherence behaviors need to be carefully handled so that cache blocks of all types are respected. To illustrate the proposed scheme, a number of examples are shown in Figure 17, where the classification of a cache block or memory request is indicated by its fill color and the MESI state is represented as a letter within a parentheses attached to the cache block. Dashed and solid lines are used to distinguish the eviction (including replacement, write back, and directory notification) and data lookup (including data search, directory lookup, and data reply), respectively. The examples illustrate from where the requested data is fetched upon a local L1 miss for various data classification scenarios. To simplify the illustration the MESI state changes are omitted.

Figure 17(a) illustrates a *requester* issuing a practically private memory request that misses in its local cache (L1 and L2) and fetches the block from the owner's L2 after evicting the shared

[1]A side-effect of this approach is that data elements that are privately accessed by different cores can co-exist in a cache line, potentially resulting in *false sharing*. Section 4.2.2 presents several solutions to address false sharing.

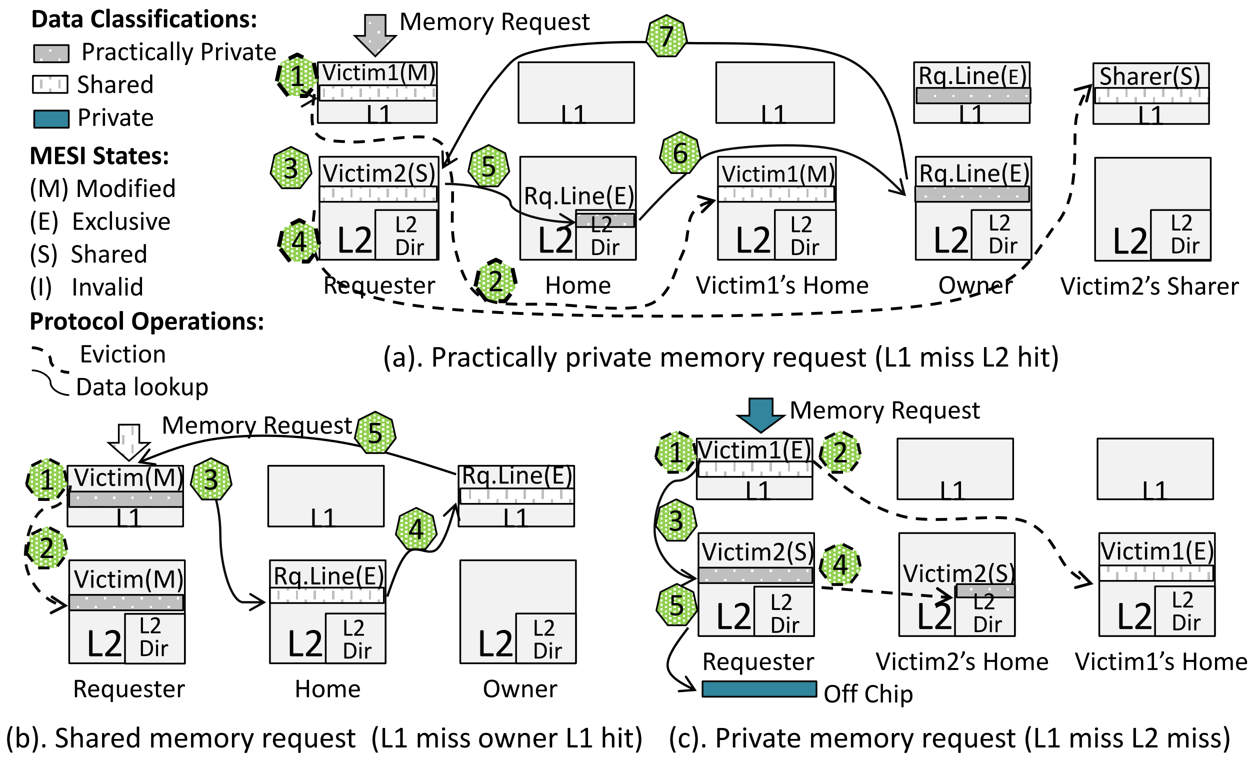


Figure 17: Examples of data flow and the coherence protocol for different data classifications

victim1 in the (M) state and the shared *victim2* in the (S) state. To accomplish this (1) the *requester* chooses *victim1* for replacement and (2) because *victim1* has shared classification, writes back to *victim1's* home, which is determined by *victim1's* physical address. At step (3), the local L2 is probed for the requested block since the request is practically private. Upon an L2 miss, *victim2* is selected for replacement. Because *victim2* is shared, the block and all its sharers are evicted at step (4). Note this eviction is not a necessary component of the coherence protocol. Rather, it is a standard procedure to maintain the inclusion property [6], which simplifies the cache coherence and allows the states of L1 sharers to be stored with the corresponding L2 tags[2]. After the eviction, the directory at the home node of the originally requested data is searched in step (5) and the message is forwarded to the owner in step (6). The owner is the only node that has a valid copy of the requested block when the block is in the (E) or (M) state. Any cache can be the owner if the directory indicates the block is in the (S) state. Finally, step (7) returns the requested block to the requester. The coherence states are updated to (S) upon a read, and to (M) and (I) as appropriate upon a write.

Figure 17(b) explains the situation when a shared memory request replaces a practically private block in the (M) state and then obtains the block from the owner's L1. Steps (1) and (2) show the victim replacement and write back. Since the request is shared, the home node is checked in step (3) and the in-cache directory indicates the owner has the only valid copy in its L1 in step (4). Step (5) returns the block to the requester. If the requested block is in the (M) or (S) in the directory, the block can be directly returned to requester without step (4).

In Figure 17(c), a private memory request replaces the shared *victim1* in the (E) state, determined in step (1), resulting in a directory update (step (2)). Since the request is private, the block is either in the local L2 or off-chip. A miss in L2 will trigger an access to main memory. In this example, a practically private *victim2* in the (S) state is replaced (step (3)) and the distributed directory at *victim2's* home is notified to make necessary changes (e.g., remove *victim2* as a sharer) in step (4). Finally at step (5), the requested line is fetched from main memory and placed in the local L2 bank.

[2]The amount of evictions is a function of the last level cache capacity and thus, it is only affected by cache block replication, not by the data classification.

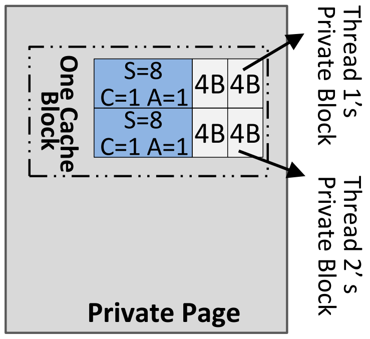


Figure 18: False sharing in a private page

4.2.2 Addressing False Sharing for Private Data

Multiple private address locations accessed by different cores residing in the same cache block are susceptible to false sharing, as illustrated in Figure 18 (the C, S and A fields have the same meaning as defined in Section 4.1). False sharing can be a severe problem in certain scenarios such as when a large number of small-sized private memory blocks are allocated by different threads in a non-contiguous fashion. Thread migration can also result in false sharing issues where a private data block is detected as shared since it is accessed by a different core after migration.

False sharing defeats data-classification-aware caching policies (see Section 4.2.1) that eliminates coherence overhead for private memory blocks and when improperly addressed can result in incorrect system function. Imagine that a falsely shared block is first modified by core 1 and cached in its local cache. When core 2 accesses the block, the system could be misled by the block's private classification and as such, not be aware of the valid copy in core 1's local cache. False sharing problems can be addressed using previously proposed approaches such as utilizing per-word valid bits in cache blocks [50, 82], at the expense of considerable hardware overhead. Alternatively, false sharing can be addressed by enforcing a cache block size alignment when allocating private memory blocks. In particular, if the memory allocator is aware of the thread allocating a particular memory block (e.g., by keeping an independent memory pool for each thread or annotating each private block with its owner), it can avoid placing private data blocks from different threads into the same cache block.

As the compiler classification is conservative in detecting the private class and optimistic in detecting practically private class, it was observed that for all the studied benchmarks, shared and practically private classifications dominate the data accesses while the amount of private data is small (see Section 4.3). Thus, the simple solution that allocates private data blocks with cache block size alignment can be adopted without significantly affecting the memory utilization. Specifically, when allocating a data block, the allocator detects that the block is private and then a block with a minimum size of a cache block is allocated. With this allocation scheme data blocks private to different threads can never reside in the same cache block, thus eliminating false sharing.

4.3 EVALUATION

In this section, the classification aware CMP design is compared with two baseline cache organizations (i.e., distributed shared and private) and a state-of-the-art, runtime page-level data classification mechanism R-NUCA[3] [42]. Wind River Simics [71] is used as the simulation environment and the relevant caching schemes are implemented as modules within the simulator. The target architecture is a tiled CMP consisting of 16 SPARC 2-way processors laid out as a 4×4 mesh. The detailed architecture parameters are presented in Table 2.

Table 2: Architecture configurations

Processor	16 SPARC cores, 2G Hz, 2 issue width
Operating System	64-bit Solaris 10
L1 Cache	32KB/core, 4-way associative, 64B block size, 1-cycle hit latency, write-back
L1 Coherence	MESI protocol
L1 Directory	L2 in cache directory, 3-cycle hit latency
L2 Cache	8/16M, 32-way associative, 64B block size, 5-cycle hit latency, write-back
L2 Coherence	customized protocol (based on MESI, data-classification aware, see Figure 17)
L2 Directory	sparse directory, 3-cycle hit latency
Network	4×4, packet switching, X-Y routing, 3-cycle per-hop latency, link reservation for contention
Main Memory	4GB, 150-cycle latency

[3]Only the data classification component of R-NUCA is simulated as the code page replication and clustering is orthogonal to data classification and can be applied to many caching schemes including this one.

To evaluate the data classification aware CMP design, a representative set of benchmarks are selected from the SPLASH 2 [5] and PARSEC 2 [15] benchmark suites, as listed in Table 3. The size of the dataset can bias the results toward a particular type of cache, one that favors private–i.e., a small dataset/working set that can easily fit into the cache even with significant replication, and one that favors shared–i.e., a larger dataset/working set where cache capacity is at a premium. Unfortunately, the simulation problem size was limited due to the intractably long simulation time of large workloads. To simulate these two conditions, two configurations, *shared-averse* and *private-averse*, are employed.

Table 3: Benchmarks

Benchmark Suit	Benchmark	Input Workload	
		Shared-averse(16M)	Private-averse(8M)
SPLASH 2	BARNES	524288 particles;	1048576 particles;
	OCEAN	1026x1026 matrix;	2050x2050 matrix;
	RADIX	10485760 radix;	104857600 radix;
	FFT	$2^{2}6$ even integers;	$2^{2}6$ even integers;
	CHOLESKY	input file *tk*23.*O*;	input file *tk*29.*O*;
	RAYTRACE	input file *teapot*;	input file *car*;
	WATER-SPATIAL	3000 molecules;	27000 molecules;
PARSEC 2	BLACKSCHOLES	20000 options;	200000 options;
	STREAMCLUSTER	1024 data points;	1024 data points;
	SWAPTIONS	64 swaptions;	512 swaptions;
	DEDUP	496K input;	3516K input;
	X264	640×360 8 frames;	640×360 32 frames;
	PBZIP	35*K* bytes;	100*K* bytes;

In the shared-averse configuration, the aggregate L2 cache capacity is configured as 16M and the smaller of two workloads is selected. This favors private caches as the application can still remain relatively cache bound even if a significant amount of data replication occurs. To simulate a private-averse system the cache size is reduced to 8M bytes to make the overall capacity a bigger factor and used the largest possible workload. All other system parameters for shared-averse and private-averse configurations are the same, as shown in Table 2.

4.3.1 Effect on Coherence Traffic

Traditional coherence protocols incur large volumes of coherence messages, especially for heavily accessed data with numerous sharers. This impedes the scaling of future many-core CMPs. Compiler classified caching significantly reduces the number of coherence messages since it only maintains the coherence for data identified as practically private, which are likely to have few or no sharers. Figure 19 reports the percentage of reduced coherence traffic compared with private caches with the MESI coherence protocol. The compiler-based classification technique eliminates between 11% to 78% of the coherence traffic, depending on application. On average, coherence traffic is reduced by 46%.

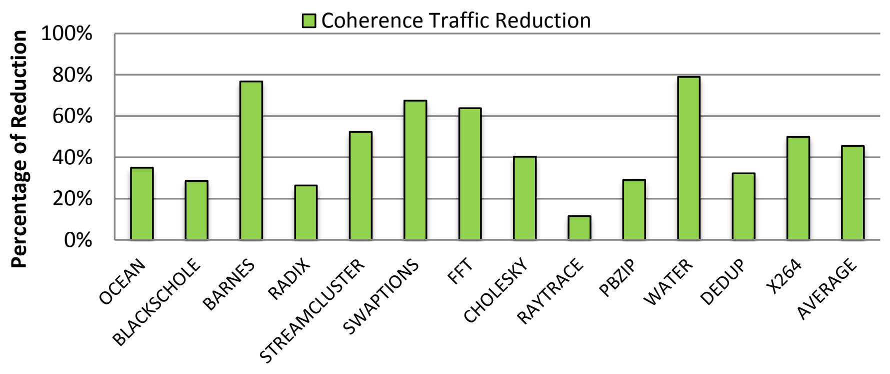


Figure 19: Percentage of coherence traffic reduced compared to private caches

4.3.2 Performance Evaluation

To evaluate the performance impact of using compiler-based data classification, the presented caching scheme (denoted as PSP) is compared with distributed shared [56], private [19] and R-NUCA [42] caches in terms of cache miss rate, average memory access latency and speedup.

4.3.2.1 Miss Rate Figures 20 and 21 show the L2 cache miss rate for shared-averse and private-averse configurations, respectively, each normalized to the distributed shared cache. In general, distributed shared caches have the lowest miss rate because replication is not allowed and as such, the cache capacity is used most effectively. Conversely, in the private cache organization, multiple cache blocks become replicated and consume more capacity, typically resulting in a higher miss rate. R-NUCA has an undesirable miss rate for some benchmarks, especially those exhibiting low raw misses, due to the impact of its page re-classification mechanism on cold start misses. When a page initially classified as private is re-classified as shared, all the cache blocks within the page that have been cached must be invalidated, resulting in a higher miss rate although the total number of misses could be quite low. For longer running applications with larger number of iterations R-NUCA's re-classification misses can be largely amortized and in such a scenario access latency is more critical than the reclassification miss rate.

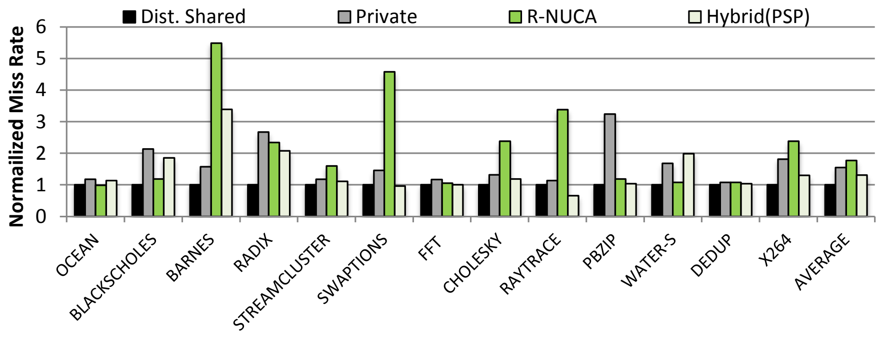


Figure 20: Miss rate for the shared-averse configuration

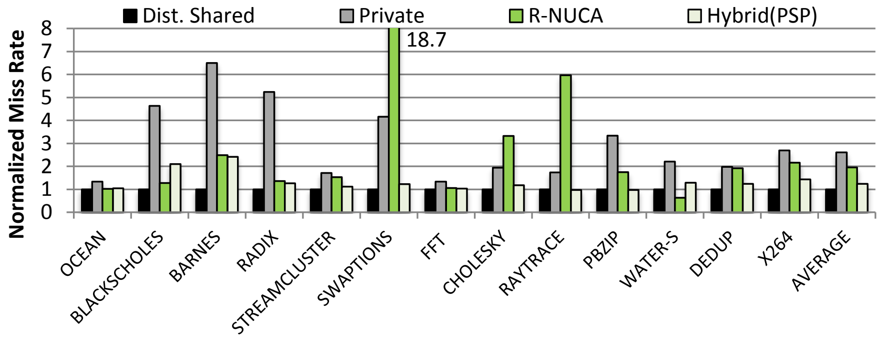


Figure 21: Miss rate for the private-averse configuration

4.3.2.2 Latency Average memory access latencies of all relevant schemes are reported in Figures 22 and 23. Data access latency is affected by both miss rate and on a hit, the distance that must be traversed to retrieve the data from a potentially remote tile. In distributed shared caches, most data is stored in a remote tile from the core that heavily accesses it, resulting in a higher latency, especially in the shared-averse configuration. In contrast, private caching absorbs all the data to the local tile and thus, has a lower hit latency because off tile cache accesses are minimized. This is true especially when the working set size does not exceed the cache capacity, as shown in Figure 22 for the shared-averse configuration. As the cache capacity is pressured by an increasing working set, the latency is dominated by off-chip misses and the performance begins to degrade, as demonstrated in Figure 23.

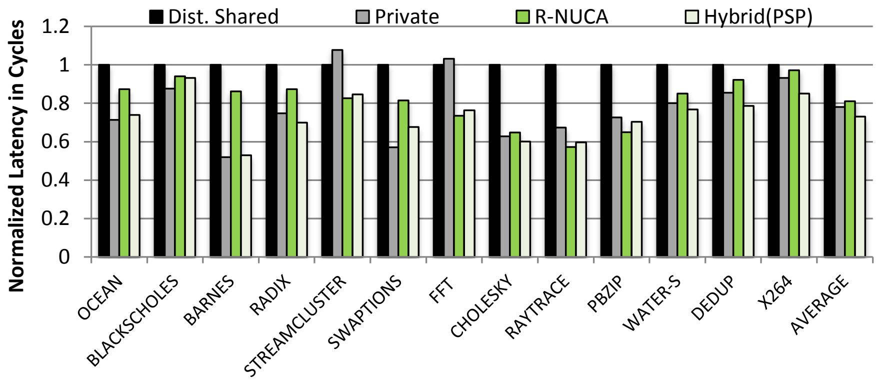


Figure 22: Average memory access latency for the shared-averse configuration

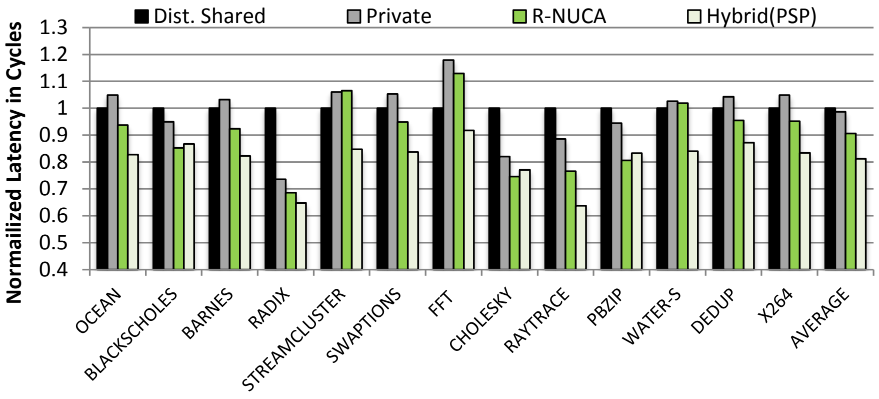


Figure 23: Average memory access latency for the private-averse configuration

R-NUCA reduces the access latency when pages are initially accessed locally and remain private to a particular processor. However, R-NUCA suffers from a relatively high access latency similar to distributed shared caches when the pages are classified as shared. Another problem of the data classification mechanism used in R-NUCA is that the OS page granularity makes it impossible to optimize smaller memory blocks. One byte of shared access in a private page results in the whole page being re-classified as shared. Additionally, R-NUCA is an “all-or-nothing” approach. For a single access by another core the page is classified as shared even if this is an uncommon or one time occurrence (e.g., data initialized by the main thread but used by another working thread).

The compiler-assisted caching addresses these problems through customized placement policies for classified data, packing a page with data of the same classification, and for tolerating a stray shared access in a practically private configuration. In the shared-averse configuration, PSP reduces memory latency for distributed shared, private and R-NUCA by 27%, 7% and 10%, respectively. In the private-averse configuration, the corresponding latency reductions are 19%, 18% and 10%.

4.3.2.3 Performance Improvement Figure 24 shows that in the shared-adverse configuration PSP outperforms distributed shared caches by 12% while still providing noticeable gains over private (3%) and R-NUCA (5%). These gains are from keeping the data local to the core(s) that use it and a reduction in coherence traffic. R-NUCA’s performance suffers from data PSP classifies as practically private being categorized as shared and requiring longer access latency than PSP. Figure 25 indicates that for large working set sizes (private-averse configuration), PSP outperforms shared, private and R-NUCA caches by 9%, 8% and 4%, respectively. Thus, a conclusion can be drawn that PSP provides benefits of leveraging application specific behavior with a global viewpoint not possible with hardware-based techniques.

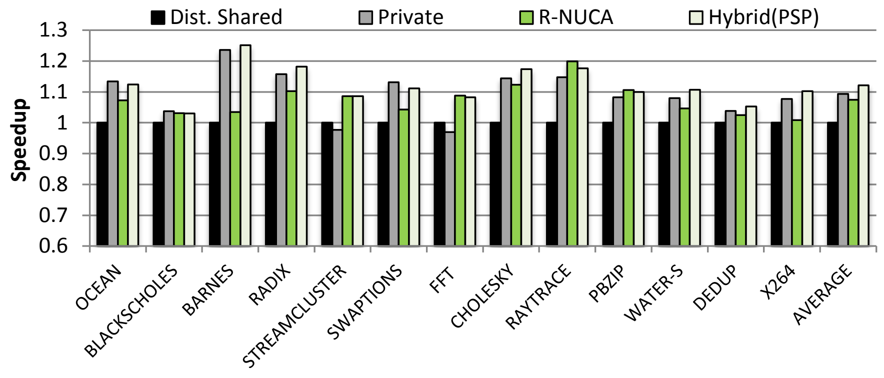


Figure 24: Application speedup for the shared-averse configuration

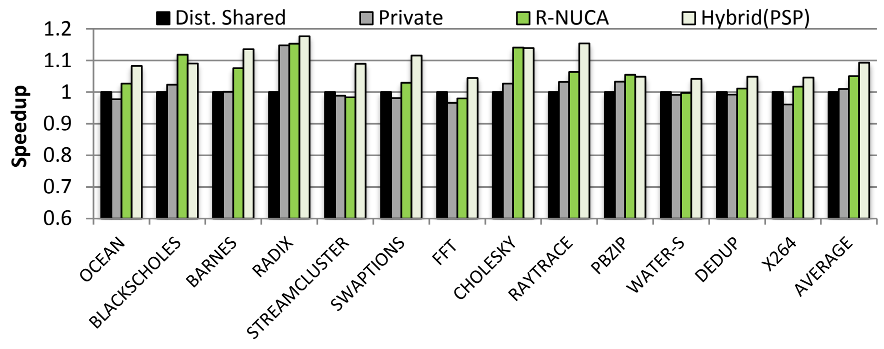


Figure 25: Application speedup for the private-averse configuration

5.0 TLB OPTIMIZATION USING DATA CLASSIFICATION

Translation-lookaside buffers (TLBs) have been shown to be effective in accelerating virtual to physical address translation. They are a standard component of commodity CPU products, including multi-core systems such as the AMD Opteron and Intel Sandy/Ivy Bridge processors. To date, these processors use multi-level per-core TLBs that are exclusive and private to each core. The primary reason for this private design is due to the timing-critical nature of the TLB, which is on the critical path of cache/memory accesses. In addition, private TLBs simplify translation lookup in multi-program environments. In multi-core systems, private TLBs also lend themselves to lightweight methods of addressing TLB consistency problems. For example, in contrast to caches utilizing complicated coherence protocols, translation coherency can be maintained using TLB shootdown, a process that evicts invalid TLB entries using Inter-Processor Interrupts (IPIs) [104]. This mechanism leverages the infrequency of translation entry changes in the TLB compared to data changes in the memory system.

Unfortunately, the merits of private TLBs are often traded for an increased translation miss rate. This increase is typically due to poor TLB capacity utilization of private TLBs from replicated entries. Additionally, the inability to locate entries cached in remote TLBs due to the lack of coherence results in potentially unnecessary TLB misses. These tradeoffs are particularly undesirable for architectures such as Intel IA-32, in which filling up a TLB entry upon a miss may require up to four memory accesses traversing a four-level hierarchical page table structure, which is expensive.

In addition to the above issues, traditional translation operations such as TLB shootdowns [16] and TLB flushes caused by context switches exacerbate these inefficiencies. These operations create significant off-chip translation misses, further adding to the aggregate translation penalty.

Thus, to provide a scalable alternative to a physically shared TLB without requiring the latency and storage overhead of a tagged solution [102], this chapter introduces a novel design called the *partial sharing TLB (PS-TLB)*. PS-TLB extends traditional private TLBs with a small *partial sharing buffer (PSB)*. Assisted by compiler-oriented or OS page-level data classification schemes, private translations are placed locally for low-latency access and shared translations are distributed across all cores' PSBs in a similar fashion as non-uniform access memories. However, unlike the data-classification-aware caching discussed in Chapter 4, PS-TLB does not require a fine-grained or speculative data classification mechanism such as the one used in Chapter 3 to prevent data classification pollution. This is because translations entries are accessed and shared by processing cores at the page granularity. Thus, the translation classification mechanism used in PS-TLB may be a simple and efficient OS-assisted page classification, similar to the one used in R-NUCA [42].

PS-TLB provides a significant performance improvement over state-of-the-art second level TLB techniques that leverage translation sharing while reducing storage and runtime overheads. PS-TLB also optimizes other TLB functions. For example, using PS-TLB, TLB shootdowns can often be improved by downgrading them to individual invalidations, preventing stalls in cores not utilizing that translation. Additionally, like tagged TLBs, PS-TLB can reduce the impact from flushing required in private TLBs during a context switch, by retaining entries in the PSBs, which do not require flushing. As such, this chapter demonstrates that PS-TLB reduces several categories of translation misses and overhead, as classified below:

TLB capacity misses Private TLBs duplicate requested entries that are shared by multiple cores, leading to capacity misses for workload sizes that exceed the capacity of local TLBs.

TLB sharing misses Even with adequate capacity, a private TLB structure still suffers from unnecessary misses since it is not aware of non-local requested entries already cached in other processor TLBs.

TLB coherence overheads TLB shootdown may unnecessarily stall all cores to invalidate a private page and TLB flushing may unnecessarily evict translations from multiple processes running on the same core.

The impact of PS-TLB is evaluated in terms of translation latency, translation miss rate, system performance, TLB shootdown and TLB flush effects. Experimental results demonstrate a 45%

latency reduction and an application performance improvement of 9% compared to the state-of-the-art TLB mechanism that leverages sharing [12]. Sensitivity analyses are conducted to show that PS-TLB is effective for TLBs with different sizes and performance scales well with the size of the sharing buffer. Finally, experiments show a considerable reduction of TLB shootdown and context switch misses.

5.1 MOTIVATION

Address translation and TLB handling are known to consume a considerable amount of system running time [101, 83]. This can be attributed to the need for address translation to occur in the critical path for all memory accesses (including instructions). With uniprocessor systems, researchers have shown that TLB related overhead can be as high as 40% of the total running time [46] and a wide range of literature has been produced to mitigate this potential bottleneck.

With the advent of shared memory chip-multiprocessors, many inherited assumptions from uniprocessor TLBs lead to new inefficiencies. In particular, multi-threaded parallel workloads exhibit ubiquitous sharing. Figure 26 shows that for representative multi-threaded benchmarks [17, 5, 15], an average of 62% of all pages are heavily shared (i.e., with three or more sharers) and only 29% of them are private. This heavy sharing behavior is due to the fact that large granularities increase sharing due to page-level false sharing. Thus, an entirely private per-core TLB leads to the potential for severe performance degradation due to avoidable misses from poor capacity utilization and lack of awareness of on-chip shared entries cached in remote private TLBs.

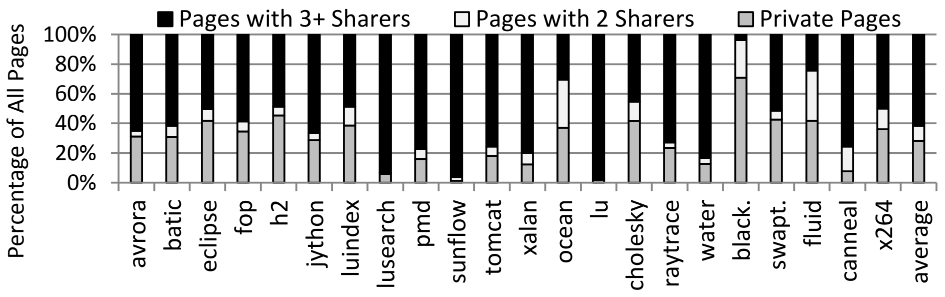


Figure 26: Application page sharing characteristics

Recently, a physically shared (or centralized shared) last level TLB has been shown to improve TLB translation latency in a multi-core context with four cores [12]. Unfortunately, a physically shared solution is not expected to scale well to a large number of cores. Scaling will be hampered by increases in end-to-end latency of accessing a shared TLB and the increased pressure on a shared L2 TLB caused by a higher aggregate of L1 TLB misses due to adding more cores into the system.

A potentially scalable solution is to use a static non-uniform TLB access architecture similar to the S-NUCA concept used in last-level caches [56]. This requires adding process ID tags to the L2 TLBs [102], a technique already proposed to avoid flushing the TLB on a context switch. The details of this approach are further described in Section 5.3.1. For 16 cores, the access latency of such a distributed-shared TLB is compared with a physically shared approach, which also requires tags, in Figure 27. In most cases the distributed shared TLB latency is higher due to the latency of using the NoC. This doubles the latency on average[1]. Thus, as the physically shared solution outperforms both private TLBs [12] and distributed shared TLBs (Figure 27), it is used as the baseline for performance comparison in the rest of the chapter.

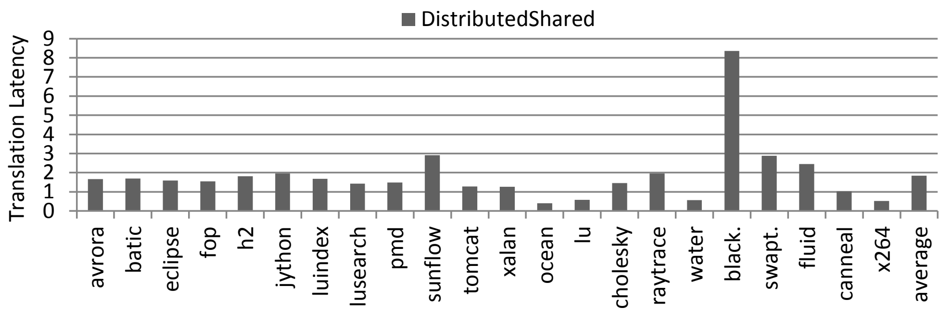


Figure 27: Translation latency for a L2 TLB using a non-uniform access shared TLB model compared with a centralized shared approach for 16 cores (normalized to centralized shared)

This analysis in part demonstrates that the creation of an efficient and scalable last level TLB architecture that leverages sharing of translations is a difficult problem. Unfortunately, the scalability of the physically shared solution is not expected to reach into many-core architectures. Recognizing this concern, researchers from the same group that proposed the physically shared TLB

[1]For this comparison the previously used access latency for a four-core system [12] was used. A detailed list of parameters is contained in Section 5.4.

solution also proposed a distributed solution that uses prediction to prefetch shared translations from other private TLBs [14].

In contrast, PS-TLB described in this chapter is scalable, uses less resources and outperforms both the leading shared [12] and distributed [14] solutions.

In particular, this chapter presents the following contributions to the literature:

- This chapter demonstrates the inefficiencies of using purely traditional, private TLBs and shared TLBs over 20 benchmarks. Experiments demonstrate that fast, efficient and scalable translation cannot be simultaneously achieved without considering translation classification in TLB designs.
- This chapter describes PS-TLB, which combines private and shared translation classification with an efficient translation architecture. PS-TLB offers inter-core translation sharing for shared translations to reduce TLB misses while preserving the low latency feature of private TLBs to satisfy the timing-critical requirement of on-chip translations.
- Experiments demonstrate a significant performance improvement of PS-TLB over the-state-of-the-art methods to leverage page sharing in TLBs while also reducing complexity overheads.
- PS-TLB supports efficient TLB operations for dealing with coherence and mitigating the impact of context switching and TLB shootdowns.

5.2 BACKGROUND AND CONTEXT

To frame PS-TLB in the context of previous work, this section begins with an overview of a typical translation architecture found in commodity chip-multiprocessors (CMPs), followed by a detailed discussion of research efforts related to improving translation performance and a comparison between prior efforts and PS-TLB.

5.2.1 Background

As an overview of the virtual address translation process in CMPs this section describes a basic TLB architecture, methods for translation and standard approaches for maintaining consistency.

5.2.1.1 Address Translation Architecture

Figure 28 illustrates the address translation flow on a typical CMP microarchitecture in which each processing node consists of a CPU, a coherence directory, a MMU (memory management unit), a network interface, L1/L2 caches and L1/L2 TLBs. The TLB is organized as a per-core hierarchical structure that features separate L1 instruction and data TLBs backed by an *unified* L2 TLB, serving both instruction and data translations. As in the most common case of virtually-indexed physically-tagged caches, the virtual address (VA) issued from the CPU must be translated to a physical address before the requested data can be accessed from either the caches or main memory. The resolved translations are cached in the local TLBs to accelerate further translation requests.

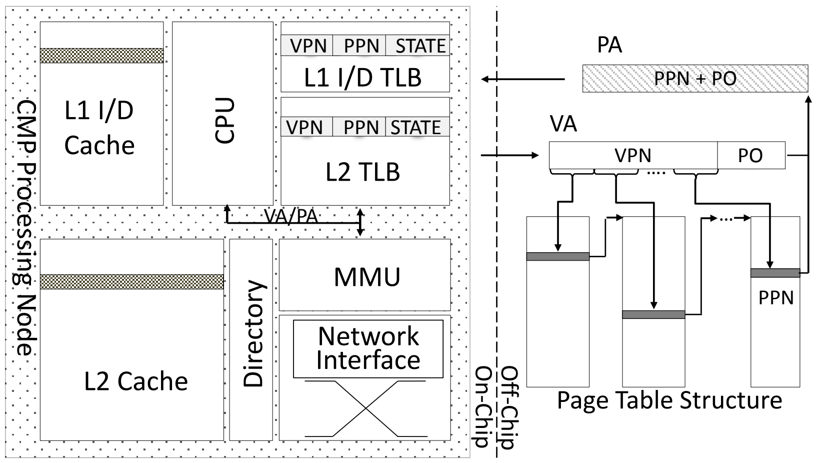


Figure 28: Baseline architecture with 2-level TLB translation and a hierarchical page table

5.2.1.2 Address Translation Basics

As the MMU handles a translation request, it first looks up the virtually indexed TLB hierarchy using the virtual address. Upon a last-level miss, either a hardware page table walker or a software TLB miss handler will be invoked to traverse all levels of the page table hierarchy for the target physical page number (PPN), as illustrated on the right hand side of Figure 28. The hardware mechanism, as adopted in the Intel x86 architecture, usually offers a performance benefit by preventing the pipeline from being polluted by the execution of a miss handler's code as occurs in a software-managed TLB. In contrast, software approaches reduce hardware complexity and enable more flexible page table structures that are largely independent of the underlying architectures (e.g., MIPS, SPARC). In both cases, a page table walk that traverses several page table hierarchies to locate the requested translation from main memory incurs a significant performance overhead. Even if all page table entries (PTEs) are present in the L2 cache, accessing them in a daisy chain fashion still incurs a penalty of several tens of cycles per TLB miss [10].

5.2.1.3 Address Translation Consistency

To ensure the translation consistency in CMPs, certain TLB operations are typically performed in response to modification of PTEs. One such operation is TLB shootdown [16], which is necessary in scenarios where unsafe changes [80] take place (e.g., page re-mapping, page swapping, decreasing page privileges, etc.). TLB shootdown is also recommended even for safe changes (e.g., dirty/access bit reset, privilege increasing, etc.) to avoid undesirable consequences. For example, choosing not to shootdown an entry upon the OS reseting the dirty bit in the corresponding PTE may result in the processor not setting the dirty bit again in response to a subsequent write access to the corresponding page. Consequently, the software cannot rely on the dirty bit being set as an indication that the page is dirty. Moreover, shootdowns typically require all cores to be stalled while the shootdown takes place even if the changed TLB entry is not stored within many of the cores' local TLBs.

Another important operation is TLB flush, an operation that invalidates all except global entries upon context switches. Flushing the TLB is necessary since TLB entries are indexed by virtual addresses from the local running process, which might overlap with those from another process running on the same core.

In summary, there are several inefficiencies from standard private TLBs used in CMP architectures. These inefficiencies include the inability to leverage shared TLB entries stored in remote private TLBs, necessity to flush TLB entries on a context switch, and system-wide stalls related to coherence.

5.2.2 Comparison with Prior TLB Proposals

Prior research efforts indicate that "tagged" TLB sharing can be accomplished by adding tags (e.g., context ID, process ID (PID) or address space ID (ASID)) to the private TLBs in order to associate TLB entries with specific processes [102, 14]. This prevents the need to flush the TLB upon a context switch. To reduce the coherence overhead of shootdowns, Villavieja et al. [104] extend the "tagged" concept to include a coherence directory to alleviate the need TLB shootdown overhead.

Based on a study of TLB sharing behavior, Bhattacharjee *et al.* recently proposed a TLB prefetching scheme [14] to reduce TLB misses. In this state-of-the-art distributed scheme, alluded to in Section 5.1, a *prefetch buffer* is used to avoid page table accesses for translations that miss in the private TLB. Using a technique called *leader-follower*, when a tile misses in the local TLB it becomes the "leader." The fetched TLB entry is sent to the prefetch buffer of other "following" cores that utilize the leader's pages frequently. A second technique, *distance-based cross-core*, matches the historical distance between TLB misses and predicts/prefetches TLB entries based on pattern matching. The system records the two distances between three successive TLB misses in a *distance table*. When two misses in any core match the first distance, the page matching the second distance is prefetched into the prefetch buffer.

Synergistic TLBs [93] utilize victim allocation and migration to improve the TLB hit ratio, similar to techniques applied to last-level caches. By leveraging victim entries in remote TLBs they can emulate a distributed shared TLB for increased capacity. Specifically, processing nodes are classified as *donors* versus *borrowers*, based on their TLB pressure, to achieve desirable sharing. Synergistic TLBs also allow translation migration and replication to reduce latency without harming TLB capacity.

All the above schemes provide a significant benefit for reducing TLB misses. However, these schemes have a significant overhead of adding tags/IDs to distinguish process IDs necessary for sharing entries between cores. Unfortunately, the storage overhead from adding tags can be significant, potentially requiring more than 25% additional storage. TLB Prefetching adds considerable additional complexity and overhead including complicated prefetching logic for pattern matching, prefetch buffers, distance buffers and a distance table, which contains hundreds of entries. In addition, TLB Prefetching requires $O(n^2)$ confidence counters to reduce bad prefetches (i.e., entries prefetched but never used) and performs a considerable amount of prefetch broadcasts when the TLB miss rate is high, which can lead to poor performance. Synergistic TLBs also rely on significant amount of hardware resource including access counters, saturation counters and complex policies for migration, replication and victim allocation.

To achieve a similar goal with lower hardware complexity, Bhattacharjee *et al.* recently propose a physically shared last-level TLB [12]. This system is demonstrated as effective for a four-core system, significantly outperforming private TLBs with the same overall number of translation entries. This approach represents the current best performing state-of-the-art shared L2 TLB architecture and as mentioned in Section 5.1. As such, it is used as the baseline in the performance comparisons for PS-TLB.

In contrast, PS-TLB distinguishes between shared and private TLB entries and uses a small tagged partial sharing buffer to retain the most heavily used shared translations on chip. All presented PS-TLB configurations use considerably less storage than a shared structure that requires tags[2]. Compared with schemes such as Synergistic TLBs and TLB prefetching, PS-TLB requires considerably less complexity and avoids the hardware counters used for for migration and prediction. Yet, it retains fast (local) access for all translations, either private or shared. Additionally, by leveraging the translation classification support available in PS-TLB, the proposed scheme reduces the context switch and TLB shootdown overhead. In the next section, the PS-TLB is described in more detail.

5.3 PARTIAL SHARING TLB

As indicated by the prior discussion, the translation performance is dependent on a variety of factors including off-chip translation rate, on-chip translation latency and performance of other TLB operations. This section details the proposed PS-TLB, which leverages page classification information based on translation sharing to reduce off-chip translations while performing on-chip translations as fast as a traditional private TLB. Additionally, this section also discusses how optimized translation operations can be developed based on the PS-TLB architecture and their performance impact in different scenarios.

[2]Any shared TLB structure (either non-uniform access or physically shared) requires a mechanism, such as tags, to distinguish between translations for different processes/threads. This includes all of the previously proposed schemes presented here.

5.3.1 Sharing TLB Entries

Tagged TLBs, described in Section 5.2.2, can be used to create a shared last level TLB either using a distributed shared last level TLB similar to the mechanism used for sharing data in a distributed last level cache [56] or with a physically shared structure [12]. The two sharing mechanisms are quantitatively compared in Section 5.1. Figure 29 provides a comparison of distributed method (tagged shared) with private TLBs (both 16-core, 256-entry L2 TLB/core system). As shown in the figure, the 8% miss rate for a private TLB drops to less than 3% for a shared TLB. However, shared TLB significantly increases translation latency, as shown in Figure 30. In this configuration, most applications perform better with the private TLB while some perform better with the shared TLB. By contrast, with a 64-entry per core L2 TLB, most applications perform better on the shared TLB, as shown in Figure 31.

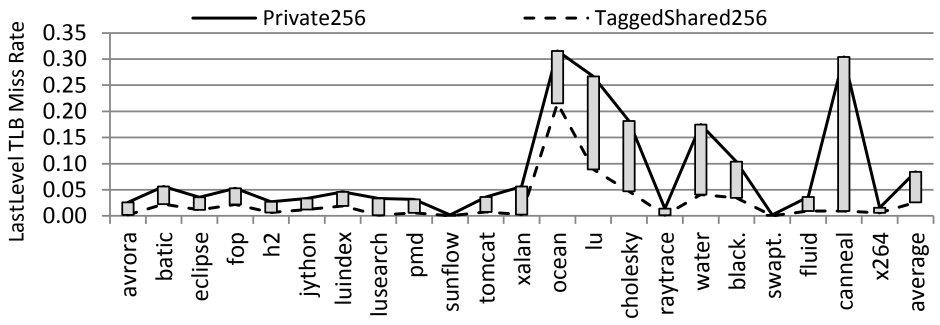


Figure 29: Last level TLB miss rate for private vs tagged shared TLB with 256 TLB entries/core

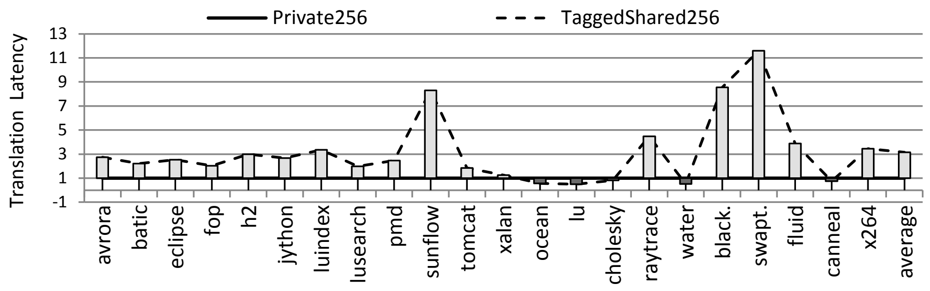


Figure 30: Latency for private vs tagged shared TLB with 256 entries/core (normalized to private)

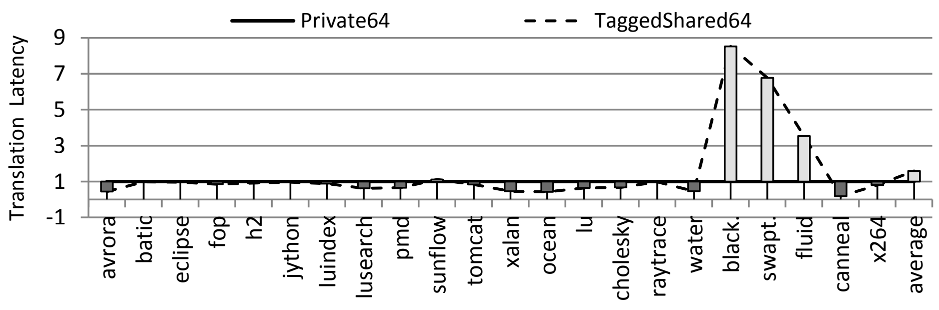


Figure 31: Latency for private vs tagged shared TLB with 64 entries/core (normalized to private)

Thus, a conclusion can be drawn that neither a simple, S-NUCA style shared TLB nor a traditional private TLB is always suitable for scalable distributed systems. A scalable last level TLB that includes benefits of both private and shared translation caching is required.

5.3.2 PS-TLB Architecture

The PS-TLB, shown in Figure 32, assumes a tiled CMP architecture where each core is locally equipped with a 2-level private inclusive TLB. This allows the design to inherit the merit of low translation latency of traditional private TLBs. To facilitate inter-core sharing, each core is augmented with a tile of a *partial sharing buffer* (PSB). The PSB serves as the local contribution of a global pool for page translation sharing. Compared to a TLB entry, a PSB entry has an extra field, *PID*, to distinguish translations from different processes, similar to the tagged shared L2 TLB. Even by adding a tagged PSB, an effective PS-TLB will still use less resources than a fully tagged shared L2 TLB with the same number of entries because the private L2 TLB entries do not require tags and the PSB is typically small. The tile where a particular PSB entry is placed is determined by selected bits from the virtual address. The PSB only accommodates translations that are shared by different cores. This prevents private translations from being placed remotely. It also eliminates the pollution of the PSB by private translations and increases the likelihood that shared translations are utilized by multiple cores as much as possible before being evicted from the PSB. The next section describes the method for translation/page classification.

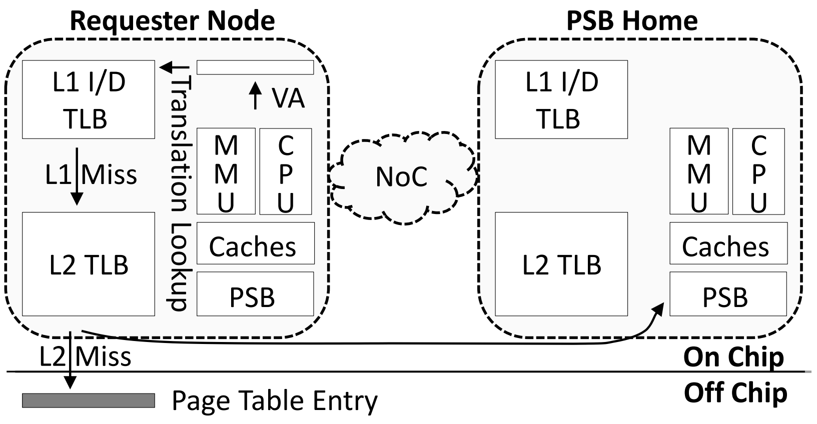


Figure 32: Partial sharing TLB organization and translation lookup

5.3.2.1 Translation/Page Classification Support The compiler-based approach from Chapter 3 can be leveraged to provide support for the private and shared classification for PS-TLB. In particular, the private and practically private data categories indicate translations that are private or almost-private and should be cached only in local TLBs. The share class implies scenarios where intensive data sharing is expected and thus the translation data should be copied to PSB for sharing.

A second approach is to employ an OS-based runtime classification mechanism. To classify pages as either private or shared, each PTE is extended with two fields, *FAC* (first accessing core) and *S* (shared), in addition to the existing *VPN* (virtual page number), *PPN* (physical page number) and *state* information fields. Figure 33 illustrates the structure of the extended PTE.

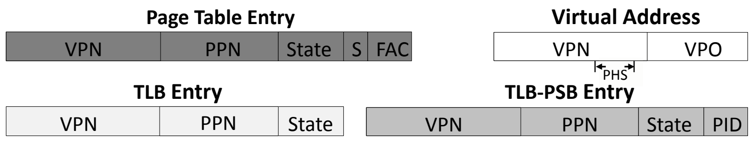


Figure 33: Structures for virtual address, TLB entry, PSB entry and page table entry(PTE)

On architectures such as MIPS and SPARC where TLB misses are processed by an OS interrupt handler, a page classification scheme similar to the one proposed in R-NUCA for caches [42] can be modified for use with TLBs. The OS initializes the *FAC* field with the first requester's core ID and clears the *S* flag, indicating that the page is private. On subsequent TLB misses, the OS handler checks the *S* flag to determine whether the accessed page has already been set as shared. If not, the OS compares the *FAC* in the PTE with the requester's ID and sets *S* in the case of a mismatch. On other CMPs (e.g., Intel x86), a TLB miss does not trap to the OS but rather invokes a hardware walker. In this scenario, the page classification can be easily completed by a hardware walker that is aware of the *FAC* and *S* fields in the page table. During a page table walk, the *FAC* and *S* fields of the target PTE are updated based on their previous contents and the current requesting core.

The page classification information can then be used to guide on-chip translation caching. The PS-TLB design only keeps the classification information in the page table, which avoids storage overhead in the TLBs, as shown in Figure 33.

To determine the location of a shared translation, the *PSB home select* (*PHS*) bits are selected from the virtual page number (VPN) field of the corresponding virtual address, as depicted in Figure 33. The number of bits required in *PHS* is $\log_2 n$ for a CMP with n cores. However, these bits are part of the virtual address and do not add storage overhead.

5.3.3 Basic Translation Operations on PS-TLB

Based on the PS-TLB architecture with the PSB and page-classification support, existing translation operations (e.g., TLB lookup, TLB placement, TLB shootdown, etc.) can be efficiently migrated to the new TLB architecture with minimal implementation overhead. To further improve translation performance in PS-TLB, a set of optimized translation operations are introduced, including exemptive flush, shootdown downgrade and PSB pre-fill that leverage the additional information available in PS-TLB.

5.3.3.1 Parallel Translation Lookup The address translation process starts when the MMU (memory management unit) issues a virtual address in request of a physical address for an instruction or data access. The local L1 and L2 TLBs are first checked in sequence for the requested translation. Upon a hit in either L1 or L2 TLB, the PPN from the target translation is retrieved and returned to the MMU. Upon a L2 TLB miss if the page/translation is shared, then it could be either at the PSB home tile on chip, or in the off-chip page table, requiring longer access time. Since the MMU would not know *a priori* whether the requested translation is private or shared, both the PSB home tile and the page table are searched in parallel, as illustrated in Figure 32. The MMU either receives the requested translation from the PSB home tile, or waits for the information from the page table access. Upon receiving the requested translation entry, the TLB, and as appropriate, the PSB home tile are filled with appropriate contents, as explained in the following section.

5.3.3.2 Translation Classification Aware Fill/Placement

From the perspective of the PS-TLB, the parallel lookup into both the PSB and the page table triggered by a L2 TLB miss could result in three different scenarios. The PS-TLB performs a *translation-classification-aware fill/-placement* depending on the scenarios. These scenarios are illustrated in Figure 34.

In Figure 34 (a), the PSB home determines that it has a valid translation by indexing its entries using the requested VPN and examining the *PID* field). The PSB replies to the requester with the corresponding PPN and state. The requester, upon receiving the reply, also adds this entry to its local TLB. This local replication will serve fast translation upon subsequent requests on the same entry. In this case, the reply from the page table, which arrives later, is discarded.

In Figure 34 (b), the translation request misses in the PSB home tile and the returned PTE from the page table is private (e.g. $S = 0$). The MMU is informed that the request is satisfied and the entry is added to the local private TLB.

In Figure 34 (c), the translation request misses in the PSB home tile and the fetched PTE is shared ($S = 1$). In the case of runtime data classification a similar action occurs if the fetched PTE is private with the requester being different from the *FAC* core. In this case, when $S = 0$ and $Req \neq FAC$, the PTE is changed by the OS from private to shared in the page table. The MMU is informed that the request is satisfied and the entry is added to the private TLB and the PSB home tile.

5.3.4 Optimized TLB Shootdown

TLB Shootdown is necessary to keep translation consistent. Normally triggered by OS changes to PTEs, TLB shootdown is traditionally handled using IPIs, which requires all processor cores to be halted to handle the IPI. OS changes to PTEs can be classified as unsafe changes and safe changes.

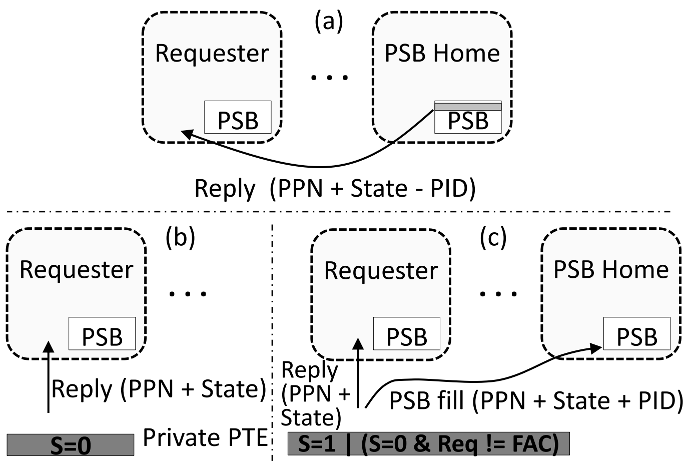


Figure 34: Translation reply and fill on PS-TLB ((a):Hit a PSB entry(b):PSB miss of a private PTE (c):PSB miss of a shared PTE

Considering private versus shared PTEs, four scenarios (i.e., unsafe changes to private PTEs, safe changes to private PTEs, unsafe changes to shared PTEs and safe changes to shared PTEs) are distinguished to efficiently integrate the TLB shootdown process into PS-TLB.

TLB shootdown can be processed using the same mechanism for the first two scenarios (unsafe/safe changes to private PTEs), as illustrated in Figure 35(a). In these two scenarios, the OS is aware that the modified PTE is private. Therefore, the shootdown process can be downgraded to a simple invalidation. This *shootdown downgrade* process reduces overhead by invoking the TLB invalidation instruction (e.g., INVLPG in Intel x86) to invalidate the entry in the owner's TLB without interrupting any of the other processing cores.

When an unsafe change occurs on a shared PTE, as shown in Figure 35(b), all the corresponding TLB entries, as well as the PSB entry (if any), must be invalidated. This is largely the same as the traditional shootdown process.

The fourth scenario (Figure 35(c)) involves safe changes to shared PTEs. In this scenario, the corresponding TLB entries in all sharers of the PTE are invalidated, as normally occurs in a shootdown process. Meanwhile, the OS *pre-fills* the PSB home with the updated entry. This avoids future off-chip translation misses when at least one of the sharers re-accesses the page after the shootdown process. Such a pre-filled entry in the PSB is likely to be used in the near future by its sharers (i.e., prior to eviction from the PSB) as TLB entries invalidated due to safe changes typically contain active translations. Pre-filling the translation entries into PSB can avoid unnecessary stalls due to off-chip misses caused by shooting down entries upon safe changes.

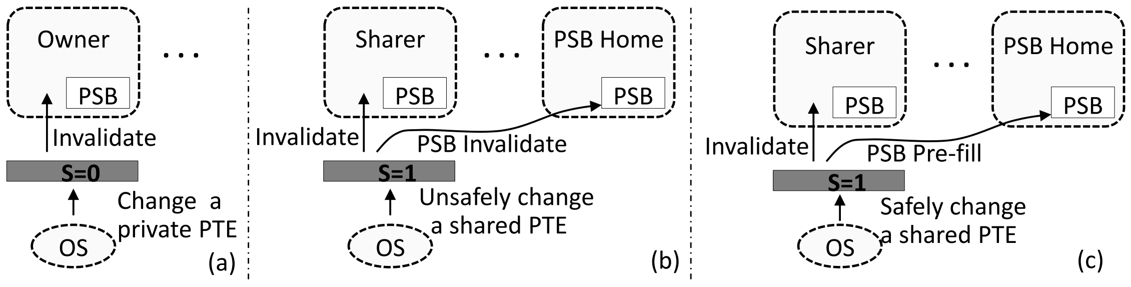


Figure 35: TLB shootdown process on PS-TLB

5.3.5 Optimized TLB Flush

For a typical private TLB organization, a TLB flush is performed upon a context switch to avoid translation conflicts. In PS-TLB, an *exemptive flush* is conducted to flush only the local TLBs. For the PSB, the *PID* uniquely identifies a process and similar to the tagged L2 TLBs [104], a PSB flush is not required during a context switch. When a previously switched out process is reactivated, its surviving shared translations residing in the PSB can still be utilized, reducing the context switch overhead. Since shared translation entries are used by multiple threads, entries left over in the PSB by the switched out thread may be still needed by other threads. Thus, retaining PSB entries can significantly reduce translation misses and avoid translation stalls during context switches.

5.3.6 Atomicity and Race Conditions

As shared resources, PTEs should be updated atomically, which is typically guaranteed by read-modify-write operations and a page table locking mechanism. The updated information will be observed by the cores upon TLB fills or PSB pre-fills after shootdowns. In PS-TLB, however, a core may receive a stale translation entry from the PSB after the PTE is locked in the page table and the corresponding entries are updated/invalidated in either the PSB or TLB. This results in a race condition in which the core utilizes a non-updated translation entry for its local translation. To avoid this effect, PS-TLB enforces that the PSB entry must be invalidated prior to the TLB shootdown. Alternatively, the TLB can discard the reply message from the PSB for an entry that has been invalidated in a shootdown process within a certain amount of time. This time could be an experimentally determined threshold. In the PS-TLB design PSB entries are invalidated prior to a TLB shootdown to avoid the complexity and uncertainty of selecting a threshold.

5.3.7 Discussion

This section presents several design considerations for PS-TLB in different contexts.

5.3.7.1 Scalability The PS-TLB is efficient for future CMPs with increasing numbers of cores (i.e., many-core CMPs). First, each tile retains a local copy of the translation, enabling low and constant translation latency upon an L2 TLB hit independent of the number of cores and NoC topologies. The PSB is designed in a distributed manner that naturally scales with the number of cores/tiles in the CMP. Additionally, only a very small PSB is required, compared to the size of a private L2 TLB. Further, the PSB is also considerably smaller than the space required to add tags to the TLB, making the overhead much less than a tagged shared solution.

5.3.7.2 Multi-program Workloads The PS-TLB can be applied to multi-program workloads directly, without any significant performance reduction compared to a private approach. In the context of multi-program workloads, most pages are classified as private, resulting in less usage of PSBs. However, the PS-TLB will outperform a physically shared solution that increases the access latency of all L2 TLB accesses, which in this scenario are predominantly private. More, the PSB can still serve any global pages that are shared among all workloads, which could also provide some benefit.

5.3.7.3 Thread Migration Thread migration can be handled in PS-TLB in a similar way as a context switch. Typically, migration results in a TLB flush to the cores involved in the migration.

This avoids any potential conflicts with the translations from different processes. In the PS-TLB, an exemptive flush is performed. Thus, translations in the PSB remain intact during the thread migration. The *PID* will resolve any conflicts between processes that use the same virtual address. This allows the migrated thread to continue using its shared translations in the PSB after the migration. Consequently, off-chip translations penalty caused by thread migration is mitigated.

5.4 EVALUATION

This section presents the evaluation of the PS-TLB. First, PS-TLB is evaluated using the compiler and OS runtime page classification support. Based on the impact of the two classification schemes (compiler and OS), an appropriate one is selected to provide support for PS-TLB, which is then compared with several prior TLB mechanisms that leverage sharing [14, 12]. The latency, miss rate and performance of PS-TLB are evaluated and compared with the related work. Additionally, a sensitivity study is presented to show the effects of using different TLB/PSB sizes. Finally, additional benefits due to optimized flush and shootdown are evaluated. In all the evaluation results, the PS-TLB with a certain configuration is represented as PSTLB**m** + PSB*n*, where *m* and *n* are the numbers of the private TLB and shared PSB entries, respectively, in the configuration. For example, PSTLB**256** + PSB16 represents a PS-TLB configuration with 256 TLB entries and 16 PSB entries.

The experimental system uses the Wind River Simics [71] environment to simulate a 16-core CMP interconnected by a 4×4 mesh network. Table 4 summarizes the architectural parameters selected for the experiments. The simulated system uses 4K page size and has a 150-cycle fixed last level TLB miss penalty similar to prior work. The replacement policy for all the caches, TLBs and PSBs in the system uses LRU (least-recently used).

For input workloads, the experiments use emerging parallel C/C++ and Java workloads from the DACAPO [17], SPLASH-2 [5] and PARSEC-2 [15] benchmark suites, which are representative of diverse application domains including scientific computing, text processing, database transaction handling, web hosting, financial modeling, etc. Table 5 shows the description and input working set sizes used for these benchmarks.

Table 4: Architecture configurations

System Parameters:	
Processor	16 SPARC cores, 2G Hz, 2 issue width
Operating System	64-bit Solaris 10
Network	4×4 Mesh, packet switching, X-Y routing, 3 cycles per hop
Main Memory	4GB, 150-cycle latency
Cache Parameters:	
L1 Cache Size	16KB/core private (MESI protocol)
L1 Cache Associativity	4-way associative
L1 Cache Latency	1-cycle hit latency
L2 Cache Size	512KB/core distributed shared
L2 Cache Associativity	32-way associative
L2 Cache Latency	3-cycle hit latency
Cache Block Size	64 Bytes
TLB Parameters:	
L1 TLB Size	32 entries/core
L1 TLB Associativity	4-way associative
L2 TLB Size	256 entries/core
L2 TLB Associativity	16-way associative
L2 TLB Latency	1-cycle hit latency
Off-chip Translation Latency	150 cycles
PSB Parameters:	
PSB Size	16 entries/core
PSB Associativity	4-way associative

Table 5: Benchmarks

Benchmark Suits	Benchmarks	Input Workloads
DACAPO	avrora	default input size
	batic	3 SVG files (default)
	eclipse	small input size
	fop	default XSL-FO file
	h2	400 SQL database transactions
	jython	small input size
	luindex	default set of documents
	lusearch	64 keyword query batches
	pmd	default input size
	sunflow	range of 1-256 (default)
	tomcat	range of 1-4 (small)
	xalan	range of 1-100 (default)
SPLASH-2	ocean	1026x1026 matrix
	lu	2048x2048 matrix
	cholesky	*tk*23.*O* input
	raytrace	*teapot* input
	water	3375 molecules
PARSEC-2	blackscholes	200000 options
	swaptions	256 swaptions
	fluidanimate	in-35K.fluid input
	canneal	100000.nets
	x264	640 × 360 4 frames

5.4.1 Impact of Classification Mechanisms

The PS-TLB architecture does not rely on a specific page classification mechanism and the required classification information can be derived from different mechanisms. To demonstrate that PS-TLB can effectively use different data classification schemes, this section compares PS-TLB's translation miss rate and performance achieved by two different data classification schemes, namely the PSP classification presented in Chapter 3 and the OS classification used in R-NUCA. The tested benchmarks are limited to C/C++ programs that can be compiled by the SUIF infrastructure, in which the PSP analyses are conducted.

Figure 36 compares the miss rate elimination for PS-TLB with PSP and OS based classification. For some benchmarks such as LU, SWAPTIONS and RAYTRACE, the OS driven PS-TLB achieves a slightly better miss rate reduction. For other benchmarks the compiler-driven classification performs better. With the page granularity in TLB, the effectiveness of each data classification mechanism largely depends on certain program data access behaviors or TLB/PSB configurations. For example, if the program's practically private page actually has few sharers and the PSB suffers from high pressure, the compiler-assisted classification PS-TLB works better as it absorbs practically private page entries to local TLBs without caching them in the PSB. On the other hand, if the OS-based classification can place an appropriate amount of shared pages into PSB and those shared pages are actually heavily shared, then OS-based PS-TLB will have an advantage. On average, the difference in miss rate reduction between the compiler-driven and the OS-driven PS-TLB is less than 4%. Performance wise, the two classification schemes do not exhibit a remarkable differentiation, as shown in Figure 37.

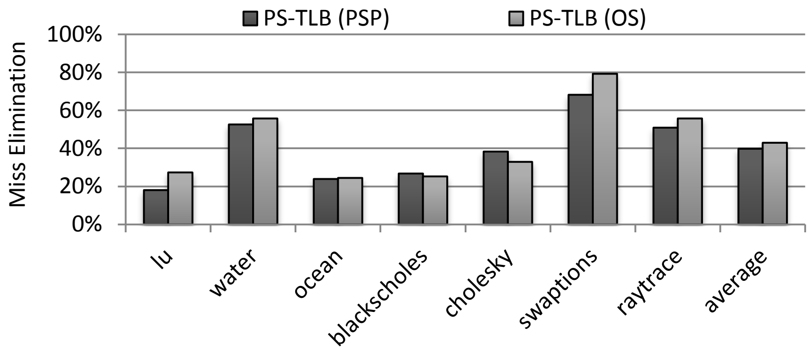


Figure 36: TLB miss elimination on PS-TLB with different page classification schemes

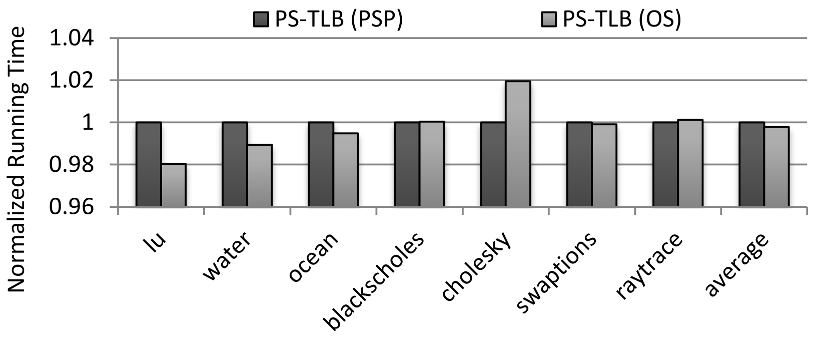


Figure 37: PS-TLB performance comparison with different page classification schemes

The effectiveness of the OS page classification in PS-TLB architecture does not conflict with the conclusions in Chapter 3, in which several problems (i.e., false sharing, data pollution, initialization) of the OS page-level classification are discussed. In the PSP classification based coherent caching scheme the classification information is required at cache block granularity. With respect to TLBs, the data unit is explicitly at the operating system page level and thus, the cache block level information is not needed. For example, classifying a page that contains individual blocks private to different cores as shared results in false sharing from the viewpoint of a cache. However, this false sharing becomes actual data sharing from the perspective of a page transaction and the TLB entry of that page.

Thus, due to the small quantitative advantage of the OS versus compiler classification shown in Figure 37 and the reduced assistance the compiler can provide due to the granularity of access, OS-based classification is employed in the remaining evaluation of the PS-TLB.

5.4.2 Comparison with Shared TLB

Centralized shared TLB has been recently studied [12] as an alternative to industrial standard private TLB to provide larger translation buffer at the expense of scalability and single translation latency. This section compares PS-TLB with the centralized shared TLB in terms of translation miss rate, latency and performance.

5.4.2.1 Translation Miss Rate The miss rate of the centralized physically shared L2 TLB and PS-TLB is presented in Figure 38. Recall that in the PS-TLB, last-level TLB misses may be served by the PSB if the requested entries are shared and cached in the PSB. Thus, the PSB hit rate indicates the percentage of page-table translations (i.e., misses) that can be eliminated. As shown in Figure 38, for most applications PS-TLB is competitive with physically shared, but does increase the miss rate, in some cases by a significant margin. These benchmarks that perform poorly often have a high number of shared pages (e.g., XALAN, LU, WATER, CANNEAL) that put more pressure on the smaller PSB capacity for shared pages than an entirely shared L2 TLB (see Figure 26). The significant advantage of PS-TLB in miss reduction can improve translation access latency, which is evaluated in the following section.

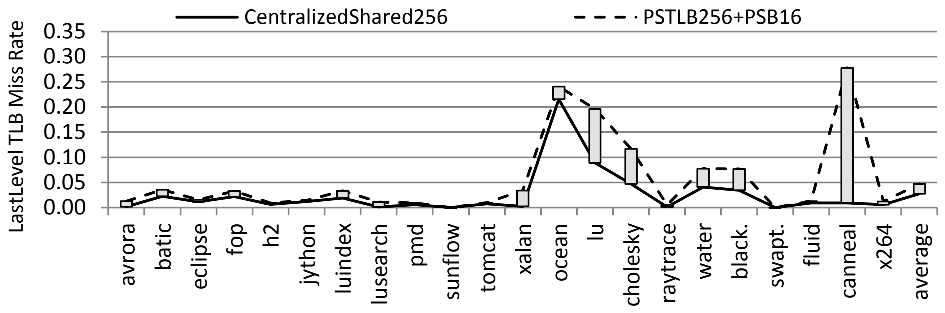


Figure 38: Miss rate comparison of the PS-TLB with a PSB size of 16 entries compared with a centralized shared TLB

5.4.2.2 Translation Latency The key component of performance impact of the TLB design is the reduction of stall cycles per access. Figure 39 shows the translation latency reduction of the PS-TLB normalized to the baseline of a physically shared L2 TLB. Due to the reduced latency of primarily local hits, nearly all of the tested benchmarks exhibit considerable latency reduction, reaching a reduction of more than 80% in some cases. The three benchmarks with degradations are LU, CHOLESKY and CANNEAL, for which the centralized shared L2 TLB has a significant miss rate advantage. On average, PS-TLB reduces translation latency by 45%.

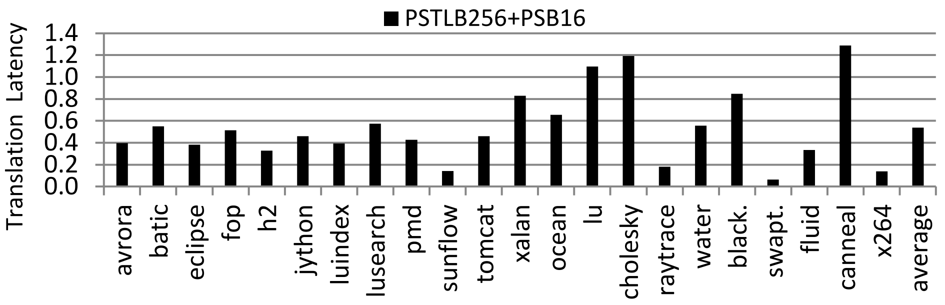


Figure 39: Translation latency (normalized to centralized shared)

5.4.2.3 Overall Performance Impact The impact of translation on overall system performance varies significantly for different applications and is dependent on a variety of factors including memory access intensity, workload/translation set sizes, data access patterns, instruction execution time, etc. It can be seen from Figure 40 that some benchmarks, such as FLUID, WATER and SWAPTIONS exhibit negligible performance gain, although they have achieved significant translation latency reduction in Figure 39. This is due to the translation latencies not contributing significant fractions to the entire execution time in these benchmarks. In a similar fashion, benchmarks that saw latency degradations such as LU and CHOLESKY did not introduce a performance degradation. In many of these cases, the TLB miss latency was often overlapped with other delays from memory system misses and/or poor load balancing in the application design. BLACKSCHOLES achieves little speedup because its private-dominant data/translation does not benefit much from any translation sharing mechanism.

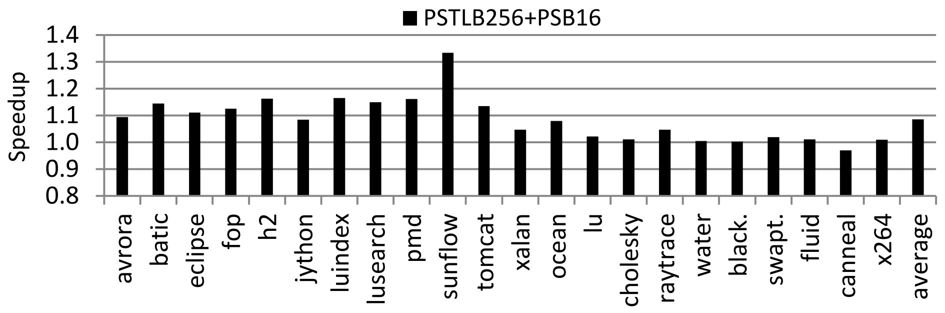


Figure 40: Speedup over a centralized shared TLB

In contrast, the latency improvement is critically important in some cases. For example the SUNFLOW benchmark has a 33% performance improvement due to a more than 80% TLB latency reduction. The average performance improvement over all benchmarks is 9%.

5.4.3 Comparing with Prefetching Mechanism

In addition to comparing with shared L2 TLBs, the PS-TLB is compared with the state-of-the-art scalable scheme that leverages prefetching, TLBPrefetch [14].

The TLBPrefetch technique was simulated for comparison using a 16-entry per-core prefetch buffer with a centralized distance table that has four entries per-tile as well as distance buffers, which are local distance table caches consistent with [14]. In comparison, PS-TLB uses a 20-entry PSB, which is comparable to the combined resources of the prefetch and distance buffers.

The PS-TLB scheme is actually considerably simpler than TLBPrefetch as it does not require the distance table resources, which are as large as adding an additional TLB. PS-TLB also saves the $O(n^2)$ confidence counters required in TLBPrefetch to determine the tiles from which translation data can be prefetched.

Figure 41 compares the PS-TLB's capability of reducing last-level TLB misses over TLBPrefetch. PS-TLB reduces miss rate for all benchmarks between 10% and 75% with an average of a 32% reduction. The reduced off-chip translations result in a maximum of 48% and an average of 16% latency reduction over TLBPrefetch, as illustrated in Figure 42. All benchmarks were either improved or achieved the same latency with PSTLB**256** + PSB20.

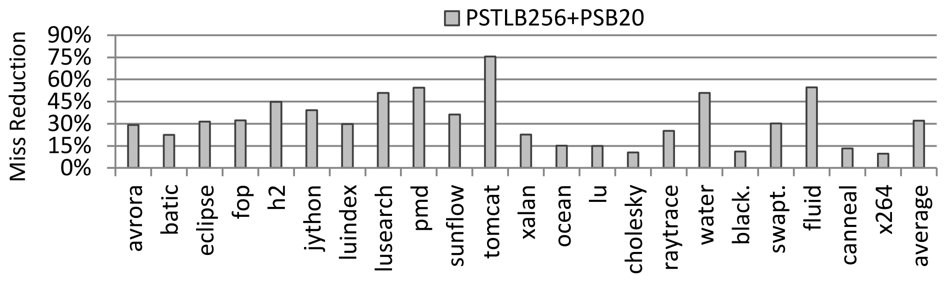


Figure 41: Percentage of last-level TLB miss reduction compared to prefetching scheme

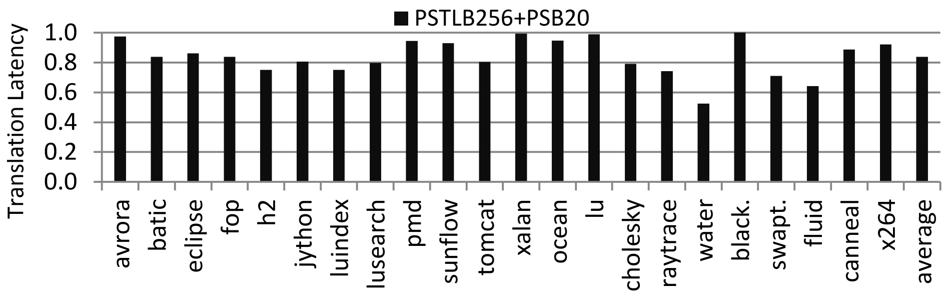


Figure 42: Translation latency compared to prefetching scheme

Figure 43 reports the speedup achieved by PS-TLB over TLBPrefetch. For some benchmarks such as H2, JYTHON and LUSEARCH, the PS-TLB outperforms the prefetching scheme by over 5%. These benchmarks also achieve remarkable latency reduction (20% reported in Figure 42).

For many other benchmarks (e.g., PMD, XALAN, WATER, etc.) PS-TLB only achieves modest or negligible improvement. On average, PS-TLB performs 2.5% better than the prefetching scheme.

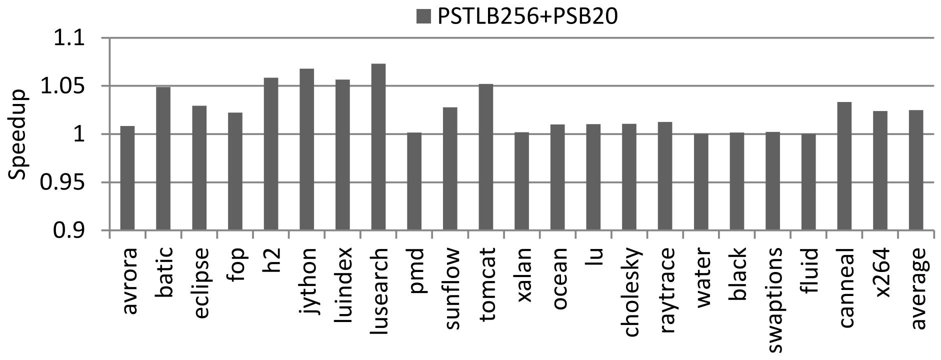


Figure 43: Speedup over prefetching scheme

5.4.4 Sensitivity Analyses

To demonstrate the benefit of the PS-TLB in different configurations, this section presents a study of a smaller TLBs with 64 entries/core and the effect of different PSB sizes. Figure 44 and Figure 45 show the translation miss elimination and latency reduction on configurations with 256 and 64 TLB entries per core. In each case the same number of L2 TLB entries per core is compared with a zero sized PSB (A PS-TLB with a zero sized PSB is essentially equivalent to a private L2 TLB).

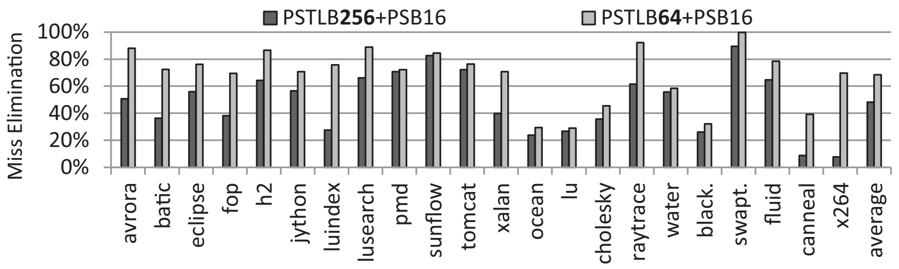


Figure 44: Percentage of last-level TLB miss elimination for a PS-TLB with 64 TLB entries/core

Clearly, smaller TLB capacity (or equivalently larger working set size) increases off-chip translation rate and prolongs the translation latency, enlarging the optimization opportunity for PS-

TLB. As can be observed from Figure 44, last-level TLB miss elimination increases from 48% on PSTLB**256** + PSB16 to nearly 70% on PSTLB**64** + PSB16. On some benchmarks (e.g., LU and WATER) the improvement is not remarkable since the working set sizes are relatively small and fit into the 64 entry/core configuration.

Figure 45 compares the latency improvement over a zero sized PSB. In PSTLB**64** + PSB16 configuration, the latency reduction from the PSB is amplified due to more frequent TLB misses than the 256 entry per core configuration. On average, the translation latency reduction is 55% on a 64 entry per core configuration, as compared to 32% in a 256 entry per core system.

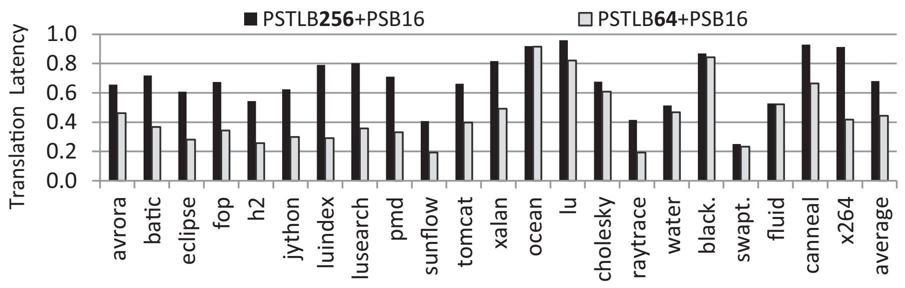


Figure 45: Translation latency of a PS-TLB with 64 TLB entries/core (normalized to zero PSB)

Figure 46 reports the percentage of last-level TLB misses eliminated by a PS-TLB with variable sized PSBs normalized to a zero sized PSB. With just four entries in the PSB the miss rate is reduced by 33% on average and this reduction rises to 55% by a 20-entry PSB. This leads to a corresponding scaling in the latency reduction, as shown in Figure 47. The average latency reductions brought by 4, 8, 16 and 20-entry PSBs are 23%, 27%, 32% and 36%, respectively.

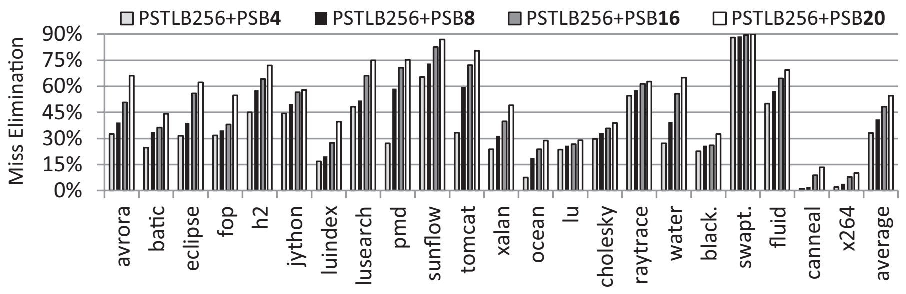


Figure 46: Percentage of last-level TLB miss elimination for a PS-TLB with different PSB sizes

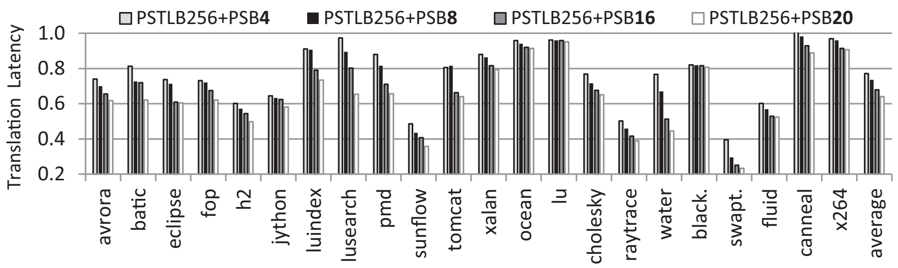


Figure 47: Translation latency of a PS-TLB with different PSB sizes (normalized to a zero PSB)

One inefficiency incurred by PS-TLB is the parallel lookup upon a last-level TLB miss. A lookup into the PSB would be unnecessary if the requested entry turns out to be private, since that entry is never placed in a PSB. Unfortunately, the requester cannot know this until it receives the translation from a TLB or the page table. Such unnecessary lookups are *wasted PSB lookups* that could incur a power inefficiency. As revealed from Figure 48, most benchmarks have less than 20% wasted PSB lookups. Only a few benchmarks (BLACKSCHOLES and SWAPTIONS) have high percentages of wasted PSB lookups. This is likely due to the extremely low frequency of L2 TLB misses compared to hits (please refer to the TLB miss rate in Figure 29). It is expected that the benefit from eliminating page table lookups significantly outweighs the overhead incurred by wasted lookups and does not bring a significant negative impact, in terms of energy consumption, on the overall TLB operations.

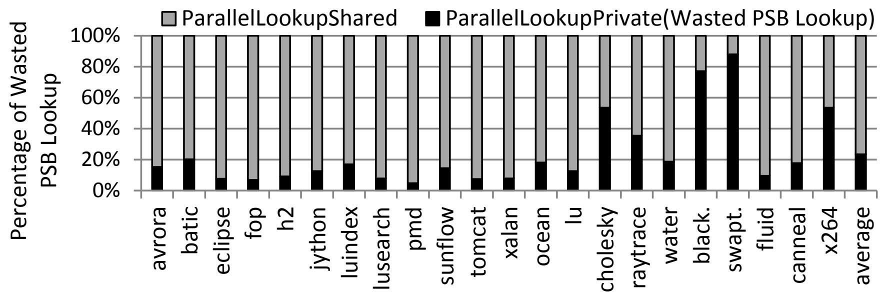


Figure 48: Unnecessary (wasted) PSB lookups after a L2 TLB miss

5.4.5 Additional Benefits from PS-TLB

In addition to the translation miss and latency improvement, the PS-TLB design enables optimized shootdown and context switch operations. This section quantify these additional benefits.

5.4.5.1 Shootdown The private and shared classification of translations used by the PS-TLB provides an opportunity to improve the TLB shootdown process. Recall that if a private entry is identified for a shootdown by the OS, the shootdown can be downgraded to a simple invalidation only stalling a single core. Shared entries with a common address that are de-mapped in multiple cores correspond to a shootdown of a shared PTE, which reverts to the standard shootdown process. In multi-core systems, TLB shootdown is supported by low-level hardware primitives (e.g., INVLPG instruction in Intel x86 and TLB demap in SPARC/PowerPC) to invalidate TLB entries. This makes it easy to implement an invalidation instead of shootdown for private PTEs. In the experimental machine, the TLB demap operations are tracked to estimate the impact of shootdown downgrades. Figure 49 estimates the percentage of shootdowns that can be downgraded based on the amount and classification of the de-mapped translations. On average, more than 30% of the shootdowns can be downgraded to invalidations, which avoids considerable stalls.

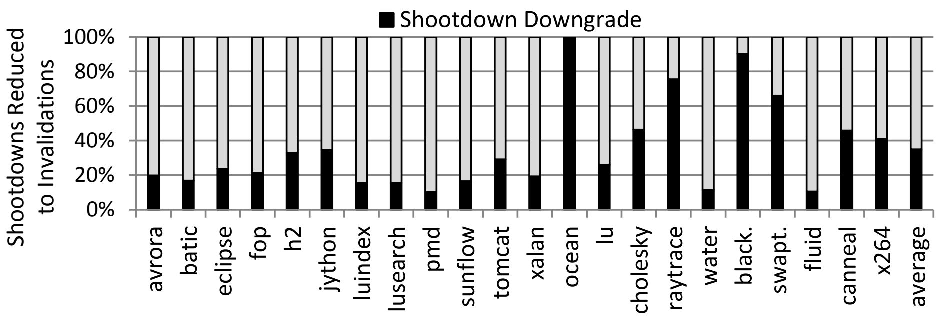


Figure 49: Estimation of TLB shootdown downgrade savings

5.4.5.2 Context Switching The principal reason to introduce tags into the TLB was to avoid flushing due to context switching. PS-TLB provides a similar benefit with a much smaller tagged resource, the PSB. To study the impact of context switches, the context switch experiments group benchmarks into pairs and run each pair with 32 threads (16 for each benchmark) on the simulated 16-core machine. Thread binding is used to ensure that two threads, one from each benchmark in a pair, multiplex the same processing core. Context switches are identified by detecting changes of core mode and values in the context register. The experiments measure the last-level TLB miss rate and latency of PS-TLB with different PSB sizes normalized to a fully tagged L2 TLB.

From Figure 50 it can be seen that the PSB size has a significant impact on TLB misses due to context switching. For FLUID and SWAPTIONS, the last-level TLB miss rate drops from 7.6% to 1.4% by introducing a 16-entry PSB, an 81.6% reduction. For other testing pairs, a 16-entry PSB improves miss rate reduction from 12% to 68% across all benchmarks, which are directly reflected in translation latency savings as shown on the left hand side of Figure 50. Comparing with a tagged baseline, which uses 16 times as many tags, PSTLB256+PSB**16** achieves close or better miss rate for all pairs except WATER and BLACKSCHOLES due to the high percentage of private pages in BLACKSCHOLES, which are not stored in the PSB. When the PSB is increased to 32 entries, PSTLB256+PSB**32** outperforms the tagged TLB on both translation miss rate and latency on average. In terms of total additional resources, the tagged TLB requires $256 \times 16 \times 13 = 53248$ bits, assuming each tag has a size of 13 bits (e.g., ASID on UltraSparc) compared to only $32 \times 16 \times (64 + 13) = 39424$ bits for PSTLB256+PSB**32**.

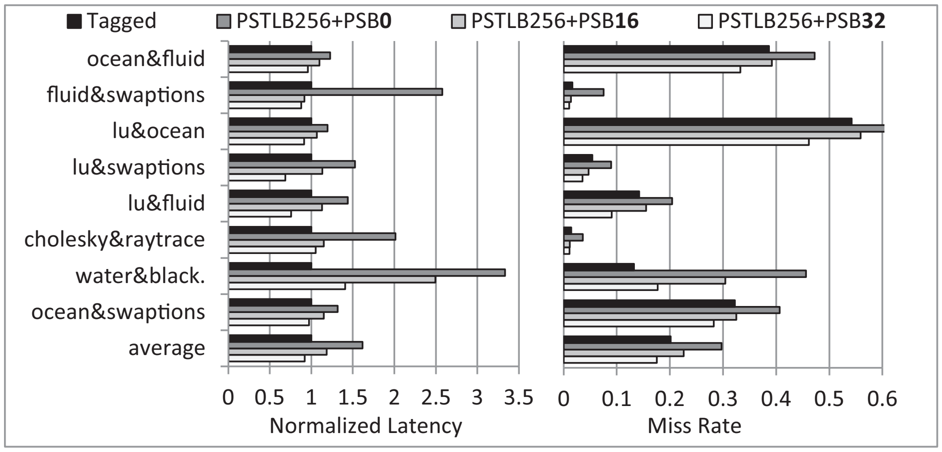


Figure 50: Latency and miss rate saving during context switches

6.0 COMPILER-BASED DATA PARTITIONING AND COMMUNICATION PATTERN ANALYSES OF PRACTICALLY PRIVATE DATA

Earlier, a lightweight compiler analysis was conducted to determine whether data accessed in the system was private or shared (see Chapter 3). A large portion of the data was classified into a new category of practically private, either due to limitations of the compiler analysis or because the private access dominated a small amount of sharing. However, it is possible to use the compiler to further examine this practically private class of data and determine more details about its nature and sharing properties.

As such, this chapter presents compiler analyses to detect compile-time deterministic data access patterns, data partitions and communication patterns of the compiler classified practically private data, found in multi-threaded applications. These analyses can be used to classify potentially large portions of practically private data as purely private. Further, the proposed techniques reveal more detail about the available locality and data access behavior of different applications. Thus, the analyses can also be used to expose the synergy of accessed memory locations and determine the data partitioning implied by multi-threaded applications. This information can then be used to assign ownership to data blocks to avoid using schemes such as first touch to determine ownership. Additionally, determining the application ownership *a priori* makes it possible to enforce a partitioning on a system employing a multi-banked logically shared cache organization. Further, this partitioning lays a foundation for a communication pattern detection routine that can be used to guide a configurable NoC [1].

6.1 OVERVIEW

Figure 51 shows a compilation flow for the experimental compiler framework. Multi-threaded applications with optional user directives are fed into the front end and converted to the compiler's intermediate representation. Inter-procedural analysis (IPA) is then applied to handle global data and propagate the information from one procedure to another. The call graph generated in this pass can be used to keep track of the parameter passing when necessary. For each procedure, control flow graphs (CFGs) are generated to represent the control paths in the program. Each node in the graph is a *basic block* in which there is no branch instructions. The sequence of instructions in a basic block represents the data flow in that basic block. Conventional analyses such as IPA and control data flow analyses form the basis of the compiler pass introduced in this chapter. Other relevant analyzing approaches include the reference list based pointer analysis and TI variable detection technique described in Section 3.1 and Section 3.3.1. The reminder of this chapter introduces compiler techniques to extract data partitioning and communication pattern information from practically private data in parallel applications running on shared memory CMPs.

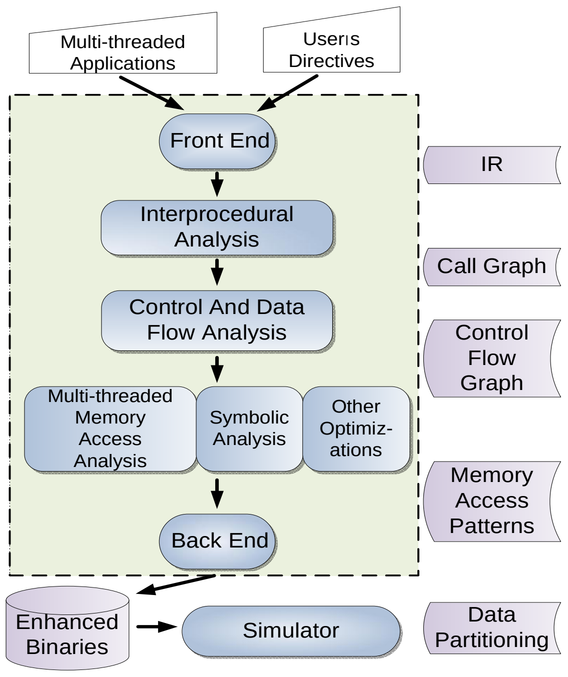


Figure 51: Experimental compiler framework

6.2 MULTI-THREADED MEMORY ACCESS PATTERN ANALYSIS

Traditional data access analyses such as reuse, dependence and locality analyses [3, 73] focus on affine array subscript patterns in loop nests of a single-threaded application. The primary goal of these techniques is to find the relationships of memory locations accessed by different loop iterations. In contrast, the multi-threaded analysis approach discussed in this chapter detects the memory access patterns in a parallel programming context. The pattern information is then used to determine the detailed privacy and sharing information of the data. This information is represented as data partitioning, which implies a thread-data affinity relationship called *ownership*. Based on the data partitioning and ownership concept, communication pattern can also be derived to reflect the inter-thread communication behavior.

This section begins by first introducing traditional memory access regions for dealing with affine array subscript functions in single threaded programs [73, 74]. The region theory forms the basis of the analysis to describe affine array access patterns in an execution phase of the application[1]. The region concept is then extended to MMAP (Multi-threaded Memory Access Pattern) for multi-threaded applications. Finally, multiple phases are considered to create the data partition.

6.2.1 Array Access Regions

Within a loop nest for a single threaded program, a reference to an array A can be generally represented as $A[f(L)]$, where $f(L)$ is the subscript function defined on a set of loop indices $L = i_1, ..., i_m$. The **span** of $f(L)$ resulting from i_k is the maximum distance traversed by varying only i_k from its lower bound l_k to its upper bound u_k [73, 74]:

$$\begin{aligned} span_{i_k} = |&f(i_1, ...i_{k-1}, u_k, i_{k+1}, ..., i_m)- \\ &f(i_1, ...i_{k-1}, l_k, i_{k+1}, ..., i_m)| \end{aligned} \tag{6.1}$$

Similarly, the **stride** is defined as the minimum distance across memory by changing only i_k by its step s_k :

$$\begin{aligned} stride_{i_k} = |&f(i_1, ...i_{k-1}, i_k + s_k, i_{k+1}, ..., i_m)- \\ &f(i_1, ...i_{k-1}, i_k, i_{k+1}, ..., i_m)| \end{aligned} \tag{6.2}$$

[1] An execution phase is defined as a loop nest in the code.

Thus, an array access region can be described by the following form, where O denotes the starting offset:

$$\mathbf{R} = (A^{stride_{i_1},\ldots,stride_{i_m}}_{span_{i_1},\ldots,span_{i_m}} + O) \tag{6.3}$$

The original access region theory has been developed to simplify data access analyses such as array dependence and privatization analysis. It also offers a set of manipulations on regions such as region coalescing and intersection to allow corresponding code transformations [73]. Regions with their corresponding manipulations lay a foundation for the multi-threaded memory access analysis. To simplify explanation, all analyses and techniques introduced in later sections are assumed to be performed on a single array A. Thus, a whole program can be analyzed by applying the following concepts on each array in the code.

6.2.2 Multi-threaded Array Analysis

The original array access region cannot represent data accesses in multi-threaded applications because the same array access in a loop nest results in multiple instances of accesses by different threads. Additionally, the bounds of the loop nest and array subscripts may be different for each thread. Thus, each thread requires a number of unique regions to represent its data accesses. To address these issues, the following techniques are developed to extend the region concept.

6.2.2.1 Thread-Identifying Structures For the compiler to correctly discover the memory accesses of each thread it is necessary to interpret how the code structures use the TI variables (defined in Section 3.3.1). The study of multi-threaded code from a variety of program domains including scientific computing, multimedia, image processing and financial processing reveals that there are particular programming structures, such as loops and conditionals, that determine the memory access pattern of the threads. These code structures are named *Thread Identifying (TI) Structures*.

TI structures place constraints between allocated blocks of memory and the threads who can access them. Discovering all possible types of TI structures is not possible. However, widely used TI structures can be identified. These TI structures are summarized into three categories as follows:
TI Variables in Loop Bounds: One method to partition data accesses during a particular phase

of the application is to use a function of TI variables as loop bounds within a loop nest. As the different loop iterations typically access different array indices, in this method, TI variables place a constraint on which iterations of the nested loop can be executed by each thread. As a result, different threads access different, and often independent, array indices.

TI variables in loop bounds usually imply a block access pattern or a nested access pattern, as shown in Figures 52 and 53, respectively. When both lower and upper bounds are functions of TI variables this often implies a block pattern. For example in Figure 52, the loop bounds `myfirst` and `mylast` are functions of TI variables and a blocksize. The access pattern is shown for `nprocs=4` and contains four blocks of size `blocksize`. As `mylast` is the only function of a TI variable in Figure 53, the pattern is nested. The thread with `pid=0` accesses a single block of size `blocksize`, which is nested in partition one, which in turn is nested in partition two, etc.

```
bs = datasize/nprocs;
myfirst = bs*pid;
mylast = bs*(pid+1);
for(j=myfirst;j< \
        mylast;j++)
   A[j]......
```

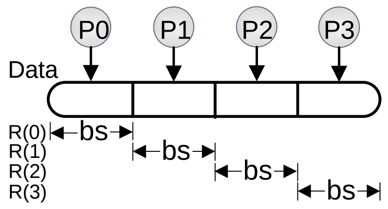


Figure 52: Example TI structure and the corresponding block pattern ($nprocs$ = 4)

```
bs = datasize/nprocs;
mylast = bs*(pid+1);
for(k=0;k<mylast;k++)
    A[k]......
```

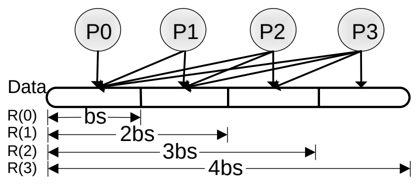


Figure 53: Example TI structure and the corresponding nested pattern ($nprocs$ = 4)

Within loop nests, the TI structures can be more complex. Figure 54 shows a grid-like access pattern and its corresponding TI structure where `bsm` and `bsn` are the block size partitioned along two dimensions.

```
ps = sqrt(nprocs);
bsm = m/ps; bsn = n/ps;
for(i=(pid/ps)*bsm; \
 i<(pid/ps+1)*bsm;i++)
for(j=(pid%ps)*bsn; \
 j<(pid%ps+1)*bsn;j++)
    A[i][j]......
```

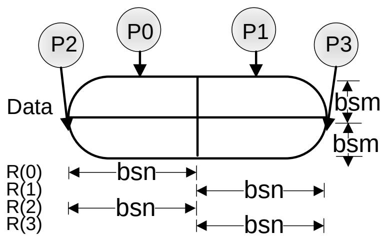


Figure 54: Example TI structure and the corresponding grid pattern (*nprocs* = 4)

TI Variables in Array Subscripts: Another way to distribute array accesses among multiple tiles is to have TI variables serve as array subscripts. The statement `A[pid+j] = x` is an example of a TI structure of this style. For a TI structure of this kind, knowing the value of the function of TI variables used in the array subscripts is essential to understand how the data is accessed. For four threads, the access pattern from this TI structure is similar to the one shown in Figure 52.

TI Variables in If Statements: Another common TI structure includes a conditional statement that impacts the array access. In this case the TI structures are difficult to analyze because the flexibility in the forms that can be employed in this code structure is high. For example, it is difficult to determine a pattern from the TI structure: `if (foo(i) == pid)` unless the structure of the function *foo()* is known. A method for analyzing TI structures from this category is to select regular structures that use conditionals that are extensively used in many programs. For example, the TI structure `if((i%4) == pid)` in Figure 55 implies an interleaved pattern as shown in that figure.

```
   for(i=0;i<16;i++){
     if(i%4 == pid){
       A[i]......
```

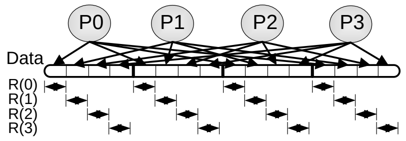


Figure 55: Example TI structure and corresponding interleaved pattern (0 ≤ *pid* ≤ 3)

6.2.2.2 Multi-threaded Memory Access Patterns The purpose of TI structure analysis is to discover patterns of memory access for each thread in the system. Thus, the MMAP is defined as a formal representation to describe the way multiple threads access blocks of data. Given a phase of parallel code for n threads, a list of n regions can be created for each array access by replacing TI variables with the actual values for that thread. Then the MMAP for an array access can be defined in a parallel program phase as:

$$\mathbf{M} = \{R(0), ..., R(n-1)\} \tag{6.4}$$

In the above equation, region $R(x)$ follows the notation in Eq. (6.3) and corresponds to the thread with the id x. Access weight is also defined for each region in the MMAP to reflect the number of times the thread accesses its region. It can be calculated based on the information from loops associated with this region. Assume that an array access appears N times in a m-deep nested loop, and the lower bounds, upper bounds and steps of loop indices for thread x are $l_1(x), ..., l_m(x), u_1(x), ..., u_m(x)$ and $s_1(x), ..., s_m(x)$, respectively. Hence, the formula to calculate access weight for $R(x)$ is:

$$\mathbf{W}(\mathbf{R}(\mathbf{x})) = N * \prod_{k=1}^{m} \frac{(u_k(x) - l_k(x))}{s_k(x)} \tag{6.5}$$

For further analysis convenience, some properties for MMAP are defined as follows:

Definition 4. *A MMAP,* $\mathbf{M}$*, is* access uniform *when* $W(R(0)) = W(R(1)) = ... = W(R(n-1))$

Definition 5. *A MMAP,* $\mathbf{M}$*, is* size uniform *when* $\|R(0)\| = ... = \|R(n-1)\|$ *where* $\|R(x)\|$ *is the number of elements in* $R(x)$*.*

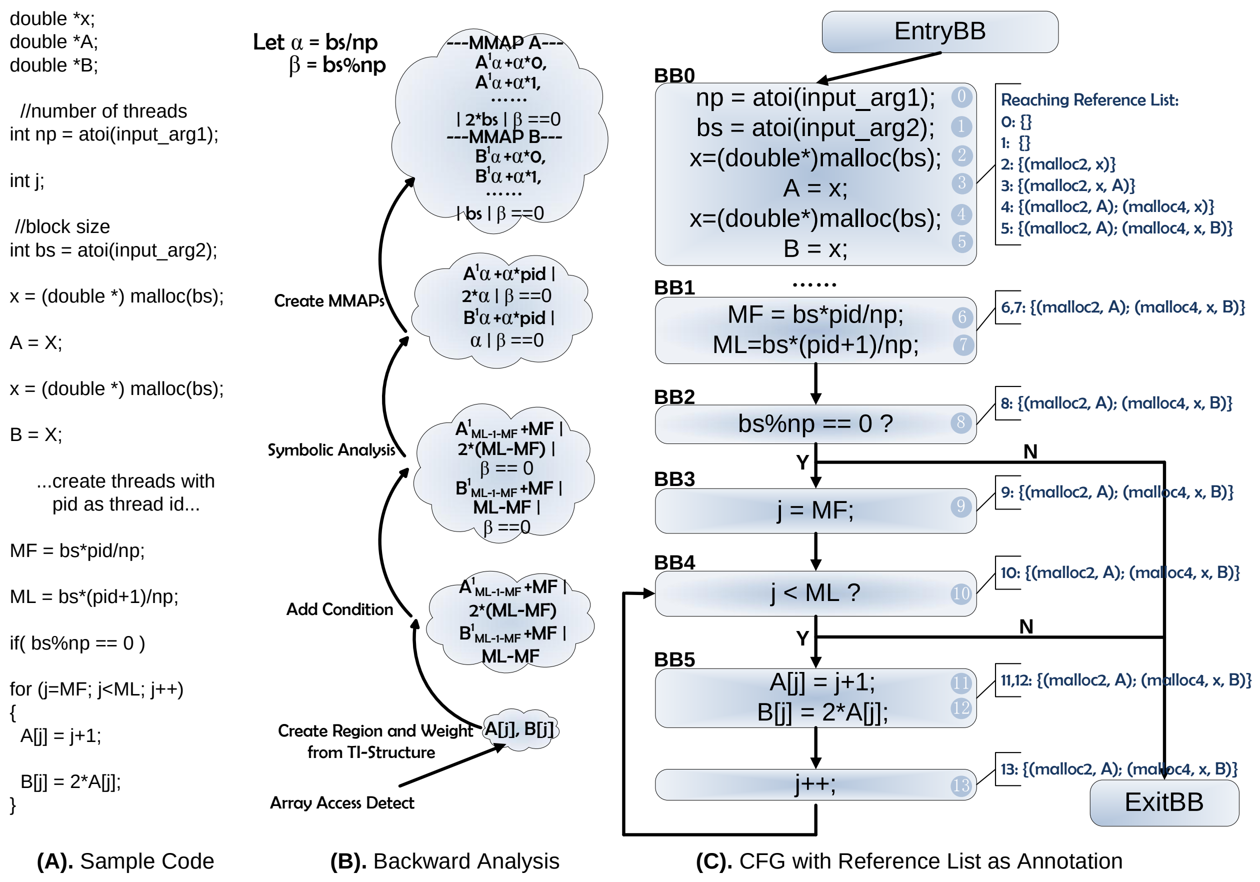
The patterns shown in Figures 52, 54, and 55 are all size and access uniform MMAPs.

Definition 6. *A MMAP,* $\mathbf{M}$*, is* non-overlapping *when*$\forall R(x), R(y)$ *in* $\mathbf{M}$*,* $R(x) \bigcap R(y) = \emptyset$*. Otherwise, the MMAP is* overlapping.

In addition to the properties defined above, some other attributes can be explored such as the *shape* of a MMAP. For example, the MMAP that meets the condition $R(1) \subset R(2) \subset ...R(n)$ has a nested shape, shown in Figure 53. Other shapes could be block (see Figure 52) or interleaved (see Figure 55), etc.

6.2.3 MMAP generation

Although MMAPs are primarily related to array accesses and TI-Structures, there are other programming components affecting the actual access patterns. For example, the accesses might appear in a conditional branch. Also, useful information such as loop bounds and array indices could be variables rather than constants at compile time. In this case, the regions and MMAPs are constructed using symbolic expressions, which are resolved at run time. To handle relatively general situations and perform analyses in a systematic way, control flow and symbolic analysis are adopted, as done in [26] and [49]. The analysis approach is illustrated by an example shown in Figure 56.



```
double *x;
double *A;
double *B;

  //number of threads
int np = atoi(input_arg1);

int j;

 //block size
int bs = atoi(input_arg2);

x = (double *) malloc(bs);

A = X;

x = (double *) malloc(bs);

B = X;

     ...create threads with
       pid as thread id...

MF = bs*pid/np;

ML = bs*(pid+1)/np;

if( bs%np == 0 )

for (j=MF; j<ML; j++)
{
  A[j] = j+1;

  B[j] = 2*A[j];
}
```

Figure 56: MMAP generation flowgraph

The MMAP generation phase consists of forward and backward passes. In the forward pass, the CFG is traversed from top to bottom. As the CFG is being traversed, `malloc()` routines and pointer assignments are detected. Each static `malloc()` call site is given a unique id and a reference list. Initially, there is only a return value of the `malloc()` (*i.e.* the pure pointer) in the reference list. For example, as the analysis reaches statement 2 in $BB0$ (`x=(double*)malloc(bs)`), it creates the reference list $\{(malloc2, x)\}$ where 2 indicates the static malloc id and x is the pure pointer. Subsequent pointer assignments update the reference list as the analysis pass continues forward through the CFG. Thus, the assignment from statement three (`A = x`) adds `A` to the reference list resulting in $\{(malloc2, x, A)\}$. This pointer analysis is used to associate array accesses with the corresponding memory allocations, to which the MMAPs are eventually attached. In addition to pointer analyses, optimizations such as constant propagation and TI-variable recognition are also performed and used to annotate the corresponding nodes in the CFG. However, for clarity only the results of the pointer analysis are shown on the right-hand-side of Figure 56 (C).

Whenever an array access is encountered, the backward pass is initiated at the current node in the CFG and traverses the nodes in reverse to parse the enclosing TI-structures, calculate the weight, and search for the memory allocation conditions for this particular array access. Information found is then used in the construction of regions and MMAPs for the array access. In most cases where input parameters are unknown, regions or MMAPs are calculated symbolically from the expressions available in the analyzing context rather than represented using constants. If there are conditional branches encountered during this process, the eventually generated MMAPs are appended with that condition. The MMAPs, their weights and the condition are thus denoted as a tuple: $\{M|W|C\}$.

Figure 56 (B) shows how the backward analysis is performed to produce the results for each stage of the analysis. For example, as the array access $A[j]$ is detected at $BB5$, the backward analysis generates region and weight (shown in the second cloud from bottom) using the loop bounds information. In the region expression $A^{1}_{ML-1-MF} + MF$, 1 is the stride and $ML - 1 - MF$ is the span. MF is the offset of the region. They are calculated using Eq. (6.1) and Eq. (6.2) based on loop and array access subscript information. $2 * (ML - MF)$ is the weight for the array access. As the backwards traversal continues, the condition ($bs\%bp == 0$) is appended at $BB2$. At $BB1$, symbolic terms MF and ML are updated and represented in terms of bs, pid and np (α and β for short). Finally, the cloud on the top shows the generated MMAPs, each with several regions in it. The number of regions in each MMAP is equivalent to the number of threads np.

Note that if there are different array accesses patterns to the same array in one phase, multiple MMAPs must to be attached to the corresponding `malloc()`. After constructing the MMAP(s) for the particular array access, forward analysis resumes to reach and analyze the next array access. This process continues until the entire CFG is traversed.

6.3 GENERATING DATA PARTITIONING AND COMMUNICATION PATTERN

6.3.1 Data Partitioning

One particular goal of the compiler analysis is to determine the data partition for each array and to use this to enforce data ownership. This has been motivated by the fact that for array accesses in most data-parallel benchmarks, each thread has a set of data on which the thread operates most. These sets of data imply a partition of the accessed array, which in many cases the compiler can discover. Thus, the *data partition* for an array A can be defined based on regions and MMAPs as follows:

Definition 7. *For a program with n threads numbered from 0 to $n-1$, a* partition $\mathbf{P}$ *is a set of non-overlapping regions: $\{R_*(0), ..., R_*(n-1)\}$, as defined in Definition 6. Tile x is said to be the owner for region $R_*(x)$ in the partition.*

The above definition ensures that each element of array A appears in exactly one region, and hence, will be owned by one thread. This condition can be examined by region intersection manipulation [73].

In order to determine a good partitioning for an array, all phases that contain accesses to this array in each program segment should be examined[2]. This can be done by applying the MMAP representation to every phase and comparing these MMAPs to find the partition. To extend the MMAP representation across multiple program phases an additional subscript is added to denote MMAPs in different program phases.

Thus, to determine a data partition for the entire application the *dominating MMAP* is described as follows:

[2]Program segment could consist of a single phase, multiple phases, or the entire application. Methods for subdividing a program into segments are beyond the scope of this work but could use an approach similar to the one proposed by Shao *et al* [87].

Definition 8. *Given a set of MMAPs $M_0, ..., M_{\rho-1}$ corresponding to all the phases in a program with n threads (threads number x ranges from* 0 *to* $n-1$*), a MMAP M_i is said to be the* dominating *MMAP if it has the largest total access weight:*$\sum_{x=0}^{n-1} W(R_i(x))$ *among all ρ MMAPs.*

To determine the dominating MMAP, weights of all MMAPs associated with the same memory block are compared. The one with largest weight is chosen as the dominating MMAP.

From the dominating MMAP, $\mathbf{M_D}$, the partition can be created. First if by Defintion (6) $\mathbf{M_D}$ is non-overlapping, then $\mathbf{M_D}$ is a partition. If $\mathbf{M_D}$ is overlapping, non-overlapping portions of $\mathbf{M_D}$ are constructed by region manipulation such as intersection and subtraction. The weight of the non-overlapping portions is then compared with the dominating non-overlapping MMAP. The one with larger weight becomes the partition. An investigation into the benchmarks show that almost all the studied data-parallel benchmarks exhibit a single non-overlapping dominating MMAPs. This reduces the probability of performing expensive region manipulations.

Partitioning example: Consider the parallel program for four threads shown in Figure 57. The compiler detects that the program has three nested loops or phases. From the analysis of each TI structure, the multi-threaded access patterns for each phase can be determined as follows: The first phase has an interleaved uniform access pattern, the second is block uniform and the third is a nested access pattern. According to relevant definitions, the second program phase produces the dominating MMAP as it has the largest access weight $(4*(pid+1)-4*pid)*4*4*64*2 = 8192$.

```
for(i=0;i<64;i++)
 for(j=0;j<64;j++)
  if(i%4 == pid)
   A[i][j] = ......;

for(i=pid*4;i<(pid+1)*4;i++)
 for(j=0;j<4;j++)
  for(k=0;k<64;k++)
   A[4i+j][k] = ......;
   ...... = A[4i+j][k];

for(i=0;i<(pid+1)*16;i++)
 for(j=0;j<64;j++)
   A[i][j] = ......;
```

Figure 57: Example code

To generate regions in the dominating MMAP M_2, thread-specific values are used to replace the TI variables within the TI structure. In this example, thread ID *pid* ranges from 0 to 3, so the region list for phase 2 would be $\{R_2(0), R_2(1), R_2(2), R_2(3)\}$. First $R_2(0)$ for thread 0 is generated by replacing *pid* with 0. The array subscript $4i + j$ and k can be linearized to $(4i + j) * \sigma + k$ [73] where σ is the size of second dimension of array A. In this example $\sigma = 64$. Then the array subscript becomes $(4i + j) * 64 + k = 256i + 64j + k$. Using Eq. (6.1) and Eq. (6.2), the span of $256i + 64j + k$ by changing i from its upper bound 3 to lower bound 0 is $(256 * 3 + 64j + k) - (256 * 0 + 64j + k) = 256 * (3 - 0) = 768$, and the corresponding stride is $256 * (1 - 0) = 256$. The second span-stride pair, resulting from analyzing variable j, is a span of $64 * (3 - 0) = 192$ and a stride of $64 * (1 - 0) = 64$. The third pair from k is $1 * (63 - 0) = 63$ and 1. The starting offset of the $R_2(0)$ is $256 * 0 + 64 * 0 + 0 = 0$. So the access region descriptor for $R_2(0)$ can be expressed as $A^{256,64,1}_{768,192,63} + 0$.

The access weight can be calculated using Eq. (6.5): $4 * 4 * 64 * 2 = 2048$. Regions for other threads can be built in a similar way: $R_2(1)$: $A^{256,64,1}_{768,192,63} + 1024$; $R_2(2)$: $A^{256,64,1}_{768,192,63} + 2048$; $R_2(3)$: $A^{256,64,1}_{768,192,63} + 3072$.

Since these regions do not intersect, the MMAP they form is non-overlapping. Thus, the data partition of array A is private and can be represented using this MMAP: $P = \{R_2(0), R_2(1), R_2(2), R_2(3)\}$.

6.3.2 Granularity of Data Ownership

For each thread, the partition from the above analyses exposes the data ownership information specifying which portion of memory, in terms of region, it accesses frequently. Depending on the situation, regions in the partitions provide the flexibility to represent access patterns at as fine grain as an array element. In a target system, however, data is usually organized at either the cache-line or page granularity. A fine granularity such as the cache-line provides the most accurate ownership information at the cost of expensive computation and storage overhead. Conversely, large granularities reduces overhead but increases the danger of a data block owned by one tile being polluted by others. Another option is to divide the page into sub-pages and assign each sub-page an owner based on the partition from the compiler analyses. Regardless of the granularity used, the requested addresses need to be compared with the address ranges of regions in the partition to determine the ownership.

6.3.3 Calculating Communication Patterns

Data partitioning defines an owner tile for each piece of data of an array in the form of array access regions. When data owned by a particular tile is accessed by remote tile this results in communication in the system. As defined in Section 6.3.1, $R_*(y)$ represents the data region owned by tile y in a partition. Then in the same fashion, if data is distributed to the tiles as dictated by the compiler, then accesses by other tiles to the data in $R_*(y)$ require communication. In order to calculate the volume of these remote accesses, *non-local access weight* is defined as the amount of communication between the two tiles x and y in a particular phase as follows:

Definition 9. Non-local access weight *of processor core within tile x to data owned by tile y during phase i, denoted by $W(x, y, i)$, is the number of times the thread executing on processor core x accesses the data in the intersection of $R_i(x)$ and $R_*(y)$.*

In order to estimate the non-local access weight W, the *ending offset*, *overlap range* and *density* of the region must be computed first. Consider a region $R_i(x)$ that has m span-stride pairs. $S_{ik}(x)$ and $T_{ik}(x)$ are the k_{th} span-stride within the region and $O_i(x)$ is the region's initial offset. The corresponding ending offset $E_i(x)$ is defined as the last element in the region. Specifically:

$$E_i(x) = O_i(x) + \sum_{k=1}^{m} S_{ik}(x) \tag{6.6}$$

The overlap range of $R_i(x)$ and $R_*(y)$ thus can be defined as:

$$\Delta = min(E_i(x), E_*(y)) - max(O_i(x), O_*(y)) \tag{6.7}$$

where $O_*(y)$, $E_*(y)$ are the initial offset and ending offset, respectively of $R_*(y)$.

The region $R_*(y)$'s density $D_*(y)$ is the access weight of the region relative to the range of that region. Thus, $D_*(y)$ is defined as:

$$D_*(y) = \frac{1}{E_*(y) - O_*(y)} * W(R_*(y)) \tag{6.8}$$

Thus, the non-local access weight of tile x to the data in region $R_*(y)$ in program phase i can be approximated as follows ($D_*(y)$ is the density of tile y's region of the partition $\mathbf{P}$):

$$W(x, y, i) = \frac{\Delta}{E_i(x) - O_i(x)} * D_*(y) * W(R_i(x)) \tag{6.9}$$

The amount of communication between two tiles is estimated in the compiler based on Eq. (6.9). The *communication pattern* of the application includes communications from all program phases between every two tiles.

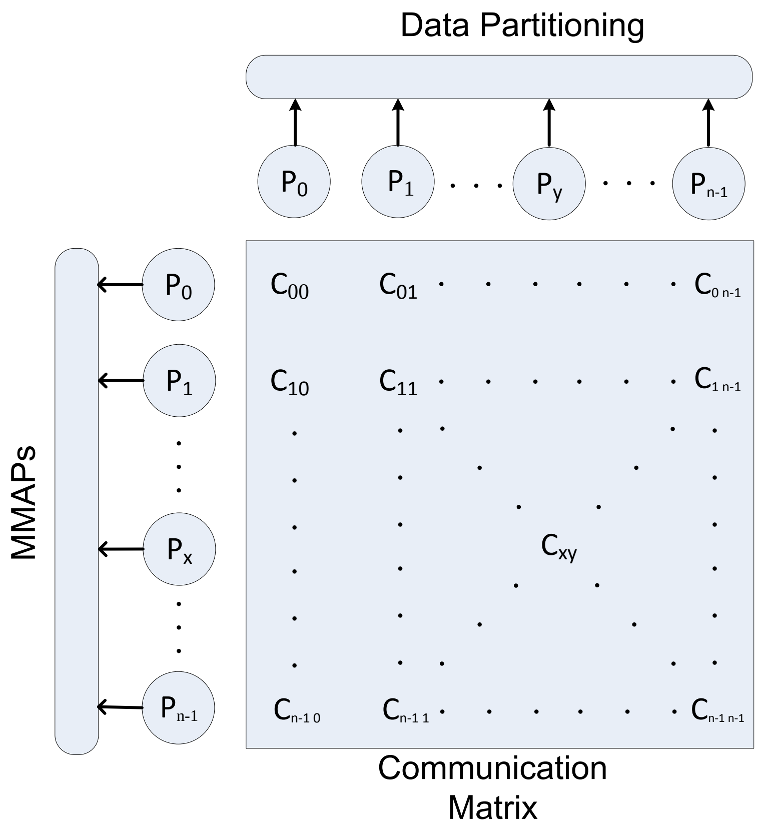


Figure 58: Communication matrix

To represent each communication pattern, a communication matrix is constructed for a parallel program segment with n threads, as shown in Figure 58. Each value in the matrix denotes the communication traffic between two tiles. Based on the techniques introduced in Section 6.2.2, every element in this matrix can be calculated using Eq. (6.10) where ρ is number of MMAPs in the program.

$$C_{xy} = \sum_{i=0}^{\rho-1} W(x, y, i) \tag{6.10}$$

For more complicated applications that contain several distinct segments of the applications with distinct communication patterns it is possible to generate different partitions for different segments of execution using a similar method as described in [88].

6.4 EVALUATION

Since the proposed scheme produces data ownership and communication information from compiler analyses, this section presents the results to demonstrate the compiler's capability of discovering data partitions and communication patterns. Several benchmarks from the SPLASH-2 [5] and PARSEC 2.0 [15] benchmark suites are selected and the affine array accesses in those benchmarks are partitioned using the TI variable based technique (see Section 6.2). Benchmark input parameters are listed in Table 6. Both larger and smaller working sets are used to study the scalability of the compiler partitioning analysis.

Table 6: Benchmark description

Benchmark	Smaller Working Set	Larger Working Set
Blackscholes	4096 options	16384 options
Swaptions	8 swaptions	32 swaptions
Barnes	8K particles	16K particles
Ocean	258x258	514x514
LU	512x512 matrix	1024x1024 matrix
Water Spatial	512 molecules	3375 molecules
Radix	1048576 keys	10485760 keys

6.4.1 Capability in Discovering Ownership

The data ownership analyses are applied to heap memory blocks, which are only part of the overall data space used by programs. Figure 59 shows the proportions of sub-pages that are assigned ownership by the compiler versus those that compiler did not specify an ownership. The proportion of data with ownership is expected to increase as the input working set grows since benchmarks with larger working sets tend to allocate more memory on the heap. On average, 13.8% of all sub-pages used by the tested benchmarks with smaller working sets are assigned ownership by the compiler. In contrast, the weighted average percentage increases to more than 48.2% when using the larger working sets from Table 6, which corroborates the expected outcome.

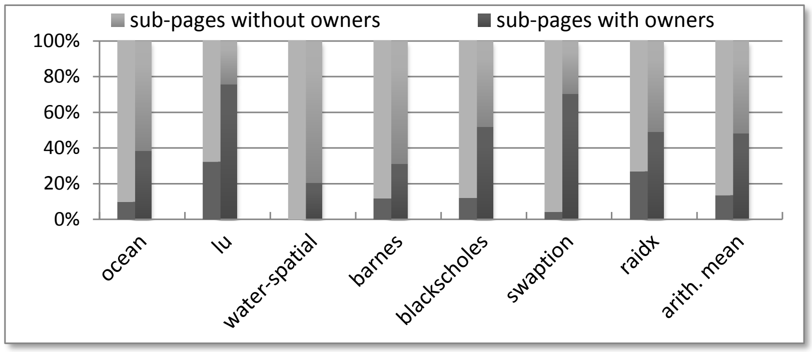


Figure 59: Percentage of sub-pages with owners for (smaller working set on left)

6.4.2 Compiler Communication Pattern Accuracy

The results from the compiler analysis are compared with the actual communications from a multi-threaded memory access trace (MMAT) using the partition provided by the compiler. MMATs are automatically generated for the multi-threaded applications by instrumenting the code with printing instructions inserted for each memory access. The MMAT captures the memory access information, which is used to build communication patterns at runtime. To demonstrate the visual impact and scalability of the compiler approach, the communication patterns are derived for programs with 64 threads (running on a machine with 64 processors).

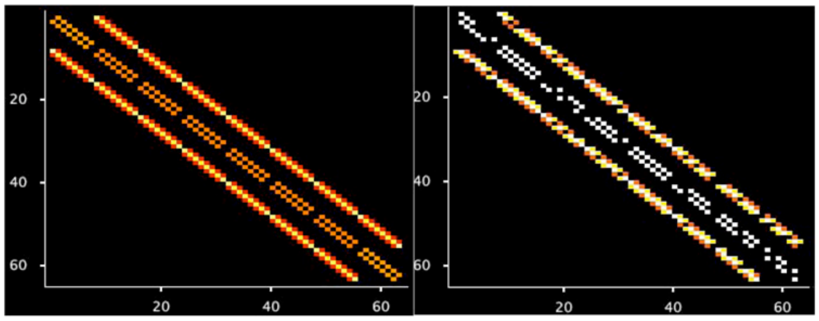


Figure 60: Static vs dynamic communication pattern for OCEAN

Figure 60 compares the OCEAN benchmark where the left matrix is the compiler generated pattern and the right is the runtime generated pattern. Communication is indicated by the intensity of the color, black is negligible communication, and red, orange, yellow, and white is increasingly heavy communication. The communication patterns for LU and WATER (spatial) are shown in Figure 61 and Figure 62, respectively. The communication patterns are also identified for BLACKSCHOLES and SWAPTIONS (not-shown), in which all-to-all communications are observed. Although the patterns obtained from different methods show minor discrepancies in communication weight, the same hot spots and intensive communication patterns are observed.

Not all benchmarks could be analyzed effectively by the compiler. For example, BARNES contains a significant amount of accesses that are dependent on the data-set and could not be handled by a compile-time communication analysis. However, for the data that could be analyzed in BARNES, the compiler discovers an all-to-all communication pattern. In this case the compiler can fall back to the analyses conducted in Chapter 3.

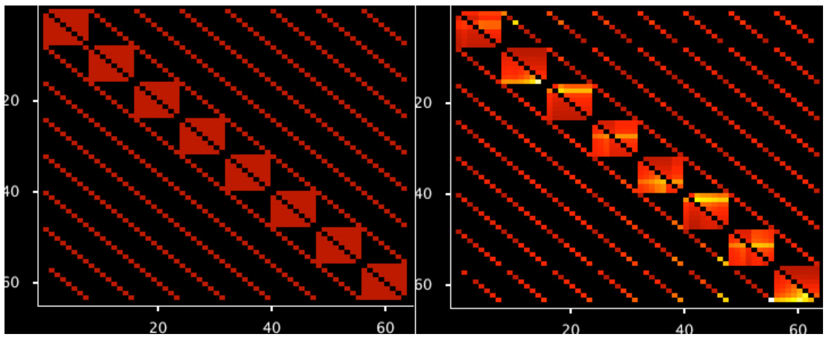


Figure 61: Static vs dynamic communication pattern for LU

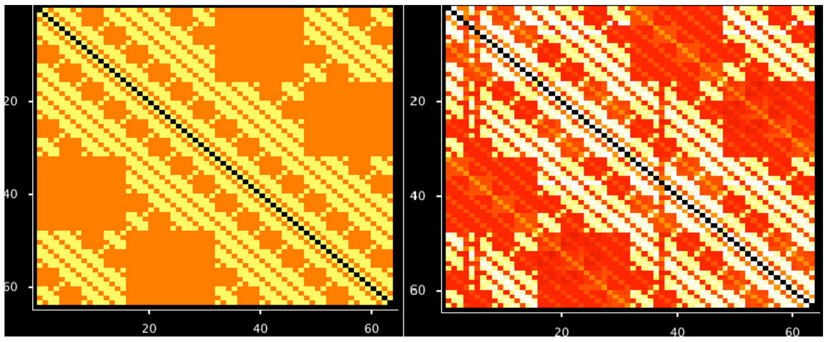


Figure 62: Static vs dynamic communication pattern for WATER-SPATIAL

7.0 UTILIZING COMPILER DETERMINED DATA PARTITIONING AND COMMUNICATION PATTERN IN CMPS

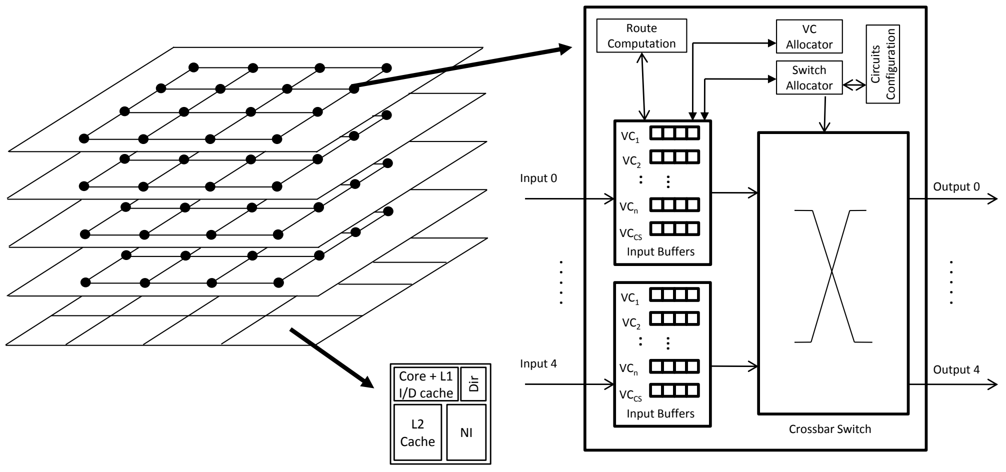


Figure 63: System overview

The compiler techniques introduced in Chapter 6 can be used to determine the data partitioning of an application, which can be passed to the runtime system of a CMP to enforce a data placement in the shared cache with the intent of promoting locality. Based on the partitioning the system can calculate the communication pattern of the application. On a reconfigurable NoC such as the one presented in [1], the communication pattern can be used by the circuit configuration logic, as shown in Figure 63, to guide the runtime system to establish circuits between tiles that frequently communicate. The target CMP system in this work is organized as a two-dimensional grid where each tile contains a processor core, a private L1 instruction and data cache, an L2 cache bank, a directory bank, and a network interface (NI). Tiles may communicate on one of four two-dimensional mesh network planes, each of which combines circuit switching and packet switching. Routers in these planes are detailed on the right of Figure 63. These assumptions are made to ensure reasonable comparisons with the most relevant related work.

7.1 RELATED WORK

In the next two sections, relevant related architectural work is described in some detail. In the first section, cache mechanisms that assign data an owner are considered and compared with the compiler data partitioning approach. In the second section, reconfigurable on-chip interconnection approaches are described that favor distinct point to point communication partners that exhibit heavy traffic volumes. These techniques are compared to the compiler guided interconnection approach discussed in this chapter.

7.1.1 Relevant Runtime Cache Enhancements

One runtime caching mechanism relevant to the architecture optimization presented in this chapter is the recent attempt [81] that has been made to reduce access latency by assigning pages to cache banks within the tile containing the core that touches the page first (*first touch*). A similar first touch scheme is implemented at cache block granularity in Abousamra *et al.*'s work [2]. First touch is a promising caching mechanism since it preserves the locality and reduces remote accesses by absorbing a data block to the local cache bank of the first requesting core, which becomes the owner of that data block. To adapt to dynamic access behavior and enhance performance, data is allowed to migrate to another processor's local cache bank if that processor accesses the data more frequently than the current owner. Counters are used to keep track of the accesses to each cache line by different processors. In first touch caching, a particular cache block can be placed in any location within the aggregate sets of all cache banks. A directory, composed of a tag and a tile number, is required to keep track of the location of each cache block. Any requester (except the owner) needs to consult the directory for the location of the requested block.

Reactive NUCA (R-NUCA) [42], previously discussed in Chapter 2, is another data placement policy based on runtime data access behaviors. The goal of R-NUCA is to reduce data access latency based on page classification. To classify a page, the OS must keep track of the requesters and within each page table entry (PTE) maintain an additional field which is used to store the core ID of the previous requester for that page. Data pages are initially classified as private. Private pages are cached in the local tile to the core that "owns" that page. A data page is reclassified to be shared when the OS detects a requester from a different tile from the one that is recorded. The shared page is then stored in the standard distributed shared location based on the memory address. Naturally, there is an overhead for evicting the appropriate lines from the local tile and either migrating the data or reloading the data from main memory.

The cache/NoC optimization presented in this chapter is compared with the aforementioned first touch mechanism as well as R-NUCA because both of them enforce a unique location (owner) to place a data block, which is similar to the compiler-guided scheme. In contrast to R-NUCA and first touch, the compiler-assisted partitioning (CAP) is used to guide the data distribution. For the data that can not be partitioned through compiler analysis (no ownership information), either the approach presented in Chapter 4 or the first touch policy can be used at the cache-line granularity.

7.1.2 Relevant Reconfigurable Networks

The relevant reconfigurable interconnect systems work on a principle similar to EVCs [62, 59] allowing hybrid circuit-switched (CS) and packet-switched (PS) traffic. PS traffic requires significant overhead at each switch point including stages like virtual channel allocation, route computation, switch allocation, switch traversal, and link traversal. In an EVC-like system, CS traffic pre-establishes a path between source and destination and packets can traverse the intermediate switch points without all of the considerable packet-switched overhead reducing communication latency[1].

Prior to [1], circuits were either established in regular patterns [62] or on-demand [47]. Circuit pinning uses a training period to discover the most important circuits to establish and pin those circuits for a pre-defined time quantum. For traffic that does not have an appropriate circuit to traverse, the standard packet-switch mechanism is used. To further improve the performance, a partial path routing scheme is possible to allow traffic to partially traverse a circuit and if necessary to also partially travel via packet-switch.

[1]In this chapter the terms fast-path and circuit are used interchangeably.

In contrast, the compiler-assisted configuration (CAC) for NoCs determines the communication pattern *a prior* and statically applies the circuit pinning concept. CAC also allows the partial-path routing optimization and avoids the need for a complex centralized runtime system to determine the most heavily used connections for establishment.

7.2 SYSTEM SUPPORT

In order to leverage the partition and communication information from the compiler analyses, a facility is required to communicate the information from the compiler to the run-time system. To enforce CAP, the information about memory regions and their owners must be passed to the runtime system to notify the partition for a particular block of memory. This mechanism requires the usage of memory allocation hooks, which are used to modify the behavior of `malloc()`. Each time the `malloc()` function is called, a customized hook function is invoked to record the return value, the size of the allocated memory block and the partition information into a memory partition table (MP-Table). The hook function for the memory allocator can be specified through the hook variable `__malloc_hook` defined in *malloc.h*. The hook driver is responsible for processing the partition information and determining the owner for pages/sub-pages in the page table. A page/sub-page that cannot be classified by the compiler is not added to the MP-Table and the runtime caching policy (*e.g.* first touch) is applied for the corresponding accesses.

In the experimental architecture, a page is logically partitioned into several sub-pages. Sub-page granularity was selected in order to be sufficiently fine grain to avoid data pollution among multiple owners yet coarse grain enough to reduce storage and owner calculation overhead. The study in Section 7.3.2.3 shows that a number of sub-pages from 2 to 8 yields the best results. To store the owner information for p cores, each sub-page is associated with $\log_2 p$ bits indicating its owner ID. For the 16-core architecture used in our evaluation, four bits are needed to store owner information for each of four sub-pages, resulting in a $4 \times 4 = 16$ bits of additional storage for each PTE. This incurs a 16/64 = 25% overhead on a typical 64-bits PTE structure.

At memory allocation time (when a memory allocator is being called), the hook driver consults the information stored in the MP-Table. To determine an owner for a sub-page, the hook driver calculates the median address ($S + P/2$) where S is the starting address of the sub-page and P is the sub-page size. If the median address belongs to a particular region, the corresponding sub-page is assigned the same owner of that region. Although this scheme is approximate, it is effective because typically regions owned by different threads in the partition do not interleave at a finer granularity.

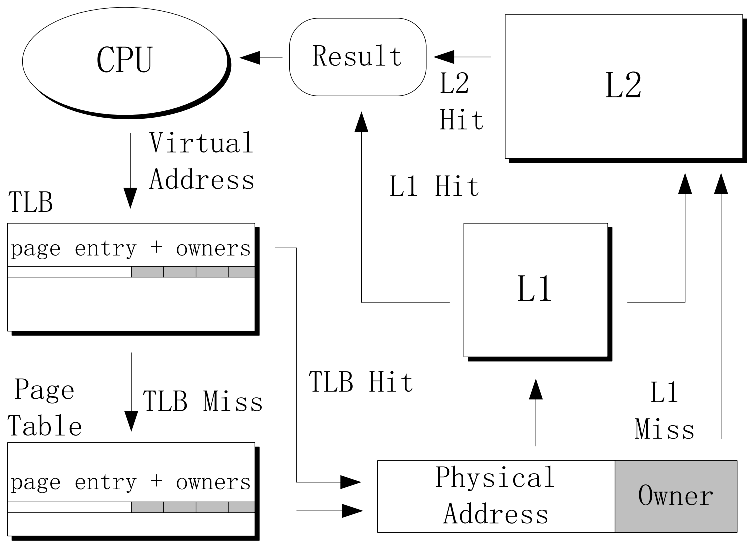


Figure 64: Retrieving ownership during address translation

During virtual address translation using the TLB, the owner of a given address can be retrieved using the information stored in the page-table, thus avoiding the overhead of directory lookup. This is illustrated in Figure 64. As shown in the graph, the physical address needs to be obtained from either the TLB or the page table whenever a processor issues a request. The ownership information stored in the page table entry is then consulted during this process and used to locate or lookup a datum upon a L1 miss.

To leverage the communication pattern from the compiler to reconfigure the NoC the compiler inserts network configuration instructions into the code using a method similar to [88]. By including a new system call (e.g., *reconfigNetwork()*) that takes as a parameter a matrix containing the circuits to establish, an arbitrary configuration can be setup at various points within the application.

7.3 EVALUATION

The impact of the data partitioning on cache miss rate and average memory access latency is studied to show the advantage of using the data ownership information for optimizing cache data placement. Additionally, this section demonstrates the capability of the compiler to approximate a complex runtime method to leverage the reconfigurable hybrid CS/PS NoC in order to save the runtime system overhead. Finally, overall performance gains are reported on a set of benchmark workloads.

7.3.1 Simulation Environment

Wind River Simics [71] is used as the simulation environment in which the relevant caching schemes and NoC schemes (*i.e.* those discussed in Section 7.1.1 and Section 7.1.2, respectively) are implemented. As before, this work does not simulate the effect of replicating application code pages or clustering as described in [42] because it is orthogonal to the data page classification. Simics is configured to simulate a tiled CMP consisting of 16 SPARC 2-way in-order processors, each clocked at 2 GHz, running the Solaris 10 operating system, and sharing a 4 GB main memory with 55 ns (110 cycles) access latency. The processors are laid out in a 4×4 mesh. Each processor has a 32 KB private 4-way L1 cache (divided equally between instruction and data with access latency of 1 cycle). Benchmark input parameters are listed in Table 6. The larger working set is used for the simulation.

For first touch and CAP, the simulated machine has 16, 16-way 1MB banks (access latency: 6 cycles) for a total of a 16 MB L2 cache. The distributed shared cache and R-NUCA use a similar configuration but are provided with an extra 4M capacity as compensation for the directory and storage overhead incurred by the first touch and CAP schemes. For CAP the system uses a classification granularity of four sub-pages (blocks of 1K bytes) unless otherwise stated.

The simulated network is a cycle-accurate packet-switched interconnect with a 64-byte link width. The routers use a 3-stage pipeline. Each port contains four virtual channel buffers, and each buffer is capable of storing four flits. For the hybrid PS/CS the link width is sub-divided into four independent "switch planes." Control messages are one flit long and data messages are four flits long. A PS flit goes through the 3-stage pipeline while CS flits traverse the switch in one cycle. Each port has an additional virtual channel buffer dedicated to CS traffic.

For runtime circuit establishment, circuits are pinned such that it requires 50ns to flush the existing network configuration, 8,000ns to configure the circuits, and the circuits are pinned for 100,000ns [1]. The compiler configured circuits are programmed at application load time and pinned for the entire duration of the application and partial circuit routing is permitted.

7.3.2 Compiler-Assisted Partitioning Performance

To study the performance of CAP compared to R-NUCA and first touch, the cache miss rate, average memory access latency and the overall performance are examined.

7.3.2.1 Impact on Cache Performance Figure 65 shows the cache miss rate normalized to the distributed shared cache. Experiments were conducted with a packet switch NoC. For most of the tested benchmarks, the miss rates are only nominally different. This result was expected since all the tested caching policies do not allow replication eliminating unnecessary capacity misses that might exist in a scheme such as private caches. For some benchmarks, such as LU, WATER, RADIX and BARNES, first touch and CAP increase the miss rate slightly. This is due to the slightly smaller cache capacity of these schemes compared to distributed shared and R-NUCA owing to the storage dedicated to maintaining the directory (see Section 7.3.1).

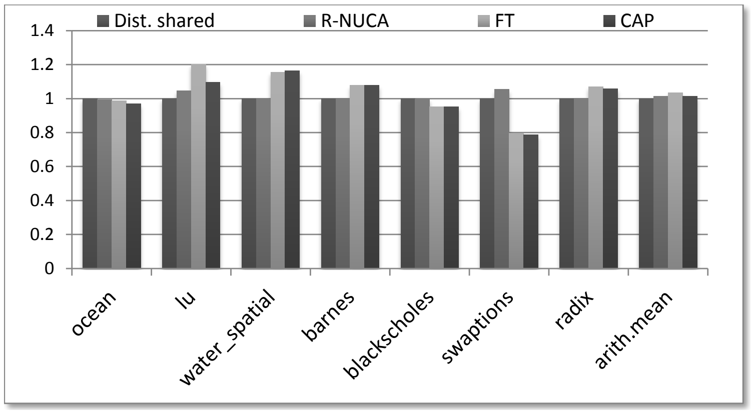


Figure 65: Cache miss rate (normalized to dist. shared)

The memory access latency plays an important role in the performance of CMPs (see Figure 66). Compared with distributed shared, R-NUCA, first touch and CAP reduces the average memory access latency by 2.3%, 26.9% and 31.3%, respectively. Notice that BLACKSCHOLES and SWAPTIONS show a remarkable decrease in memory access latency. Studies show that these two benchmarks are highly parallelized with an extensive amount of local data accesses. The infrequent sharing and high degree of parallelism ensure that these benchmarks perform well with CAP. OCEAN also receives noticeable decrease in memory access latency. This conforms to the fact that OCEAN has a 2-D nearest-neighbor sharing and exhibits a large amount of locality.

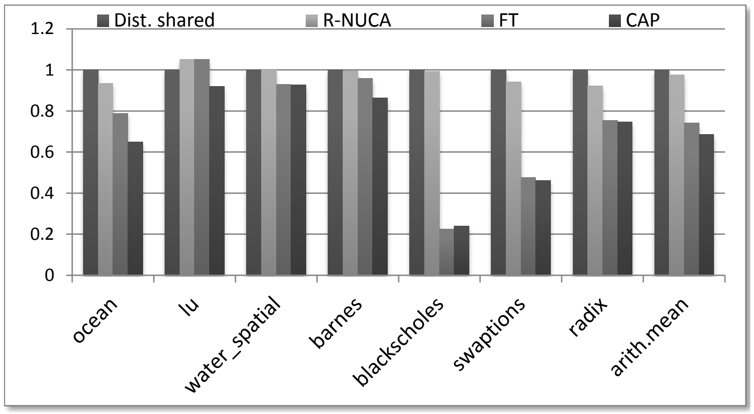


Figure 66: Average memory access latency (normalized to dist. shared)

7.3.2.2 Overall Performance Figure 67 shows that the distributed shared cache has the worst performance due to the effects described in Section 7.3.2.1. BLACKSCHOLES achieves an approximately 40% improvement with CAP. RADIX, SWAPTIONS, BARNES and OCEAN also show decent speedups. WATER and LU only see a nominal improvement due to large amount of shared data. These performance results are consistent with the average memory access latencies from Section 7.3.2.1. On average, CAP achieves a 5.0% speedup over first touch. It also outperforms R-NUCA and distributed shared caches by 19.0% and 20.5%, respectively. R-NUCA does not significantly outperform the distributed shared approach because the predominantly private data is polluted by shared access (see Chapter 4). Thus, in practice, most pages end up classified as shared, caused either by the false sharing within a page, or by initialization and access from more than one threads. Thus, the performance approaches the shared scheme.

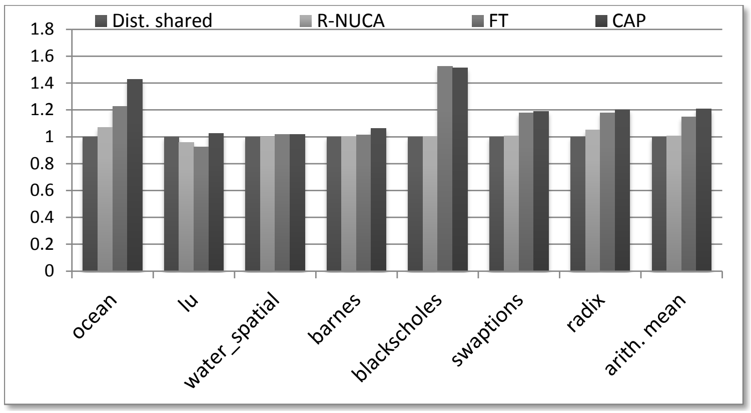


Figure 67: Speedup (normalized to distributed shared)

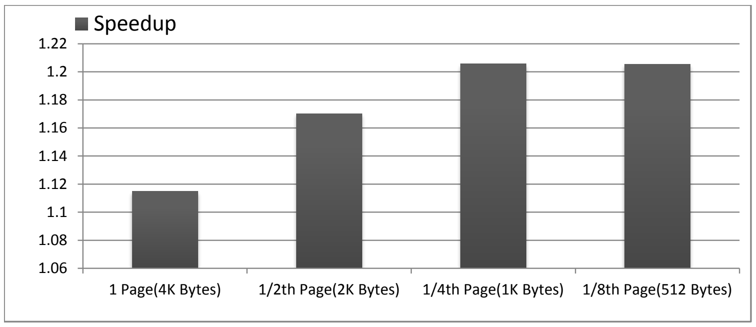


Figure 68: Average speedup of CAP over distributed shared for various block sizes

7.3.2.3 Impact of Partition Granularity

Figure 68 shows the speedup of CAP over distributed shared caching for a granularity of a full page (4K) to $\frac{1}{8}$ of a page (512 bytes). For 4K granularity CAP's benefit is approximately 11.5%, and for 1K sub-pages the benefit nearly doubles. However, experiments show that 1K is a desirable compromise between a full page and a cache line as granularities smaller than 1K do not provide significant performance improvements.

7.3.3 Impact of Compiler Assisted Network Configuration

The impact of using CAC to configure a reconfigurable network is compared with a state of the art runtime circuit establishment technique of pinning with partial path routing (see Section 7.1.2) [1]. The simulated system uses the compiler determined communication pattern to guide the establishment of fast-paths employed for the entire duration of the application.

The final performance evaluation for the system using CAP on the system using the hybrid network using CAC is shown in Figure 69 with the results normalized to a shared cache system using a PS network. The compiler-based approach (CAC) provides a 5.1% average improvement over the compiler-based data partitioning alone with a packet-switch network (PS). The runtime method (PIN) does slightly outperform the CAC method, but by less than 0.5%. A small improvement of PIN over CAC is not surprising as the runtime method has access to the runtime information not available to the compiler. Additionally, PIN adapts to changes in the communication behavior while the compiler establishes fast-paths for the entire execution duration.

In some situations, too much adaptivity as been shown to be counterproductive due to circuit establishment time. In the runtime method [1], circuits remain established for a minimum period of time to avoid the conflicts of on-demand circuit establishment. For BLACKSCHOLES and RADIX, CAC actually outperforms PIN because it changes the circuits too frequently (too adaptive) while CAC scheme pins the circuits according to an identified pattern to achieve better performance. In addition, CAC avoids the need to keep track of the connections to pin at runtime, which requires communication counters for each source destination pair and may not be scalable to a large numbers of cores. Thus, CAC still provides a competitive solution that brings a 5% improvement over the data partitioning scheme with packet switching (CAP-PS) and less than 0.5% performance degradation compared to PIN.

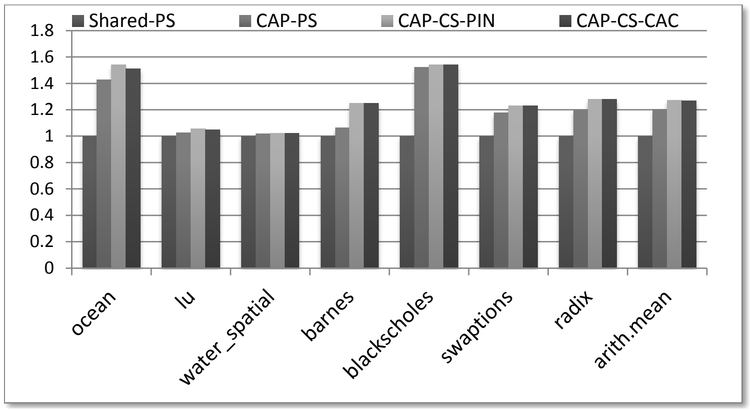


Figure 69: Comparison of application speedup (normalized to Shared-PS)

8.0 OPTIMIZING STT-RAM CACHES

STT-RAM technology has been proposed for use in chip-multiprocessor cache hierarchies as a potential replacement for SRAM particularly for LLC. STT-RAM caches can leverage near SRAM performance for read accesses, non-volatility for reduced leakage power, and increased density and capacity over SRAM. Previous conventional wisdom for STT-RAM is that writes are slower and require more power than their conventional SRAM counterparts. Thus, write performance has been considered the fundamental performance bottleneck and received the focus of attention for optimization. Several architectural solutions such as hybrid caches with fast and slow writing memory components [108, 67], various methods or preempting, avoiding, and bypassing writes [117, 36, 75], and leveraging the asymmetry of writing different logic values [76] have been proposed to mitigate the write performance problem.

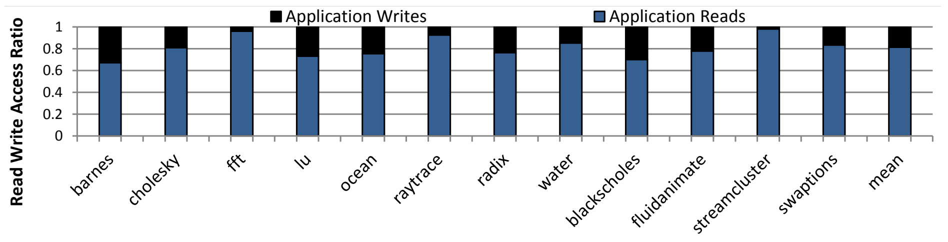


Figure 70: Application read vs writes

A large body of related research, such as those mentioned above, propose hybrid STT-RAM caches due to the complementary characteristics of STT-RAM versus SRAM in an effort to mitigate challenges that prevent the building of an STT-RAM L1 such as the increased write latency and high dynamic power. In particular, two recent techniques to improve the inherent write performance of STT-RAM by tolerating a reduced data retention time [92] have led to proposals for use of STT-RAM at the L1 level [96] where data access speed is crucial.

However, physical effects of technology scaling down to 45nm and below, in particular process variation, introduce potentially alarming trends in **read performance** of STT-RAM due to reduced sensing margins [115], especially at the L1 level. An evaluation of this trend reveals that the read performance problem creates a new bottleneck for application data reads at higher levels of the cache hierarchy, which typically dominate an application's overall data accesses. Figure 70 shows that for various benchmarks the reads contribute an average of 80% of all the data accesses.

The goal of this work is to enable efficient STT-RAM designs at appropriate level(s) of the cache hierarchy by leveraging compiler data reuse analysis to optimize both the write and the read operations. For the conventional write problem, a compiler write reuse analysis is adopted to capture write intensive behaviors and guide the data migration/distribution over a hybrid SRAM/STT-RAM cache. Data blocks with heavily write reuse patterns are migrated to SRAM portion of the hybrid cache to reduce the write penalty. The superiority of the compiler-based approach is that the data migration/distribution decision can be made *a prior*, thus reduces the penalty of expensive writes occurred on STT-RAM during the write-intensity detection process in a hardware-based data migration mechanism. Migrating data based on compiler extracted information also results in more accurate migration decision and avoids temporary misleading access behaviors.

For the read bottleneck, a new compiler analysis, *consecutive read reuse*, is introduced to leverage a dual-mode, differential-sensing-based STT-RAM cell structure. The dual mode STT-RAM structure allows the construction of L1 cache memories in which a cache block that can be configured as a standard block (SB) or through differential sensing configured as a fast read block (FB) at the expense of higher dynamic write power and reduced capacity. The proposed compiler techniques analyze cache read/write accesses and configure memory cells into the appropriate mode to accelerate data reads without incurring excessive write power. In particular, the *consecutively read blocks* (CRBs) can be identified by the compiler and mode switching instructions can be inserted to configure the corresponding cache blocks as FB to accelerate read operations efficiently. In contrast, transient read/write access patterns are serviced in SB to retain low power and high density. The proposed compiler techniques avoid the need for expensive runtime detection techniques in hardware which can add significant complexity and power consumption to the system. In summary, this chapter presents the following contributions:

- This chapter presents several compiler analyses, such as consecutive temporal read reuse (CTR), consecutive spatial read reuse (CSR) and write frequency analysis. These compiler analyses complement the traditional compiler reuse analysis to handle new challenges.

- A hybrid SRAM/STT-RAM cache is designed to provide fast data accesses, low power, and storage density benefits. The STT-RAM portion provides high density and low leakage benefits while the SRAM portion is used to mitigate the high write penalty of STT-RAM. The compiler identified write reuse behavior are used to guide data distribution on the hybrid SRAM/STT-RAM cache to avoid write bottlenecks.

- A low-overhead configurable L1 cache (C1C) architecture is demonstrated in which a data block can dynamically switch between read access modes without data loss. The standard block (SB) has high density and standard dynamic write power but suffers from lower read access speed at scaled technology nodes compared to SRAM. The fast read block (FB) mode offers higher read performance with half the density and doubled write power compared to the SB. The proposed compiler techniques is combined with the configurable cache structure to maximize the read accesses at L1 in FB mode while maintaining a low dynamic write power.

Through the compiler-guided hybrid cache over 5% performance improvement and nearly 10% power saving are achieved compared to the state-of-the-art runtime technique. Additionally, the C1C brings 5% performance gain over SRAM and 10% performance improvement with less than 2% dynamic power increase over STT-RAM designs without read optimizations. In addition, C1C performs closely (within 1.5%) of a static all differential-mode STT-RAM L1 cache with a 26% energy consumption savings.

The remainder of the chapter is organized as follows: Section 8.1 introduces background and related STT-RAM optimizations for write and read operations. In Section 8.2 the compiler techniques are introduced to analyze application read/write behaviors relevant to configurable and hybrid STT-RAM cache optimizations. Section 8.3 introduces the hybrid cache leveraging the compiler-based write analysis and Section 8.4 presents the configurable cache C1C using the compiler read reuse analysis. The performance and power impact of C1C and the hybrid cache are evaluated in Section 8.5.

8.1 STT-RAM TECHNOLOGY TRENDS AND DESIGN

The building block of STT-RAM is the Magnetic Tunnel Junction (MTJ), which contains two synthetic ferromagnetic layers (pinned and free layer) and one MgO-based tunnel barrier layer [45], as illustrated in Figure 71. The magnetic direction of the pinned layer is fixed while the magnetic direction of the free layer can vary through the application of an external electromagnetic field or spin-polarized current through that layer. When the magnetization directions of the two ferromagnetic layers are parallel, the MTJ is in its low resistance state (Figure 71 (A)). In contrast, when the directions of the two layers are anti-parallel, the MTJ resistance is high (Figure 71 (B)). The low and high MTJ resistances can be used to represent logic values. In a typical ”1T1J” [45] STT-RAM cell illustrated in Figure 71 (C), one MTJ is connected with one NMOS transistor, which serves as access controller. This NMOS transistor is typically 1.5 times the size of each of the six transistors that comprise an SRAM cell, leading to the four times density improvement assumed in SRAM replacements with STT-RAM [95].

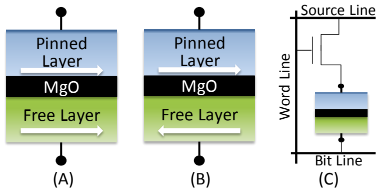


Figure 71: Illustration of an MTJ and STT-RAM cell

8.1.1 Write Optimizations

Writes to the MTJ are completed by applying the write current to the cell for a sufficient duration to set the magnetic direction of the free layer to a particular direction. This action is called a “write pulse.” When the write pulse width is longer than 10ns, the relationship between write current (I_c) and write pulse width (τ) can be expressed by Eq. (8.1) [44]. I_{c0} and τ_0 are the critical write current and the write pulse width, respectively, at $0K$; and Δ is the magnetization stability energy barrier, which determines the data retention performance of a STT-RAM cell, i.e., $T_{retain} \propto e^{\Delta}$ [32]. The Δ is defined in Eq. (8.2), where K_u is the uniaxial anisotropy energy; V is the volume of a ferromagnetic layer (free layer) of the MTJ; K_B is the Boltzmann constant; and T is the working temperature.

$$I_c(\tau) = I_{c0}(1 - \frac{1}{\Delta}\ln(\frac{\tau}{\tau_0})) \quad (8.1) \qquad \Delta = (\frac{K_u V}{k_B T}) \quad (8.2)$$

Based on these MTJ design parameters, there are several ways to improve writability [63, 64, 92, 96]. For example, changing the saturation magnetization, the effective anisotropy field, or the thickness of free layer can lower Δ. In particular, a significantly faster write speed can be achieved at the expense of reduced data retention time and this technique has been demonstrated to enable L1 STT-RAM caches [96].

Hybrid memories can be also utilized for mitigating the high write penalty of on-chip STT-RAM caches. One such solution is constructing a hybrid cache bt integrating a small portion of SRAM on top of a larger STT-RAM cache [94]. In the hybrid SRAM/STT-RAM cache, write intensive data blocks are migrated to SRAM so that . The write behavior can be captured by dedicated hardware logic, as done in many prior research attempts [94, 108, 96].

8.1.2 Read Optimization Using Differential Sensing

Reads are completed by sensing the voltage differential in the two resistance states using a read current (I_r) applied for a particular duration, called a read pulse. For all reads to MTJs, there is a probability of disturbing the stored value (Pr_{dis}) that can be expressed as [44, 22]:

$$Pr_{dis} = 1 - exp\{-\frac{t}{\tau}exp[-\Delta(1 - \frac{I_r}{I_c})]\}. \qquad (8.3)$$

Here, t is the read pulse width. Eq. (8.3) shows that Pr_{dis} is mainly determined by I_r/I_c ratio. STT design usually uses a global read driver to control I_r and supplies a reference voltage (V_{ref}) to differentiate the high- and low-resistance states of a memory cell. The sense margin of the memory cell thereby is proportional to $I_r \cdot \Delta R/2$, where ΔR is the difference between the high- and the low-resistance states. Improving memory access speed by leveraging differential circuit design has been demonstrated feasible by several prior efforts [85, 52].

As the technology scales the energy required for writing ($I_c(\tau)$) decreases. To avoid increasing Pr_{dis}, I_r/I_c must remain balanced, requiring proportional reductions to I_r, which in turn reduces the sensing margin ($I_r \cdot \Delta R/2$). Thus, typical assumption of a 100mV sensing margin required to match the read performance of SRAM, should be scaled to more realistic values such as 80mV at 45nm and 40mV at 22nm.

Sense performance is also heavily impacted by process variation creating a distribution of sensing times. Sense delays such as 50ps reported in the literature [92] often assume the typical case (i.e., the peak of the sense delay curve) and appear quite optimistic, implying no performance difference between the sense margins. To demonstrate this problem is present even at technology nodes as large as 45nm, a sense amplifier simulation was conducted using a popular sense-amplifier design in STT-RAM arrays [24] shown in Figure 72. At 45nm, the scaled read current could result in a peak sense time of 150ps, as shown in Figure 73. However, due to device mismatch and process variation it is necessary to cover the entire tail of the curve to ensure all accesses are correct resulting in a nearly 1.5ns delay for a 80mV margin compared to 650ps for a 160mV margin.

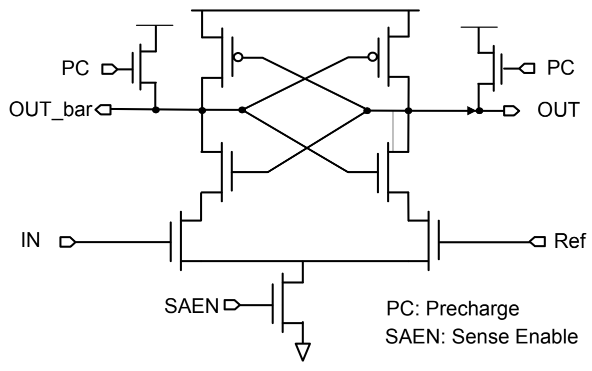


Figure 72: Sense amplifier design

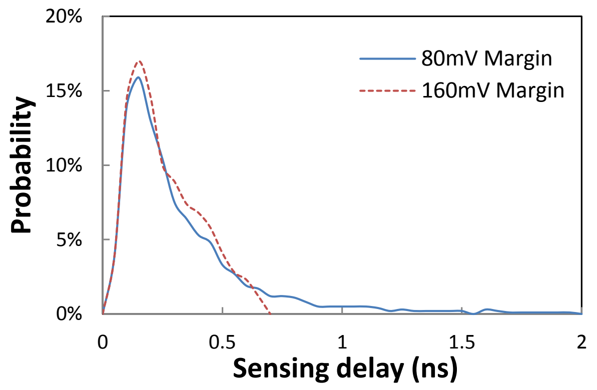


Figure 73: Sense speed distribution

In situations where read performance is critical it is possible to use differential sensing by storing both the value and its complement in adjacent cells in order to double the sense margin ($I_r \cdot \Delta R$) at the expense of reducing storage capacity. As sensing speed is a non-linear function of sense margin, this increase can provide significant improvements in read performance. A reconfigurable circuit (Figure 74) can be configured into a standard high density block (SB) by comparing the selected cell with V_{ref} or a fast access block (FB) by sensing the voltage difference between adjacent cells. To accomplish this, as shown in Figure 74, the mode selection bit `Mode` can be integrated into the source line selection logic avoid any additional latency in the critical path. The red operation demonstrates an SB read where cell 0 is compared against a threshold V_{ref}. The blue operation indicates a FB read where cell $2i$ is compared against cell $2i + 1$.

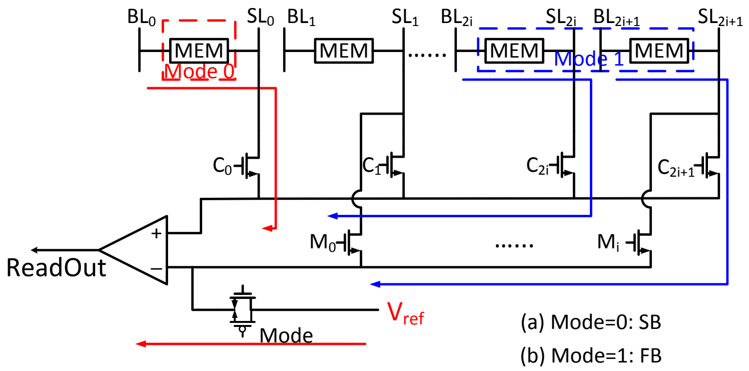


Figure 74: Configurable SB/FB memory circuit

At 22nm technology node, STT-RAM device models are simulated in SPICE to determine the performance and power consumption of individual cells. The sense margins of the sense amplifier in SB and FB are assumed to be 40mV and 80mV, respectively. The latency of SRAM is compared with that of STT-RAM in SB and FB for two potential L1 cache configurations, as shown in Table 7[1]. A scaled version of CACTI [90] is used to derive the peripheral circuitry latencies. The sense amplifier was tuned for best possible performance by sizing of the transistors and timing was derived from an HSPICE simulation.

Table 7: Peripheral circuitry and read latency for two L1 cache examples at 22nm technology

Cache Config	32K 4-Way			64K 4-Way		
Mem Type/Mode	SRAM	STT SB	STT FB	SRAM	STT SB	STT FB
H-Tree (ns)	0.0338	0.0329	0.0329	0.0378	0.0343	0.0343
Dec+Wordline (ns)	0.1523	0.1343	0.1343	0.1698	0.1406	0.1406
Bitline (ns)	0.096	0.0648	0.0648	0.1643	0.1037	0.1037
SenseAmp (ns)	0.116	0.270	0.170	0.116	0.270	0.170
Total (ns)	0.3981	0.502	0.402	0.4879	0.5486	0.4486

In Table 7, the sense delay reports 99.9% of the tail of the delay distribution similar to the curves from Figure 73. These delays comprise a significant portion of the total cache read latency.

[1]As the STT-RAM caches are configurable between SB/FB but two lines are required for FB mode, the capacity reported in the table assumes all blocks are in SB mode.

Although STT-RAM cells are smaller and thus, have faster peripheral circuitry than SRAM, for small caches (e.g., L1) this superiority is negligible and can not offset the negative impact of the larger sense delay. A large sense delay is a prohibitively expensive penalty and would prevent STT-RAM from being used in L1 caches, for which the access speed is extremely important. By adopting FB, the read access latency of STT-RAM once again becomes comparable with SRAM at L1 making it viable in an all STT-RAM cache hierarchy. However, writes to FBs require double the power of SB blocks, which already have a high dynamic write power compared with SRAM.

Thus, for such a configurable cache to be successful, care must be given to the control of the blocks between SB/FB to balance performance, dynamic power consumption and capacity. The following section describes a compiler technique to determine data reuse behaviors that can be used to optimize data accesses in STT-RAM caches.

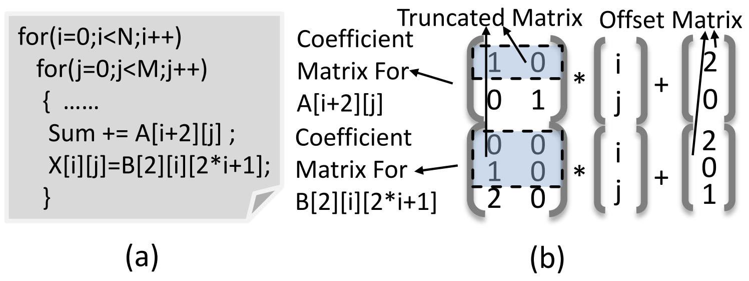
8.2 COMPILER DATA REUSE ANALYSIS

In order to accurately identify data access behaviors to support hybrid and configurable STT-RAM caches, source code level information such as the data reuse behavior can be leveraged by compile-time techniques. Data reuse analysis provides confidence that the data element stored in a particular location will be heavily read or written, indicating that the data should be cached in a certain type of memory or accessed in an appropriate mode. While a simple write reuse analysis is sufficient for determining if a block should be stored in SRAM or STT-RAM in a hybrid cache, it cannot determine if this block should use differential storage. In the later case, heavily written locations stored in differential mode induce a significant dynamic energy overhead due to writing to the complementary cells.

To optimize read performance without increasing the write power, compiler techniques based on data reuse analysis are presented to determine consecutive read reuse (CR), a special type of data reuse that guarantees read intensive behaviors with no interleaved reads and writes. In the following sections, compiler methods for identifying data reuse are described for arrays and linked data structures.

8.2.1 Data Reuse Analysis for Arrays

8.2.1.1 Basic Data Reuse Analysis Data reuse analysis aims to identify array accesses to the same or nearby locations/elements in nested loops and has been used to detect data dependency and promote locality. One approach to analyze data reuse in nested loops is to represent array accesses as matrix multiplication [3]. Consider Figure 75 as an example. Given the array accesses `A[i+2][j]` and `B[2][i][2*i+1]` in the nested loop shown in Figure 75(a), the subscript functions are first converted to the matrix expressions, as illustrated in Figure 75(b). The accessed array elements now can be represented as $C * k + O$, where C is the coefficient matrix, k is the loop index vector and O denotes the offset vector. For the array access to have temporal data reuse, different loop iterations (i.e., different k) need to reference the same array element (the matrix expression $C * k + O$). Therefore, determining whether the array access has temporal reuse now is equivalent to deriving the condition under which the equation $C * k' + O = C * k'' + O$ has solutions [107] (k' and k'' represent two different index vectors in the iteration space). The necessary and sufficient condition under which the above equation has solutions is that C is not fully ranked. In the example, the coefficient matrix of `A[i+2][j]` has a rank of two, indicating no temporal reuse. `B[2][i][2*i+1]` has temporal reuse since the rank of its coefficient matrix is one, which is smaller than its dimension.



```
for(i=0;i<N;i++)
  for(j=0;j<M;j++)
   { ……
    Sum += A[i+2][j] ;
    X[i][j]=B[2][i][2*i+1];
   }
```

Truncated Matrix
Offset Matrix
Coefficient Matrix For A[i+2][j]
Coefficient Matrix For B[2][i][2*i+1]
1 0
0 1
0 0
1 0
2 0
i
j
2
0
2
0
1
(a)
(b)

Figure 75: Array accesses and the corresponding matrix representations (a): array accesses (b): matrix representation

A data access exhibits spatial reuse when the innermost enclosing loop index varies only the last coordinate of that array. To discover spatial data reuse, a truncated coefficient matrix (dropping the last row of the original coefficient matrix) can be used, as illustrated in Figure 75(b). If the rightmost column in the truncated coefficient matrix (the coefficients that correspond to the innermost loop index) is a null vector and the rightmost element in the dropped row is nonzero, it is assured that the innermost loop only varies the last coordinate of the corresponding array.

In the above example, `A[i+2][j]` exhibits spatial reuse since the rightmost column in the truncated matrix (the coefficient corresponding to the innermost loop index j) is a null vector and the rightmost element in the dropped row is nonzero. Using the same rule it can be determined that `B[2][i][2*i+1]` does not have spatial reuse since the innermost loop index j does not vary in the last coordinate of array `B`.

The above data reuse analysis can be used to identify read and write reuse, depending on where the array access appears in the program expression. An application of this analysis is to determine locations with frequent writes for mapping to the SRAM portion of a hybrid SRAM/STT-RAM cache.

8.2.1.2 Consecutive Read (CR) Analysis

Given a location with heavy read reuses, the next step is to determine if these read reuses are consecutive. Read reuses become consecutive if there is no interleaved writes among the reuses. Consecutive reads can be identified by adding additional constraints on the basic temporal and spatial data reuse analyses for reads. First, a data block is defined to have *consecutive temporal read reuse* (CTR) if the same address in the block is read multiple times without interleaved writes on the same block. Similarly, a data block is defined to have *consecutive spatial read reuse* (CSR) if nearby addresses within the block are read with no interleaved writes on the same block. A block exhibiting either CTR or CSR is a good candidate for being promoted to FB for read optimizations without causing high dynamic write power. The compiler analyses for identifying CTR and CSR are presented in the following sections.

8.2.1.3 CTR Analysis

Identifying CTR is similar to analyzing temporal read reuse except that writes that potentially break CTR should be taken into consideration. A read to an n-dimensional array A within a m-deep loop nest has the form of $... = A[fr_1(L)]...[fr_n(L)]$, where $fr_*(L)$ are the subscript functions for the array read defined on a set of loop indices $L = i_1, ..., i_m$, from the outermost to the innermost. The corresponding lower bounds and upper bounds of these indices are $l_1, ..., l_m$ and $u_1, ..., u_m$, respectively. One special scenario of temporal read reuse is when array accesses exhibit temporal read reuse over the innermost loop nest. This is the case when the same address is accessed many times because the innermost loop iterates over its index i_m and the reuse distance [33] is smaller than $u_m - l_m$. This special type of read reuse is denoted as *i-reuse*. Other temporal reuse (over outer loops) is denoted as *o-reuse*. An i-reuse can be identified by examining whether the rightmost column of the coefficient matrix of an array access is all zeros. This indicates that the array access is invariant to the innermost loop index. Take Figure 75 as an example, the array access `B[2][i][2*i+1]` exhibits i-reuse since its coefficient matrix has an all-zero rightmost column and thus the same element of `B` is accessed repeatedly as the innermost loop index j iterates.

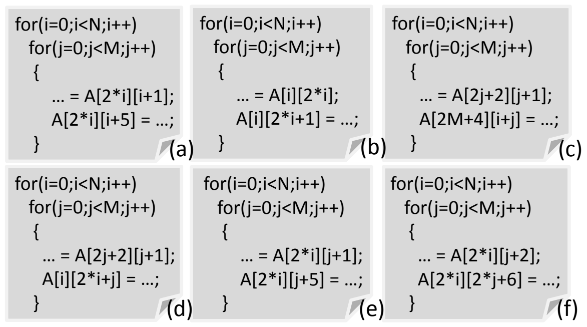
Array reads with i-reuse are temporally close to each other as the same array element is reused multiple times during innermost loop iterations. Thus, the read reuse distance (RRD), essentially the number of loop iterations between read accesses, is small for reads with i-reuse. In other words, it is less common for this type of read reuse to be interleaved by a write and thus i-reuse tends to result in CTR.

However, it is still possible for a write, with the form of $A[fw_1(L)]...[fw_n(L)] = ...$, where $fw_*(L)$ are the subscript functions for an array write in a similar fashion as described for reads above, to break the CTR pattern if the following conditions are satisfied:

$$\begin{aligned} &fw_1 = fr_1, fw_2 = fr_2, ..., fw_{n-1} = fr_{n-1} \text{and} \\ &|fw_n - fr_n| \equiv c < T (\text{c is a constant}) \end{aligned} \tag{8.4}$$

The first condition in Eq. (8.4) ensures that the read and the write index the same locations at all but the last dimension (n_{th}) of the array A. Given the first condition is met, the second condition $|fw_n - fr_n|$ calculates the *read-write distance* (RWD) over the last dimension and examines if this distance is a constant and has an absolute value smaller than T. If the absolute value of RWD is a constant smaller than T, then the write consistently occurs less than T elements on the array away from the read and thus, is likely to break the CTR on a cache block at runtime. If the RWD is not a constant or larger than T, then there is typically CTR as the write will be far enough away from the consecutive reads that exhibit i-reuse.

For example, consider the array accesses in Figure 76 assuming a value $T = 3$. In Figure 76(a) the value $|fw_n - fr_n|$ is a constant 4 and the analysis would determine that CTR does exist, as illustrated in Figure 77(a). In contrast, Figure 76(b) produces a constant $|fw_n - fr_n|$ value of 1, indicating CTR broken by a nearby write, as illustrated in Figure 77(b).



```
(a)
for(i=0;i<N;i++)
  for(j=0;j<M;j++)
  {
     ... = A[2*i][i+1];
     A[2*i][i+5] = ...;
  }
```

```
(b)
for(i=0;i<N;i++)
  for(j=0;j<M;j++)
  {
     ... = A[i][2*i];
     A[i][2*i+1] = ...;
  }
```

```
(c)
for(i=0;i<N;i++)
  for(j=0;j<M;j++)
  {
     ... = A[2j+2][j+1];
     A[2M+4][i+j] = ...;
  }
```

```
(d)
for(i=0;i<N;i++)
  for(j=0;j<M;j++)
  {
     ... = A[2j+2][j+1];
     A[i][2*i+j] = ...;
  }
```

```
(e)
for(i=0;i<N;i++)
  for(j=0;j<M;j++)
  {
     ... = A[2*i][j+1];
     A[2*i][j+5] = ...;
  }
```

```
(f)
for(i=0;i<N;i++)
  for(j=0;j<M;j++)
  {
     ... = A[2*i][j+2];
     A[2*i][2*j+6] = ...;
  }
```

Figure 76: CTR and CSR code examples

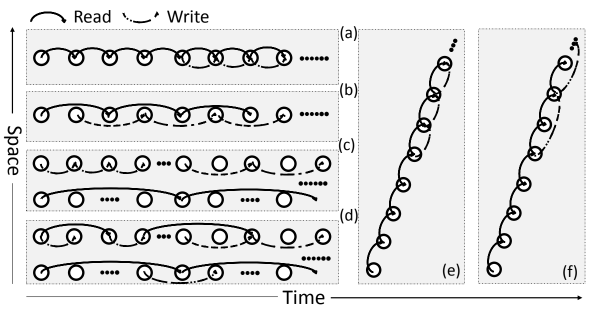


Figure 77: CTR and CSR access patterns

For o-reuse, the RRD is typically much larger than that of an i-reuse and thus several reads with o-reuse are more likely to be interrupted by a write. An o-reuse can be determined to be CTR if there is no write to the same array location during that loop (and all contained within it) that overlaps with the read using the following conditions:

$$fw_{1_{max}} < fr_{1_{min}} || fw_{1_{min}} > fr_{1_{max}} || ... || \\ fw_{n_{max}} < fr_{n_{min}} || fw_{n_{min}} > fr_{n_{max}} \tag{8.5}$$

In Eq. (8.5), $fw_{*_{min}}$, $fw_{*_{max}}$,$fr_{*_{min}}$ and $fr_{*_{max}}$ can be calculated from the lower and upper bounds l_* and u_* of the corresponding loops. The condition in Eq. (8.5) guarantees that the indexed locations of the write are entirely out of the scope of the locations accessed by the read for at least one dimension of the array. An o-reuse induced CTR example is shown in Figure 76(c) in which the array write `A[2M+4][i+j]` falls out of the range of the array read `A[2j+2][j+1]` for any possible j value within the loop bounds. In Figure 76(d) the read and write could potentially interfere with each other and thus no CTR is detected. The access patterns for the above two examples are illustrated in Figure 77(c) and Figure 77(d), respectively.

8.2.1.4 CSR Analysis Analyzing CSR requires identifying writes that interfere with spatial read reuse. Given an array access with spatial read reuse in a loop nest, CSR exists if there are no writes on the same array in the loop. In the presence of potential interfering writes, CSR exists if one of the following two conditions are met (given the same notation used in CTR analysis):

- The read and write do not overlap, as can be verified by Eq. (8.5).
- The subscript functions satisfy:

$$\begin{aligned} &fw_1 = fr_1, fw_2 = fr_2, ..., fw_{n-1} = fr_{n-1} \text{ and} \\ &|fw_n - fr_n| > T \end{aligned} \tag{8.6}$$

Note that the condition $|fw_n - fr_n| > T$ in Eq. (8.6) ensures that the RWD is large enough that the write is never close enough to the read to break the CSR as the innermost loop iterates. This condition can be verified in the following three cases:

First, when the RWD in the last dimension $|fw_n - fr_n|$ is a constant, its absolute value should be larger than the threshold T:

$$|fw_n - fr_n| \equiv c > T(\text{c is a constant}) \tag{8.7}$$

If Eq. (8.7) is satisfied, the RWD for the last dimension of the array is consistently large enough as the loop iterates thus CSR can be assured. An example of this scenario is presented in Figure 76(e) and Figure 77(e), where the write `A[2*i][j+5]` is four iterations away from the read `A[2*i][j+1]`, which is $> T$ if $T = 3$.

Next, when fw_n is larger than fr_n at the innermost loop lower bound l_m, $|fw_n - fr_n|$ should be a monotonically increasing function on the innermost loop index i_m:

$$\begin{aligned} fw_n|_{i_m=l_m} - fr_n|_{i_m=l_m} > T \text{ and} \\ (fw_n - fr_n)|_{i_m=x} > (fw_n - fr_n)|_{i_m=y} \forall x > y \end{aligned} \tag{8.8}$$

The first condition in Eq. (8.8) ensures the RWD is larger than T when the innermost loop starts to iterate (index i_m equals to the lower bound l_m). The second condition ensures that the RWD keeps increasing as the innermost loop iterates (RWD increases as i_m increases) so that the RWD becomes larger and larger than T, resulting in CSR. An example of this scenario is presented in Figure 76(f) and Figure 77(f). At $j = 0$ the last index of the write is 6 and the read is 2 with a distance of 4, and as j increases the $2 * j$ factor increases this distance (5 for $j = 1$, 6 for $j = 2$, etc.) thus the distance will always be $> T$.

Finally, when fw_n is smaller than fr_n at the innermost loop lower bound l_m, essentially the converse of Eq. (8.8), $|fw_n - fr_n|$ should be a monotonically decreasing function on the innermost loop index i_m as in Eq. (8.9).

$$\begin{gathered} fw_n|_{i_m=l_m} - fr_n|_{i_m=l_m} < T \text{ and} \\ (fw_n - fr_n)|_{i_m=x} < (fw_n - fr_n)|_{i_m=y} \forall x > y \end{gathered} \tag{8.9}$$

8.2.2 Data Reuse Analysis for Linked Structures

Because linked data structures are not typically allocated consecutively in memory, determining data reuse can be reduced to identification of spatial reuse, which is common when several nearby fields in an object are accessed consecutively. To analyze the spatial reuse for linked data structures such as linked lists and trees, a CFG (control flow graph) of the program is constructed. As before, a CFG $G = (V, E, r)$ represents a directed graph, with nodes V, edges E, and an entry node r. Each node v in V is a basic block, which consists of a sequence of statements that have exact one entry point and exit point. To simplify the code structure, a series of traditional compiler optimizations such as expression folding and branch elimination are applied on the CFG. To determine spatial reuse the CFG is traversed while the following rules are examined:

- The analyzed memory accesses are common pointer based dereferences. That is, these memory accesses only differ in their offsets from a common base pointer.
- There are at least T data accesses whose offsets fall into a specified address range[2]. This is to guarantee the memory accesses are within small scope in the address space.
- There are no function calls or writes within the same block range amongst the analyzed accesses.
- The accesses comprising the spatial reuse sequence are either in the same basic block or in a set of direct successor basic blocks that meet these criteria. If there are conditionals, the second criterion must be satisfied in all branches.

[2]This range depends on the size of the cache block. For example, in a cache with 64-byte block size, this range is 64-bytes.

Algorithm 2: Pseudocode for spatial reuse identification of linked data structures in CFG $G(V, E, r)$

```
begin
    for each basic block $b_i \in V$ do
        create table $H$; $phase = 0$;
        for each statement $s_j \in b_i$ do
            if $s_j$ is function call then
                $phase++$;
            else
                get the base pointer $bp$ of $RHS(s_j)$; get the offset $o$ of $RHS(s_j)$;
                if $\exists$ entry $Y \in H$ such that $bp = Y.bp$ && $phase = Y.phase$ then
                    append $o$ to $Y.r$;
                else
                    create a new entry with $o$, $phase$ and $bp$ and push it into $H$;
                get the base pointer $bp'$ of $LHS(s_j)$;
                get the offset $o'$ of $LHS(s_j)$;
                if $\exists$ entry $Y \in H$ such that $bp' = Y.bp$ && $phase = Y.phase$ then
                    append $o'$ to $Y.w$;
        traverse $H$ and mark spatial reuse for each entry $Y$ with more than $T - 1$ read offsets
        $Y.r$ in a single phase that are within the specified range without an interrupting write
        offset $Y.w$;
        for each unmarked entry $Y' \in H$ do
            for each block $b_j \in \bigcup SUCC(b_i)$ do
                search $Y'.bp$ in the first phase in $b_j$ and compute the total number of offsets $n$
                that are within the specified range;
                if $n < T$ then
                    continue to process next unmarked entry;
            mark entry $Y'$;
```

These rules guarantee the analyzed accesses are mapped to the same memory block and result in consecutive access behavior at runtime. The pseudocode for identifying spatial reuse in basic blocks and their successors is presented in Algorithm 2. It iterates over each basic block and collects relevant information on memory accesses (i.e., base pointers and offsets). It organizes the collected information from different phases into a table, where each function call initiates a new phase. At the exit of each basic block, the table is traversed and the corresponding entries are marked to indicate the identified spatial reuse. For unmarked entries in the table, the first phases of all direct successors of the current basic block are further analyzed for potential spatial reuse across basic blocks in the CFG.

Figure 78 provides various cases to explain the spatial reuse identification for linked structures. As defined in Figure 78(a), the pointer *nd* is declared to point to a data structure with the type *node_t*. Since the data members *x*, *next* and *prev* have integer/pointer type and are adjacent fields in the same data structure, they are consecutive in the address space and thus, would typically reside in the same memory block such as a cache line. In the case of Figure 78(b), the three memory accesses in the same basic block (i.e., *X=nd->x*, *A=nd->next* and *B=nd->prev*) have the common base pointer *nd* and there is no interleaved function calls or writes. Thus, Figure 78(b) exhibits spatial reuse. The program in Figure 78(c) also has spatial reuse since both successors of the basic block *X=nd->x* lead to spatial reuse. Figure 78(d) and Figure 78(e) do not exhibit spatial reuse due to the presence of the function call *foo()* and write *nd->next=0*. In Figure 78(f), one of the direct successors *A=nd->next* only has one common pointer based read and thus, will not be marked as having spatial reuse.

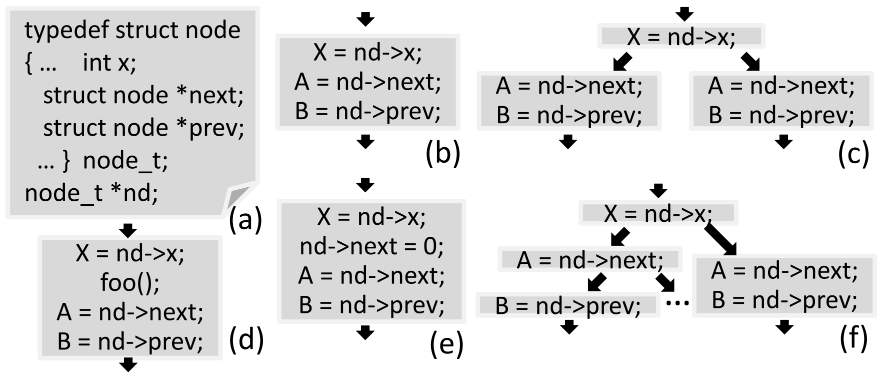


Figure 78: Code and control flow graph examples for spatial reuse identification (T=3). (a): type definition code (b): spatial reuse in the same basic block (c): spatial reuse across one basic block and all its successors (d): spatial reuse broken by function call (e): spatial reuse broken by write (f): spatial reuse broken by one successor

8.3 HYBRID SRAM/STT-RAM CACHE DESIGN

The availability of the emerging non-volatile memory STT-RAM brings the potential for building a rich set of hybrid memory systems [108, 94] that trade off among access speed, power and density. Due to the relatively high penalty of write accesses, caches utilizing STT-RAM typically need to incorporate a certain amount of SRAM for data that is write intensive.

Since the STT-RAM is *write-hostile* and the SRAM is *write-friendly*, it is desirable that write intensive data be dynamically migrated or swapped from the STT-RAM to the SRAM. A commonly used hardware approach is to keep track of a short history of write accesses using a counter and migrate/swap a data block from the write-hostile memory to the write-friendly memory if the counter indicates the data block is write intensive. Particularly in *Sun et al.*'s [94] approach, data is migrated from STT-RAM to SRAM upon two successive writes. Since the hybrid memory is typically applied for LLC, the write through policy is adopted so that the successive writes behavior can be captured by the hardware counters attached to the LLC blocks.

The abovementioned hardware approach can be easily misled by unpredictable runtime data access behavior. Significant mispredictions can incur an expensive penalty of serving accesses in the wrong type of memory, resulting in a high migration/swap overhead. In contrast, leveraging the compiler analysis introduced in Section 8.2 has the advantages of taking actions preemptively to hide the migration/swap latency from the critical path and detecting data access patterns more accurately compared to simple run-time approaches.

8.3.1 3-D Stacked Architecture with Hybrid Cache

Constructing the hybrid SRAM/STT-RAM cache can be achieved by leveraging the 3D integration technology, particularly through silicon vias (TSVs) [84], to stack two device layers (i.e., SRAM and MRAM layer) vertically [94]. 3-D integration offers numerous advantages over the traditional manufacturing process: (1) shorter global interconnects with a vertical distance between 10μ and 100μ [109]; (2) lower interconnect power consumption due to the short wire length; (3) denser form factor and smaller footprint; (4) feasibility to integrate distinct technologies such as those used to construct magnetic cells and CMOS logic.

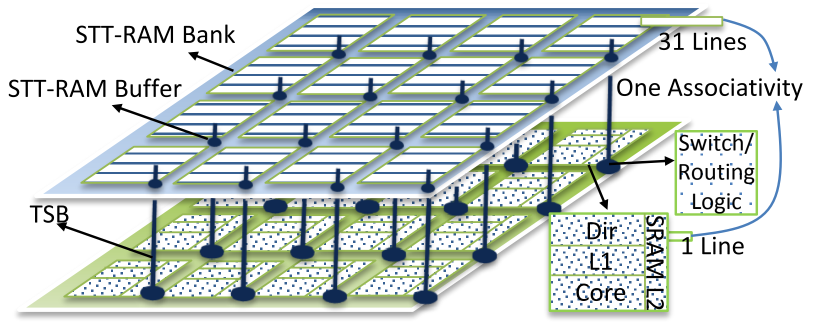


Figure 79: 3-D Architecture with hybrid SRAM/STT-RAM caches.

As Figure 79 shows, the hybrid cache is organized as a tiled architecture with the top layer consisting of STT-RAM LLC banks and bottom layer consisting of CMOS components including processing cores, L1 caches, switch/routing logic, a coherence directory and a SRAM LLC cache bank. Thus, each tile's LLC has $32 - way$ associativity where 1 line is from SRAM and 31 lines are from STT-RAM. On the STT-RAM layer, each L2 bank is equipped with an STT-RAM buffer to store incoming data blocks. Each switch on the CMOS layer connects one core to its neighbors as well as a TSB (through silicon bus), which leads to a corresponding STT-RAM bank. For a particular core in the CMOS layer, the corresponding STT-RAM bank on the other layer acts as its local LLC cache, which requires one extra cycle to access through the TSB traversing. This brings the benefits of fast line migration, message exchange and access time balance between the two layers.

Once a write reuse has been determined by the compiler, a *pre-dispatch* instruction is inserted into the code prior to the memory access to notify the CPU to perform the migration or swap operation. The pre-dispatch instruction can be implemented using the extra bits in the instruction opcode of a particular ISA. For example, in SPARC, the *prefetch* instruction provides dedicated field *fcn* to implement variants of prefetch instruction. The *fcn* value from 16 to 31 is currently reserved and can be used to implement the pre-dispatch instruction. In the ARM architecture, there are similar reserved bits that can be used to implement the migration/swap operation.

8.4 DUAL-MODE CACHE DESIGN

Based on the STT-RAM technology trends and the configurable STT-RAM memory cell from Section 8.1.2, this section presents the configurable L1 cache (C1C) architecture that can be adapted dynamically at the cache block granularity to offer fast read accesses while minimizing dynamic power overheads. The operational mode of the cache lines will be modified based on application needs determined through compiler analysis. The information will be passed from the compiler to the runtime system via mode configuration instructions instrumented into the code by the compiler. The next several subsections describe these elements in detail.

8.4.1 C1C Architecture

In the C1C design illustrated in Figure 80, two adjacent cache lines within one cache set, which independently can operate in standard (SB) mode (i.e., SB1 and SB2) are grouped to form one superline that can operate as an FB line. Each superline contains two respective tags (T1 and T2) and valid bits (V1 and V2) to allow independent SB operation. A mode bit (M) indicates whether the line is storing one FB value or two independent SB values.

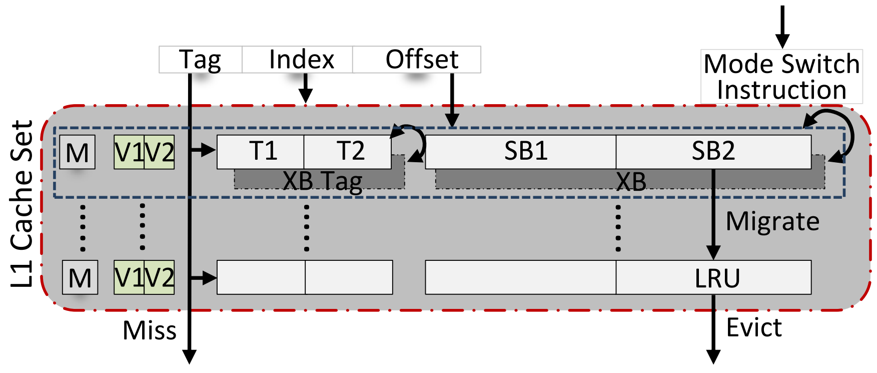


Figure 80: Configurable L1 cache architecture (C1C)

Standard cache operations for a statically designed STT-RAM cache can be accomplished in the C1C architecture with only minimal modification. For example, a cache lookup starts by comparing the tags. In C1C, the M bit can be accessed in parallel with the tag lookup and the resulting data access conducted in the appropriate SB or FB mode. If the cache access is a read and the M bit indicates the target block is FB, differential sensing will be used to more quickly read out the value stored in the two adjacent SB blocks.

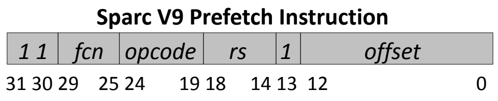
The C1C cache uses the LRU eviction policy. However, the use of two adjacent lines to store a single FB value requires a slight modification to this policy. If a superline wishes to promote an SB line into FB, an eviction is required. First, the LRU line within the set is identified. If this block is the adjacent SB value in the superline, it is evicted. Otherwise, the LRU line from a different superline is evicted and the adjacent SB value is migrated to the LRU value's previous location. Then the promoted SB line's complementary value is written to the adjacent line as depicted in Figure 80. In contrast, when a line is demoted from FB to SB (often from a period of heavy write behavior) the adjacent SB line becomes vacant and can be used to host a new value without requiring an eviction. Unlike the hybrid SRAM/STT-RAM design that uses a write through policy, the C1C design adopts a write back policy to filter writes from the LLC.

8.4.2 Design Considerations

The mode change operation described in the previous section results in some operational overheads that must be considered. As previously mentioned, a write on the same FB block requires twice the dynamic power as a standard write operation due to the writing of the value and its complement. Therefore, it is desirable that an FB line services as many reads but as few writes as possible. Additionally, promoting an SB line to an FB line also results in the overhead of two additional writes (in the worst case) and should only be done when there is high confidence that many successive reads will utilize the line.

An analysis of numerous applications indicates that L1 caches are particularly sensitive to access latency and dynamic power rather than capacity. To build an effective high performance and low power L1 cache, nearly all read accesses in L1 must occur in FB to be competitive with the performance of SRAM. FB reads do not require additional dynamic power as the read current is still only injected once for a differential comparison. However, unlike lower levels in the cache, the number of L1 writes is significant—nearly 20% of accesses on average. While the write performance in FB is not degraded in comparison with SB as both the cell and its complement can be written in parallel, writes to STT-RAM, even with reduced retention times, still require a significantly higher dynamic power than equivalent technology SRAM and about twice as much dynamic power than SB writes. An effective configurable L1 design should maximize the number of writes completed in SB and reads in FB to avoid dynamic power overheads and improve performance, respectively.

According to a study of various benchmarks, even for heavily written data locations, small numbers of writes are typically interspersed with small numbers of reads. During other application phases, the same location is often extremely heavily read. Thus, to avoid excessive dynamic power for complementary writes and mode re-configurations, a line should be configured into FB only if the subsequent access pattern exhibits large amount of consecutive reads without frequently interleaved writes. Thus, the optimization criteria to control the mode of the cache superline is to maximize the number of reads using differential mode (FB) while maximizing the number of writes to standard mode (SB) while minimizing the number of mode switches from SB to FB. The CTR and CSR analyses from Section 8.2 are used to identify candidates for FB mode in C1C. Mode switch instructions are automatically inserted into the code by the compiler to ensure that data locations with CTR and CSR are promoted to FB mode while heavily written blocks operate in SB mode.



fcn code	function
0	Prefetch for several reads
1	Prefetch for one read
2	Prefetch for several writes
3	Prefetch for one write
4	Prefetch page
5-15	*Reserved*
16-31	*Implementation-dependent*

Figure 81: Sparc V9 prefetch instruction format

Thus, the C1C configuration policy relies on an accurate prediction of CRBs in the running applications The cache block mode is controlled by mode switch instructions (MSIs) inserted by the compiler, whose operands represent the target address for the mode configuration. By utilizing compiler analysis to determine the operational mode for lines it is possible to circumvent considerable extra hardware overheads required by runtime solutions such as per/line counters, comparators, and mode-switch logic while avoiding potential thrashing possible in runtime solutions. The compiler guided mode switch instruction can be implemented using the unused bits in the existing instruction thus avoiding modification to the standard instruction set architecture. For example, the Sparc V9 processor's prefetch instruction contains a *fcn* field in its instruction code that is used to implement prefetch variants. As Figure 81 illustrates, each *fcn* value indicates a different operation and the values 5 ∼ 15 are reserved for future usage and thus can be used to implement the mode switch instruction. The next section describes a compiler analysis methodology that enables insertion of these MSIs with a high accuracy.

8.5 EVALUATION

In this section, the effectiveness of the compiler-based access frequency analysis is evaluated for the hybrid cache and C1C cache in an experimental system consisting of 16 cores laid out as a 4×4 mesh. The latency, power and area of the simulated caches are derived from HSPICE simulation and a modified CACTI [90]. The input workloads are selected from the SPLASH-2 [5] and PARSEC [15] benchmark suites. The experimental system is simulated using Wind River Simics [71].

8.5.1 Hybrid Cache Evaluation

In the hybrid cache configuration the L1 cache is purely SRAM while the L2 utilizes both SRAM and STT-RAM. The SRAM and STT-RAM layers are stacked using 3D through silicon vias (TSVs) [84] technology, as illustrated in Figure 79. The compiler-guided data distribution (denoted as SPD) is compared with the mechanism that migrates data from STT-RAM to SRAM upon two successive writes (denoted as MSW).

In the experiment the L2 adopts write-back and the L1 uses write-through. The L2 STT-RAM employs a "same area" replacement resulting in a 512KB per bank capacity, or $4X$ the size of the SRAM. The simulated architectural parameters are detailed in Table 8.

Table 8: Hybrid cache architecture configurations

	SRAM L2	STT-RAM L2
Processor	16 SPARC cores, 2G Hz, 4W/core	
Operating System	64-bit Solaris 10	
L1 Cache	16KB/core, 4-way associative, 64B block size, 1-cycle hit latency, write-through	
L1 Coherence	MESI protocol, in cache directory	
L2 Cache Basics	128KB/bank, 64B, 32-way	512KB/bank, 64B, 32-way
L2 Latency	4-cycle read, 4-cycle write	4-cycle read, 18-cycle write
L2 Area	$1.339mm^2$	$1.362mm^2$
L2 Dynamic Energy	$0.574nJ$ read, $0.643nJ$ write	$0.550nJ$ read, $3.243nJ$ write
L2 Leakage Power	$0.185W$	$0.013W$
Network	4×4 Mesh, packet switching, 3 cycles per hop	
Main Memory	4GB, 150-cycle latency	

8.5.1.1 Performance and Power Evaluation Figure 82 reports the ratio of write operations served by SRAM versus STT-RAM for MSW and SPD. In MSW, two successive writes on the same data block result in a migration and the subsequent writes on that block will be served by SRAM. This allows the SRAM, which comprises just over 3% of the cache capacity to serve 39.6% of the writes. However, MSW does not perform an accurate prediction for write reuse when applications exhibit certain interleaved read/write access patterns. In contrast, SPD dispatches more effectively and results in an average of 88.5% of all the write requests being handled in SRAM.

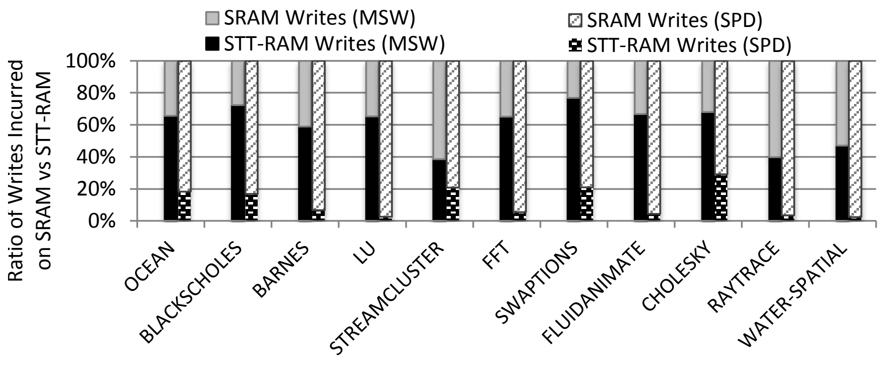


Figure 82: Ratio of writes on SRAM vs STT-RAM for MSW and SPD

Another important metric in the evaluation is the number of writes on SRAM per dispatch operation. The larger the number, the lower the dispatching overhead relative to the gain. Figure 83 presents the number of SRAM writes per dispatch for MSW and SPD. SPD has a much larger number of writes on SRAM per dispatch/migration. On average, 32 writes are served by SRAM per one migration operation in SPD while there are only 8 writes occur in SRAM per one migration in MSW.

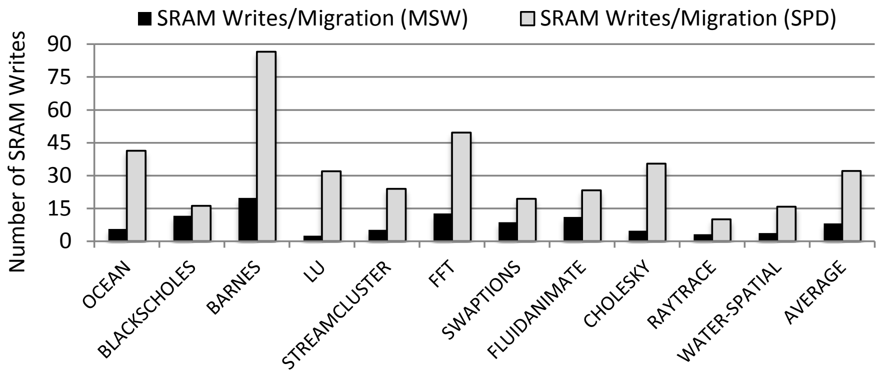


Figure 83: Number of SRAM writes per dispatch (migration)

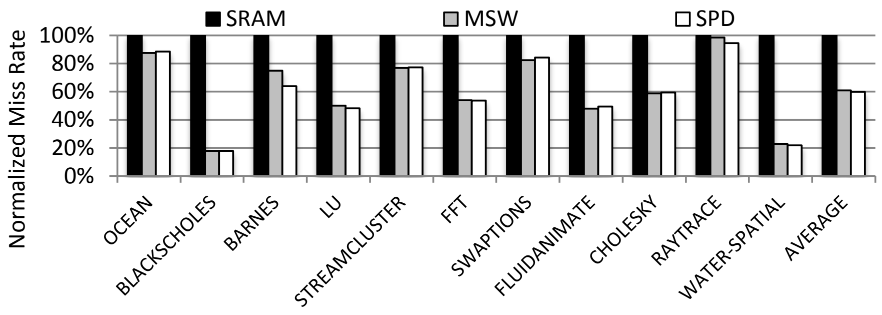


Figure 84: Normalized off-chip miss rate

The off-chip miss rate shown in Figure 84 demonstrates the advantage of reduced off-chip miss rate achieved by employing STT-RAM for on-chip storage. With 4 times more capacity than SRAM-only L2 caches, MSW and SPD reduce the off-chip misses by 38.9% and 40.0%, respectively. The reduction in expensive off-chip misses results in an average of 5% faster memory accesses for SPD, as shown in Figure 85. MSW exhibits a negligible improvement in memory access delay despite the reduction in off-chip misses as this benefit is offset by the large number of writes on STT-RAM (see Figure 82).

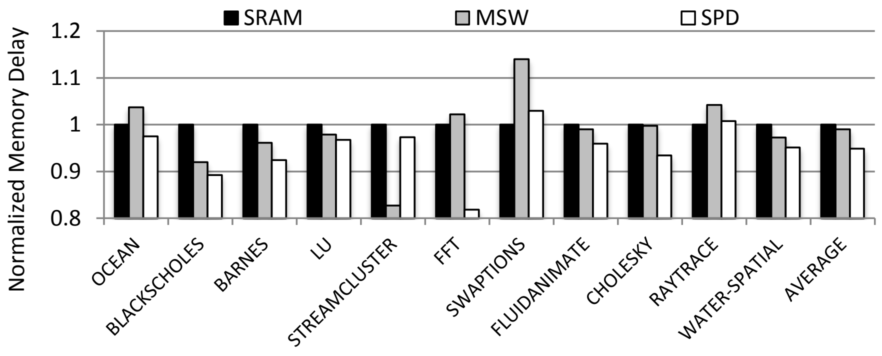


Figure 85: Normalized memory access delay

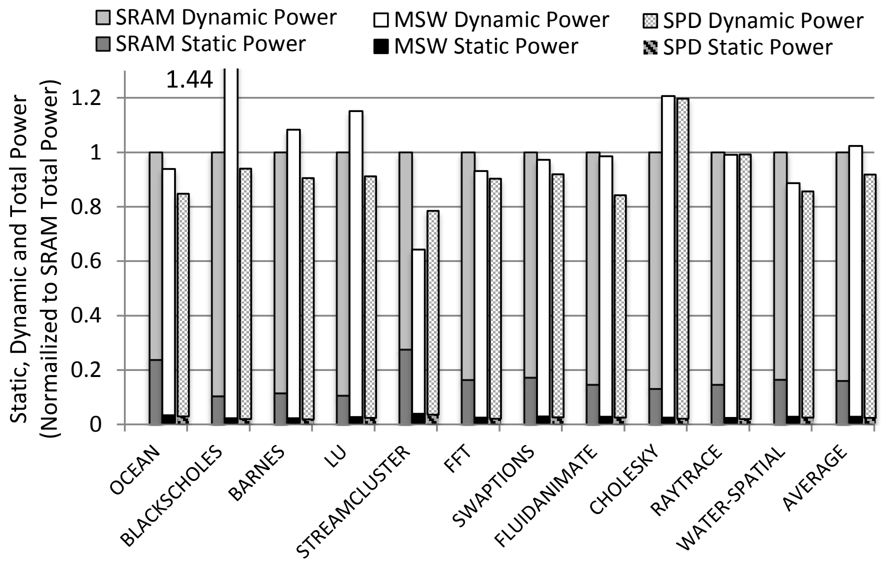


Figure 86: Normalized power consumption

Figure 86 shows the static and dynamic power breakdown normalized to the total power consumption of the SRAM cache. Due to the leakage problem inherited by CMOS devices, the SRAM-only cache consumes a non-negligible amount of static power while MSW and SPD dramatically reduce it. However, MSW consumes an average of 18.4% more dynamic power than the SRAM-only design due to the high energy overhead incurred by writes on STT-RAM. In contrast, the dynamic power consumed by SPD is close to that consumed by the SRAM-only cache since most of the writes in SPD have been dispatched efficiently onto its SRAM fraction. MSW results in an 0.3% overall higher power consumption than SRAM-only in spite of its 86.6% static power reduction. SPD has a similar static power saving of 86.8% and a total power saving of 9.8%.

8.5.2 C1C Evaluation

To evaluate the effectiveness of the configurable cache C1C Simics is used to simulate a 16-core CMP with the cache architecture as described in Table 11. The C1C scheme is compared with an SRAM-only design and the leading STT-RAM technique with reduced retention time (26.5µs) and dynamic data refresh [96] in the L1 cache but without differential sensing. All STT-RAM designs use a 64M (4M/core) LLC while the SRAM cache uses a 16M (1M/core) LLC for a same die area comparison, similar to the architecture evaluated in [96]. The scheme FL1 is statically configured with all differential blocks (i.e., FB blocks) on top of STT-RR to optimize read performance for L1. C1C is the proposed configurable L1. Just as with STT-RR, both FL1 and C1C include standard STT-RAM at the L2 and L3 cache levels. The mode configuration is conducted using source to source passes and analyses. These can be either separate extensible instructions or fields to existing instructions (see Section 8.4.2). The experimental technique is applied to the compilation flow where a new instruction (implemented using a Simics MAGIC instruction) is used to notify consecutive read addresses to the runtime system prior to the actual write/read operations occur, similar to the mechanism used by the existing compiler-instrumented prefetching instructions. The details of the input workloads are shown in Table 9.

8.5.2.1 Effectiveness of the Threshold Analysis As described in Section 8.2, the compiler analysis determines CRBs by examining use distance threshold T between reads and writes. The determination of a reasonable distance requires some consideration of application characteristics. As mentioned in Section 8.4.2, heavily accessed data locations tend to have periods of heavy read access and possibly periods of intermittent reads and writes. This behavior was examined in more detail for a subset of the benchmarks. Table 10 shows the results in varying T, in effect ensuring that a minimum of T consecutive reads must be present to form a CRB and promote the line to FB. First, it is noted that even for $T = 2$, the classification mechanism is generally effective and

Table 9: Benchmarks

Benchmark Suits	Benchmark	Description	Input
SPLASH-2	barnes	Simulation	1048576 particles
	cholesky	Matrix Calculation	tk29.O
	fft	FFT Algorithm	$2^2$6 integers
	lu	Matrix Calculate	2048x2048 matrix
	ocean	Simulation	1026x1026 matrix
	raytrace	Rendering	*teapot* input
	radix	Integer Sort	104857600 radix
	water	Simulation	3375 molecules
PARSEC	blackscholes	Financial Analysis	200000 options
	fluidanimate	Animation	in-35K.fluid
	streamcluster	Data Mining	1024 data points
	swaptions	Financial Analysis	256 swaptions

identifies a dominant percentage of the application data reads while only incurring a fairly small number of complementary writes due to mode switches or FB writes. A larger T value guards more consecutive reads for an FB promotion yielding a configuration with fewer optimized reads but also lower write overhead. By comparing the achieved read optimization and the write cost for four benchmarks, ocean, swaptions, blackscholes, and radix, across different thresholds, a value of $T = 4$ was selected for the final evaluation (Table 11) and this threshold works well for the SPLASH and PARSEC benchmarks.

Table 10: Overheads for ocean with different T values

T	2	3	4	5	6
Optimized Reads	5.12e+7	5.10e+7	5.08e+7	5.07e+7	5.06e+7
Standard Reads	782107	969789	1136674	1262467	1388100
Optimized Reads (%)	98.50%	98.14%	97.81%	97.57%	97.33%
Complement Writes	187682	166885	125793	125633	125395
Original Writes	1866391	1866391	1866391	1866391	1866391
Write Overhead (%)	10.06%	8.94%	6.74%	6.73%	6.72%

Given this threshold, Figure 87 reports the percentage of consecutive read regions that can be identified by the compiler analyses (CTR and CSR) compared with all of the CRBs present in a runtime trace. For various applications from 70% to over 95% and an average of 85% of consecutive reads can be detected by the compiler technique. This guarantees a high percentage of application data reads to be serviced in FB blocks. This is verified in Figure 88, which shows

Table 11: C1C architectural parameters (The read/write latency for LLC shown in this table is the raw access time excluding the network traversal latency)

Basics	16 cores, 2 issue width, 3.5GHz CPUs			64-bit Solaris 10 OS		4×4 mesh network with 3-cycle latency per hop, 4GB main memory, 150-cycle latency	
Caches	Private L1 Cache (MESI)			Private L2 Cache		Shared L3 Cache	
	32K 4-way 64B blk			256K 8-way 64B blk		16M 16-way 64B blk	64M 16-way 64B blk
	SRAM	STT S	STT F	SRAM	STT	SRAM	STT
Size (mm^2/core)	0.048	0.031		0.233	0.085	0.96	1.006
Read Latency(cycles)	2	3	2	4	4	5	6
Write Latency (cycles)	3	5	5	3	26	4	27
Read Energy (nJ)	0.029	0.014	0.014	0.032	0.022	0.054	0.046
Write Energy (nJ)	0.031	0.094	0.188	0.036	0.117	0.06	0.26
Leakage Power (mW)	149	63	63	664	138	1249	471

the number of reads conducted in standard mode (S Reads) and differential mode (F Reads) by employing FB mode switching with a threshold of four consecutive reads. In the C1C L1 cache, all the benchmarks have more than 80% of their data reads optimized in FBs. Many benchmarks have over 90% optimized reads leading to 91% optimized reads on average. Since the L1 cache absorbs most of the reads issued by an application, the accelerated reads in C1C are likely to bring a significant performance gain for the entire cache hierarchy.

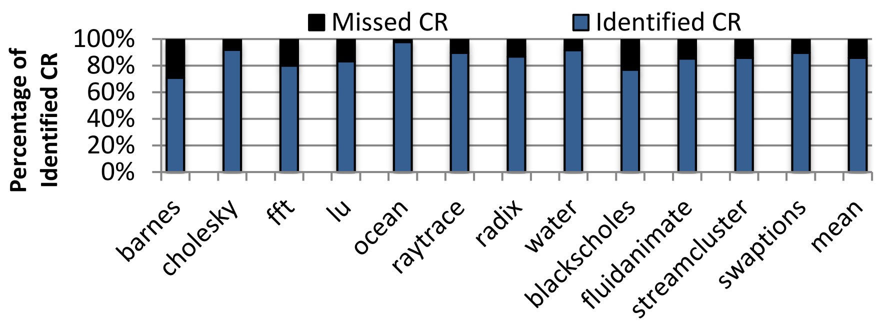


Figure 87: Percentage of identified consecutive read reuse

Due to the effectiveness of the mode configuration control, C1C accelerates a high percentage of application reads while only a tiny fraction of the writes occur in FBs, which incur additional write energy for complementary writes. Figure 89 shows the percentages of complementary writes (i.e., FB writes) are low for all the tested applications. For many applications such as RAYTRACE and FLUIDANIMATE the write overheads are almost negligible. On average, only 7% of write operations that need complementary writing. This drastically reduces the L1 dynamic power overhead compared with an all FB cache design (FL1) since STT-RAM's write power is typically high and the dominating factor of the entire dynamic cache power. The remaining results presented assume use of a mode switch threshold of $T = 4$.

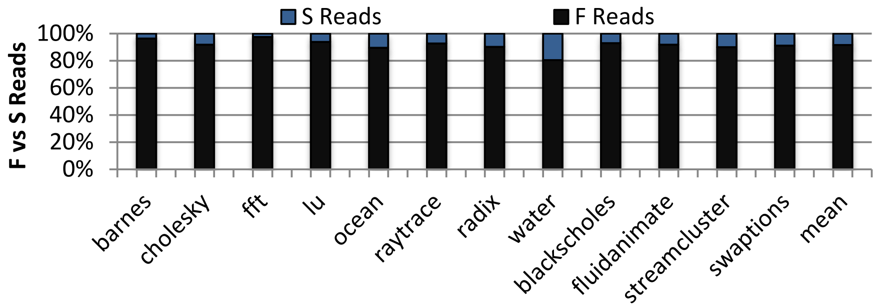


Figure 88: Reads in different modes (optimized reads)

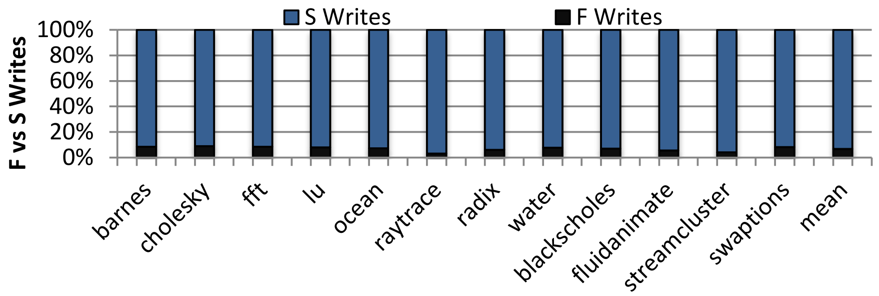


Figure 89: Writes in different modes (write overhead)

8.5.2.2 Performance and Power Evaluation For performance and power evaluation the reduced retention STT-RAM is applied to the L1 cache for all STT-RAM cache designs. The basic STT-RAM design with a retention released L1 cache is denoted as STT-RR. FL1 is the scheme that applies differential mode (FB) to all the L1 blocks. C1C is the compiler guided configurable cache that dynamically applies FB mode.

The performance comparison, presented as instructions per cycle (IPC) normalized to SRAM, is reported in Figure 90. STT-RR performs poorly compared to SRAM in spite of the capacity advantage due to the high read latency at L1. For all the benchmarks except OCEAN, which is extremely capacity sensitive, STT-RR performs worse than SRAM and leads to an average of 4% performance degradation. In contrast, by servicing all L1 reads with the read optimization technique FL1 provides over 6% improvement over SRAM and 10% improvement over STT-RR at the expense of double amount of write power at L1 for complementary writes. The performance of C1C is within 1.5% of FL1 and 5% higher than SRAM and nearly 9% higher than STT-RR.

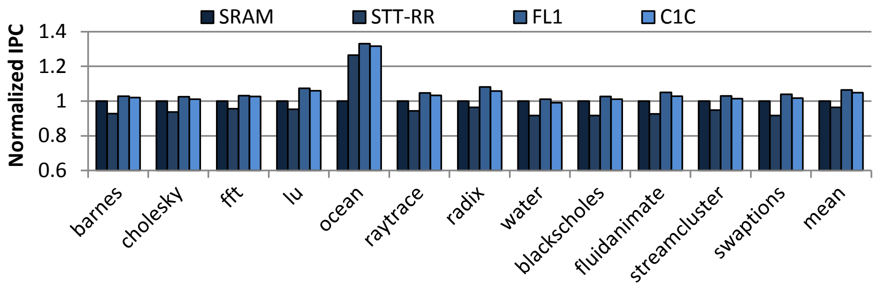


Figure 90: Performance (IPC) comparison (norm. to SRAM)

Figure 91 provides an energy comparison of the relevant schemes normalized to SRAM. STT-RR provides a 27% energy reduction over SRAM on average and as previously demonstrated the refresh energy is negligible [96]. As expected, FL1 requires much higher dynamic energy (shown separately as read and write energy in Figure 91) than SRAM and STT-RR due to the fact that all its L1 writes are conducted in FBs, incurring a high dynamic write energy overhead. For benchmarks exhibiting heavily writes such as BARNES, RADIX and BLACKSCHOLES, the dynamic power increases are unacceptable. In contrast, C1C drastically reduces the dynamic energy compared to FL1 and only requires slightly higher energy than STT-RR since there are still a small portion of writes being serviced in FBs (Figure 89) for the S to F mode switches. Compared to FL1, C1C brings similar performance with 26% less total cache energy largely due to the write energy reduction, as can be observed from Figure 91. Because of the leakage and read energy reduction, C1C also brings a 24% total energy saving over SRAM caches.

The overall benefit of C1C is demonstrated in Figure 92, which presents the IPC/watt of various schemes normalized to SRAM. STT-RR suffers from degraded performance but drastically reduced power consumption and thus brings a total of 38% IPC/watt benefit. FL1 enhances performance but increases the dynamic power leading to only 18% IPC/watt gain over the baseline. C1C brings both performance and energy benefit by dynamically configuring the modes based on application needs results in the maximum average IPC/watt improvement of 48% compared to the baseline.

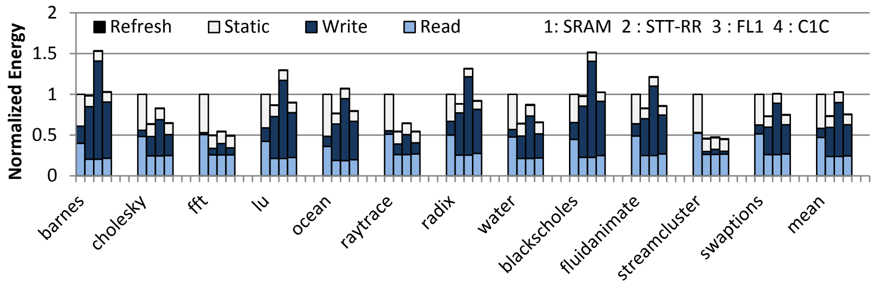


Figure 91: Energy consumption (norm. to SRAM)

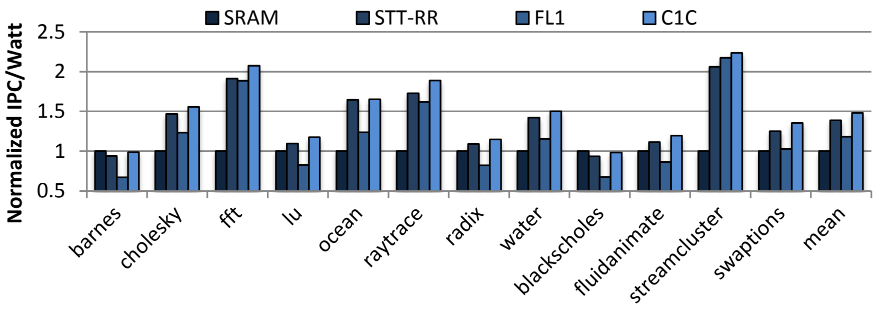


Figure 92: Performance per watt comparison (norm. to SRAM)

9.0 CONCLUSION AND FUTURE WORK

Modern CMP architecture has been evolving into an organization equipped with powerful and configurable components that are operated in a distributed fashion. This trend poses unprecedented challenges for resource utilization, which becomes a key factor for delivering high performance, low power and efficient computing platforms. As the hardware continue to scale and applications become more diverse, it is inefficient to have an one-size-fits-all approach. This dissertation work is done in the belief that future architecture will be more configurable/customizable and software defined or managed computing platforms leveraging application characteristics will become more popular. The proposed mechanisms and optimizations use application data access characteristics primarily extracted from the compiler to appropriately utilize and schedule resources. Depending on specific application and architecture characteristics, different compiler techniques can be adopted to achieve desirable tradeoffs among compilation complexity, generality and precision of analysis.

The compiler-assisted data classification presented in Chapter 3 represents a light weight but generic compiler analysis that can be used to extract classification information for a variety of parallel applications. Detecting private data is reduced to pointer and scope analyses, while the complexity of detecting private-dominant data is handled speculatively by the practically private concept. Once “TI variables” are identified, classifying data in the proposed approach can be achieved largely by conventional data flow analyses. Compared to the baseline architecture that is oblivious to application-level information, a modest performance gain (10%) is achieved by the coherent cache architecture that leverages the data classification information. On the other end of the design spectrum is the compiler-assisted partitioning and communication pattern detection presented in Chapter 6. Detecting the data partitioning requires intensive compiler analyses. In particular, the array access region analysis is NP-complete and thus, the complication time grows exponentially with the input program sizes. Additionally, discovering the data ownership requires access weight information, which can be affected by input working sets and branch conditions. Although sophisticated compiler techniques such as branch elimination and symbolic analyses can be used to simplify the ownership detection, there are cases where the proposed technique is not efficient. Therefore, compiler-assisted data partitioning and communication pattern techniques are deep program analyses suitable for applications with compile-time deterministic access patterns. For these applications, the proposed optimization can lead to significant performance gains.

Other than the aforementioned factors, data analysis granularity should also be taken into consideration when choosing an appropriate compiler approach. Typically, the required granularity depends on the target architecture optimization. A larger granularity potentially introduces inaccuracies, while a smaller granularity incurs higher storage overheads. For example, the data classification and partitioning information described in this dissertation are derived at source code data block (allocated by a memory allocator such as *malloc()*) granularity and passed to the runtime system at page or sub-page granularity. The data reuse behavior (Chapter 8) is detected and used by the reconfigurable STT-RAM cache at address location granularity. For cache-level optimizations, a runtime page-level approach such as the one used in R-NUCA results in insufficient information, and, consequently, inefficient operations. In contrast, optimizing TLBs requires only page-level data classification. In such a scenario the finer-grained compiler analysis is not necessary. Chapter 5 demonstrates that a simple page-level data classification mechanism used in R-NUCA can achieve comparable outcome in optimizing TLBs while avoiding the additional memory allocation and information passing overheads. Therefore, a conclusion can be drawn that leveraging application information at an appropriate granularity for each specific architecture optimization is crucial in reducing complexity, mitigating runtime overheads and improving system efficiency.

The application features considered in this dissertation include the data classification, communication pattern and data reuse behavior. In practice, many other useful application characteristics can be leveraged. These characteristics can be analyzed and utilized in different granularities depending on specific architecture optimization goals. In general, compared to a hardware-based methodology the proposed software-oriented approach is more flexible, less expensive, and can provide global application information to assist the architecture in resource configuration, scheduling, data distribution, etc.

The dissertation work has several limitations that may confine the application of the proposed solutions to certain scenarios. First, some compiler techniques such the region analyses adopted in the presented optimizations are NP-complete problems. If these analyses are enabled, the complication time can increase drastically for large programs. Also, the proposed analyses focus on analyzable data arrays and heap objects, which are only part of the entire data set used by an application. Although applications with large data inputs tend to have significant portions of array and heap objects, for applications that have small heap area or intensive stack operations the analyses introduced in this dissertation may have limited impact. Other limitations include the inability of handling pointer arithmetic and non-affine array accesses.

One component of future work in this direction is to improve the accuracy of the compiler analysis for the practically private data classification. The experiments in Chapter 3 show that a predominant percentage of practically private data is actually private. The current data classification analysis is conservative in detecting private data. Potential improvement can be achieved by developing a data classification algorithm with higher precision to segregate more actual private data from the practically private. Segregating private data from practically private data exposes more opportunities in coherence and data lookup optimizations. This has been partially addressed in Chapter 6, although the analysis is not used to further classify practically private data as private.

Another future direction worth considering is partitioning applications into segments and extracting the per-segment application information to help the runtime configuration. For example, the existing data partitioning and communication pattern based optimizations summarize the ownership and communication behavior for the entire span of the program. This approach is limited for applications with dynamically changing behaviors. By partitioning programs into phases demarcated by procedures or loop nests, temporally changing information can be retrieved to more accurately reflect the program behaviors during a certain time window.

With respect to the compiler-assisted data reuse analysis for the configurable STT-RAM cache, potential future work can be done to analyze more program data access patterns relevant to the asymmetric read/write access latency. Specifically, potential data access information that can benefit the designing of STT-RAM, and more generally, non-volatile memories, include write intensive access patterns, variable liveness information, etc.

BIBLIOGRAPHY

[1] A. Abousamra, R. Melhem, and A. K. Jones, “Winning with pinning in NoC,” in *Proc. of Hot Interconnects (HOTI)*, 2009.

[2] A. K. Abousamra, R. G. Melhem, and A. K. Jones, “Noc-aware cache design for chip multiprocessors,” in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT 10. New York, NY, USA: ACM, 2010, pp. 565–566.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2nd ed. Addison Wesley, 2006.

[4] L. O. Andersen, “Program analysis and specialization for the c programming language,” Ph.D. dissertation, DIKU, University of Copenhagen, 1994.

[5] J. M. Arnold, D. A. Buell, and E. G. Davis, “Splash 2,” in *SPAA ’92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1992, pp. 316–322.

[6] J.-L. Baer and W.-H. Wang, “On the inclusion properties for multi-level cache hierarchies,” in *Proceedings of the 15th Annual International Symposium on Computer architecture*, ser. ISCA ’88. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 73–80.

[7] D. Bailey, T. Harris, W. Sahpir, and R. van der Wijingaart, “The NAS parallel benchmarks 2.0,” Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Tech. Rep. NAS-95-020, December 1995.

[8] J. D. Balfour and W. J. Dally, “Design tradeoffs for tiled cmp on-chip networks,” in *ICS*, 2006, pp. 187–198.

[9] K. J. Barker, A. Benner, R. Hoare, A. Hoisie, A. K. Jones, D. J. Kerbyson, D. Li, R. Melhem, R. Rajamony, E. Schenfeld, S. Shao, C. Stunkel, and P. A. Walker, “On the feasibility of optical circuit switching for high performance computing systems,” in *Proc. of SC*, 2005.

[10] T. W. Barr, A. L. Cox, and S. Rixner, “Translation caching: skip, don’t walk (the page table),” in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA ’10. New York, NY, USA: ACM, 2010, pp. 48–59.

[11] R. Barua, D. Kranz, and A. Agarwal, "Communication-minimal partitioning of parallel loops and data arrays for cache-coherent distributed-memory multiprocessors," in *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, 1996.

[12] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *17th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2011.

[13] A. Bhattacharjee and M. Martonosi, "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 29–40. [Online]. Available: http://portal.acm.org/citation.cfm?id=1636712.1637745

[14] ——, "Inter-core cooperative tlb for chip multiprocessors," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10. New York, NY, USA: ACM, 2010, pp. 359–370. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736060

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," Princeton University, Tech. Rep. TR-811-08, January 2008.

[16] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill, "Translation lookaside buffer consistency: a software approach," *SIGARCH Comput. Archit. News*, vol. 17, pp. 113–122, April 1989. [Online]. Available: http://doi.acm.org/10.1145/68182.68193

[17] S. M. Blackburn, R. Garner, and et al., "The dacapo benchmarks: java benchmarking development and analysis," *SIGPLAN Not.*, vol. 41, pp. 169–190, October 2006. [Online]. Available: http://doi.acm.org/10.1145/1167515.1167488

[18] S. Bourduas and Z. Zilic, "A hybrid ring/mesh interconnect for network-on-chip using hierarchical rings for global routing," in *NOCS*, 2007, pp. 195–204.

[19] J. A. Brown, R. Kumar, and D. M. Tullsen, "Proximity-aware directory-based coherence for multi-core processor architectures," in *SPAA*, 2007, pp. 126–134.

[20] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," in *The 33rd International Symposium on Computer Architecture*, 2006.

[21] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, Dec. 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2010.5650274

[22] Y. Chen, H. Li, X. Wang, W. Zhu, W. Xu, and T. Zhang, "Combined magnetic- and circuit-level enhancements for the nondestructive self-reference scheme of STT-RAM," in *Proc. of ISLPED*, 2010.

[23] Y.-T. Chen, J. Cong, H. Huang, C. Liu, R. Prabhakar, and G. Reinman, "Static and dynamic co-optimizations for blocks mapping in hybrid caches," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, ser. ISLPED '12. New York, NY, USA: ACM, 2012, pp. 237–242. [Online]. Available: http://doi.acm.org/10.1145/2333660.2333717

[24] C.-T. Cheng, Y.-C. Tsai, and K.-H. Cheng, "A high-speed current mode sense amplifier for spin-torque transfer magnetic random access memory," in *Proc. of MWSCAS*, aug. 2010, pp. 181 –184.

[25] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," in *ISCA*, 2005, pp. 357–368.

[26] M. Chu and S. Mahlke, "Compiler-directed data partitioning for multicluster processors," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[27] M. Chu, R. Ravindrany, and S. Mahlke, "Data access partitioning for fine-grain parallelism on multicore architectures," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[28] B. Creusillet and F. Irigoin, "Exact vs. approximate array region analyses," in *9th Workshop on Language and Compilers for Parallel Computing*, Aug 1996.

[29] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Proceedings of the 38th annual international symposium on Computer architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000076

[30] D. E. Culler, A. Gupta, and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[31] R. Das, S. Eachempati, A. K. Mishra, N. Vijaykrishnan, and C. R. Das, "Design and evaluation of a hierarchical on-chip interconnect for next-generation cmps," in *HPCA*, 2009, pp. 175–186.

[32] Z. Diao, Z. Li, S. Wang, Y. Ding, A. Panchula, E. Chen, L.-C. Wang, and Y. Huai, "Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory," *Journal of Physics: Condensed Matter*, vol. 19, no. 16, p. 165209, 2007.

[33] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proc. of PLDI*. New York, NY, USA: ACM, 2003, pp. 245–257.

[34] H. Dybdahl and P. Stenstrom, “An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors,” in *Proceedings of International Symposium on High Performance Computer Architecture*, 2007.

[35] A. Faraj and X. Yuan, “Communication characteristics in the NAS parallel benchmarks,” in *Proc. of the Parallel and Distributed Computing and Systems Conf. (PDCS)*, 2002.

[36] X. Guo, E. Ipek, and T. Soyata, “Resistive computation: avoiding the power wall with low-leakage, STT-MRAM based computing,” in *Proc. of ISCA*, 2010.

[37] M. Gupta and P. Banerjee, “Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 2, pp. 179–193, 1992.

[38] ——, “Paradigm: a compiler for automatic data distribution on multicomputers,” in *ICS ’93: Proceedings of the 7th international conference on Supercomputing*. New York, NY, USA: ACM, 1993, pp. 87–96.

[39] M. R. Haghighat and C. D. Polychronopoulos, “Symbolic analysis for parallelizing compilers,” *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 477–518, 1996.

[40] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, “Maximizing multiprocessor performance with the suif compiler,” *Computer*, vol. 29, pp. 84–89, 1996.

[41] M. Hammoud, S. Cho, and R. G. Melhem, “Cache equalizer: a placement mechanism for chip multiprocessor distributed shared caches,” in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, ser. HiPEAC ’11. New York, NY, USA: ACM, 2011, pp. 177–186.

[42] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, “Reactive NUCA: near-optimal block placement and replication in distributed caches,” in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA ’09. New York, NY, USA: ACM, 2009, pp. 184–195.

[43] S.-Y. Ho and N.-W. Lin, “Static analysis of communication structures in parallel programs,” in *Proc. of the Int. Computer Symp.(ICS)*, 2002, pp. 215–221.

[44] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, and et al., “A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram,” *IEEE InternationalElectron Devices Meeting 2005 IEDM Technical Digest*, vol. 00, no. c, pp. 459–462, 2005. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1609379

[45] M. Hosomi, H. Yamagishi, T. Yamamoto, and K. B. et al., “A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram,” *IEDM Technical Digest*, vol. 2, no. 25, pp. 459–462, 2005.

[46] J. Huck and J. Hays, "Architectural support for translation table management in large address space machines," *SIGARCH Comput. Archit. News*, vol. 21, pp. 39–50, May 1993. [Online]. Available: http://doi.acm.org/10.1145/173682.165128

[47] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti, "Circuit-switched coherence," in *NOCS*, 2008, pp. 193–202.

[48] L. Jin and S. Cho, "Sos: A software oriented distributed shared cache management approach for chip multiprocessors," in *Intl Conference on Parallel Architectures and Compilation Techniques PACT*, 2009.

[49] A. K. Jones, S. Shao, Y. Zhang, and R. Melhem, "Symbolic expression analysis for compiled communication," *Parallel Processing Letters*, vol. 18, no. 4, pp. 567–587, December 2008.

[50] N. P. Jouppi, "Cache write policies and performance," in *Proceedings of the 20th annual international symposium on computer architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 191–201. [Online]. Available: http://doi.acm.org/10.1145/165123.165154

[51] Y. Ju and H. Dietz, "Reduction of cache coherence overhead by compiler data layout and loop transformation," *In Languages and Compilers for Parallel Computing*, pp. 344–358, 1992.

[52] H. Kalter, C. Stapper, J. Barth, J.E., J. DiLorenzo, C. Drake, J. Fifield, J. Kelley, G.A., S. Lewis, W. van der Hoeven, and J. Yankosky, "A 50-ns 16-mb dram with a 10-ns data rate and on-chip ecc," *Solid-State Circuits, IEEE Journal of*, vol. 25, no. 5, pp. 1118–1128, 1990.

[53] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '11. New York, NY, USA: ACM, 2011, pp. 393–404. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950411

[54] K. Kennedy and U. Kremer, "Automatic data layout for distributed-memory machines," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 4, pp. 869–916, 1998.

[55] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[56] ——, "Nonuniform cache architectures for wire-delay dominated on-chip caches," *IEEE Micro*, vol. 23, no. 6, pp. 99–107, 2003.

[57] J. Kim, J. D. Balfour, and W. J. Dally, "Flattened butterfly topology for on-chip networks," in *MICRO*, 2007, pp. 172–182.

[58] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor," *IEEE Micro*, vol. 2, no. 25, pp. 21–29, 2005.

[59] T. Krishna, A. K. 0002, P. Chiang, M. Erez, and L.-S. Peh, "Noc with near-ideal express virtual channels using global-line communication," in *Hot Interconnects*, 2008, pp. 11–20.

[60] J. P. Kulkarni, K. Kim, S. P. Park, and K. Roy, "Process variation tolerant sram array for ultra low voltage applications," in *Proceedings of the 45th annual Design Automation Conference*, ser. DAC '08. New York, NY, USA: ACM, 2008, pp. 108–113. [Online]. Available: http://doi.acm.org/10.1145/1391469.1391498

[61] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha, "Express virtual channels: towards the ideal interconnection fabric," in *ISCA*, 2007, pp. 150–161.

[62] ——, "Express virtual channels: Towards the ideal interconnection fabric," in *International Symposium on Computer Architecture (ISCA)*, June 2007.

[63] H. Li, X. Wang, Z.-L. Ong, Y. Z. W.-F. Wong, P. Wang, and Y. Cheng, "Performance, power and reliability tradeoffs of stt-ram cell subjective to architecture-level requirement," *IEEE International Magnetics Conference (InterMag)*, pp. AD–02, 2011.

[64] H. Li, X. Wang, Z.-L. Ong, W.-F. Wong, Y. Zhang, P. Wang, and Y. Chen, "Performance, power and reliability tradeoffs of stt-ram cell subjective to architecture-level requirement," *IEEE Transaction on Magnetics (TMAG)*, 2011.

[65] Q. Li, M. Zhao, C. J. Xue, and Y. He, "Compiler-assisted preferred caching for embedded systems with stt-ram based hybrid cache," *SIGPLAN Not.*, vol. 47, no. 5, pp. 109–118, June 2012. [Online]. Available: http://doi.acm.org/10.1145/2345141.2248434

[66] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones, "Compiler-assisted data distribution for chip multiprocessors," in *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2010, pp. 501–512.

[67] Y. Li, Y. Chen, and A. K. Jones, "A software approach for combating asymmetries of non-volatile memories," in *Proc. of ISLPED*, 2012.

[68] Z. Li and P. Yew, "Efficient interprocedural analysis for program parallelization and restructuring," in *SIGPLAN Symposium on Parallel Programming: Ezperience with Applications, Languages and Systems*, July 1988.

[69] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 367–376. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370869

[70] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-f. Ngai, "Data layout transformation for enhancing data locality on nuca chip multiprocessors," in *PACT '09: Proceedings of the 2009 18th*

International Conference on Parallel Architectures and Compilation Techniques. Washington, DC, USA: IEEE Computer Society, 2009, pp. 348–357.

[71] 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*, vol. 35, no. 2, pp. 50–58, February 2002.

[72] S. Nalam, V. Chandra, C. Pietrzyk, R. Aitken, and B. Calhoun, "Asymmetric 6t sram with two-phase write and split bitline differential sensing for low voltage operation," in *Quality Electronic Design (ISQED), 2010 11th International Symposium on*, 2010, pp. 139–146.

[73] Y. Paek, "Automatic parallelization for distributed memory machines based on access region analysis," Ph.D. dissertation, Univ.of Illinois at Urbana-Champaign, Dept. of Computer Science, Apr. 1997.

[74] Y. Paek, E. Z. A. Navarro, J. Hoeflinger, and D. Padua, "An advanced compiler framework for noncache-coherent multiprocessors," in *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 3, Mar. 2002, pp. 241–259.

[75] M. Qureshi, M. Franceschini, and L. Lastras-Montano, "Improving read performance of phase change memories via write cancellation and write pausing," in *Proc. of HPCA*, 2010.

[76] M. Qureshi, M. Franceschini, A. Jagmohan, and L. Lastras, "PreSET: Improving read-write performance of phase change memories by exploiting asymmetry in write times," in *Proc. of ISCA*, 2012.

[77] Ramanujam and P. Sadayappan, "Compile-time techniques for data distributionin distributed memory machines," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 2, no. 4, pp. 472–482, 1991.

[78] M. Rasquinha, D. Choudhary, S. Chatterjee, S. Mukhopadhyay, and S. Yalamanchili, "An energy efficient cache design using spin torque transfer STT RAM," in *Proc. of ISLPED*, 2010.

[79] R. Riesen, "Communication patterns," in *In Workshop on Communication Architecture for Clusters CAC'06, Rhodes Island, Greece*, April 2006.

[80] 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 International Symposium on High-Performance Computer Architecture (HPCA)*, January 2010.

[81] A. Ros, M. Cintra, M. E. Acacio, and J. M. Garcła, "Distance-aware round-robin mapping for large nuca caches," in *16th Intl Conference on High Performance Computing (HiPC)*, 2009.

[82] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*,

ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 241–252. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370853

[83] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta, "The impact of architectural trends on operating system performance," *SIGOPS Oper. Syst. Rev.*, vol. 29, pp. 285–298, December 1995. [Online]. Available: http://doi.acm.org/10.1145/224057.224078

[84] K. Sakuma, P. S. Andry, C. K. Tsang, S. L. Wright, B. Dang, C. S. Patel, B. C. Webb, J. Maria, E. J. Sprogis, S. K. Kang, R. J. Polastre, R. R. Horton, and J. U. Knickerbocker, "3d chip-stacking technology with through-silicon vias and low-volume lead-free interconnections," *IBM J. Res. Dev.*, vol. 52, pp. 611–622, November 2008.

[85] R. Scheuerlein, W. Gallagher, S. Parkin, A. Lee, S. Ray, R. Robertazzi, and W. Reohr, "A 10 ns read and write non-volatile memory array using a magnetic tunnel junction and fet switch in each cell," in *Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International*, 2000, pp. 128–129.

[86] R. M. S. Shao and A. K. Jones, "A compiler-based communication analysis approach for multiprocessor systems," in *In 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), Rhodes Island, Greece.*, April 2006.

[87] S. Shao, A. K. Jones, and R. Melhem, "Compiler techniques for efficient communications in circuit switched networks for multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 14, no. 1, pp. 331–345, 2008.

[88] ——, "Compiler techniques for efficient communications in circuit switched networks for multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 331–345, March 2009.

[89] D. Shires, L. Pollock, and S. Sprenkle, "Program flow graph construction for static analysis of mpi programs," in *Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications(PDPTA)*, June 1999.

[90] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," hp, Tech. Rep., August 2001.

[91] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient stt-ram caches," *Proc. of HPCA*, 2011.

[92] C. Smullen IV, V. Mohan, A. Nigam, S. Gurumurthi, and M. Stan, "Relaxing Non-Volatility for Fast and Energy-Efficient STT-RAM Caches," *Proc. of 2011 HPCA*, 2011.

[93] S. Srikantaiah and M. Kandemir, "Synergistic tlbs for high performance address translation in chip multiprocessors," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 313–324.

[94] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3d stacked mram l2 cache for cmps," in *Proceedings of the High Performance Computer Architecture*, 2009, pp. 239–249.

[95] ——, "A novel architecture of the 3d stacked mram l2 cache for cmps," in *Proc. of HPCA*, feb. 2009, pp. 239 –249.

[96] Z. Sun, X. Bi, H. Li, W.-F. Wong, Z.-L. Ong, X. Zhu, and W. Wu, "Multi Retention Level STT-RAM Cache Designs with a Dynamic Refresh Scheme," in *Proc. of MICRO*, 2011.

[97] R. E. Tarjan, "Fast algorithms for solving path problems," *J. ACM*, vol. 28, pp. 594–614, July 1981.

[98] S. W. K. Tjiang and J. L. Hennessy, "Sharlit—a tool for building optimizers," in *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1992, pp. 82–93.

[99] R. Triolet, F. Irigoin, and P. Feautrier, "Direct parallelization of call statements," in *ACM SIG-PLAN ' 86 Symposium on Compiler Construction, Palo Alto, CA*, July 1986, pp. 176–185.

[100] P. Tu and D. Padua, "Gated ssa-based demand-driven symbolic analysis for parallelizing compilers," in *Proc. of SC*, 1995, pp. 414–423.

[101] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest, and R. Brown, "Design tradeoffs for software-managed tlbs," *ACM Trans. Comput. Syst.*, vol. 12, pp. 175–205, August 1994. [Online]. Available: http://doi.acm.org/10.1145/185514.185515

[102] G. Venkatasubramanian, R. J. Figueiredo, and R. Illikkal, "On the performance of tagged translation lookaside buffers: A simulation-driven analysis," in *19th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS 2011, 2011.

[103] J. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," *Journal of Parallel and Distributed Computing*, vol. 63, no. 9, pp. 853–865, September 2003.

[104] 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 *Parallel Architectures and Compilation Techniques (PACT)*, October 2011.

[105] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarsinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "Suif: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, December 1994.

[106] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarsinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. s. Lam, and J. L. Hennessy, "Suif: An infrastructure for research on parallelizing and optimizing compilers," in *SIGPLAN Notices*, 1994.

[107] M. E. Wolf, "Improving locality and parallelism in nested loops," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1992, uMI Order No. GAX93-02340.

[108] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 34–45.

[109] Y. Xie, G. H. Loh, B. Black, and K. Bernstein, "Design space exploration for 3d architectures," *J. Emerg. Technol. Comput. Syst.*, vol. 2, pp. 65–103, April 2006.

[110] H.-C. Yu, K.-C. Lin, K.-F. Lin, C.-Y. Huang, Y.-D. Chih, T.-C. Ong, L. C. Tran, and F.-L. Hsueh, "New circuit design architecture for a 300-mhz 40nm 1mb embedded stt-mram with great immunity to pvt variation," in *International Proceedings of Computer Science and Information Technology (IPCSIT)*, 2012.

[111] Z.Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," in *Intl Symp. Computer Arch.*, June 2005.

[112] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 423–434. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669166

[113] ——, "A tagless coherence directory," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 423–434.

[114] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *32nd Annual International Symposium on Computer Architecture*, 2005.

[115] Y. Zhang, W. Wen, and Y. Chen, "The prospect of stt-ram scaling from readability perspective," *Magnetics, IEEE Transactions on*, vol. 48, no. 11, pp. 3035–3038, 2012.

[116] H. Zhao, A. Shriraman, and S. Dwarkadas, "Space: sharing pattern-based directory coherence for multicore scalability," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854294

[117] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for STT-RAM using early write termination," in *Proc of ICCAD*, 2009.