ABSTRACT

Title of dissertation: REUSE DISTANCE ANALYSIS FOR LARGE-SCALE CHIP MULTIPROCESSORS

Meng-Ju Wu, Doctor of Philosophy, 2012

Dissertation directed by: Professor Donald Yeung
Department of Electrical and Computer Engineering

Multicore Reuse Distance (RD) analysis is a powerful tool that can potentially provide a parallel program's detailed memory behavior. Concurrent Reuse Distance (CRD) and Private-stack Reuse Distance (PRD) measure RD across thread-interleaved memory reference streams, addressing shared and private caches. Sensitivity to memory interleaving makes CRD and PRD profiles architecture dependent, preventing them from analyzing different processor configurations. However such instability is minimal when all threads exhibit similar data-locality patterns. For loop-based parallel programs, interleaving threads are symmetric. CRD and PRD profiles are stable across cache size scaling, and exhibit predictable *coherent movement* across core count scaling. Hence, multicore RD analysis can provide accurate analysis for different processor configurations. Due to the prevalence of parallel loops, RD analysis will be valuable to multicore designers.

This dissertation uses RD analysis to analyze multicore cache performance for loop-based parallel programs. First, we study the impacts of core count scaling and problem size scaling on CRD and PRD profiles. Two application parameters with

architectural implications are identified: C_{core} and C_{share}. Core count scaling only impacts cache performance significantly below C_{core} in shared caches, and C_{share} is the capacity at which shared caches begin to outperform private caches in terms of data locality. Then, we develop techniques, in particular employing reference groups, to predict the coherent movement of CRD and PRD profiles due to scaling, and achieve accuracy of 80%–96%. After comparing our prediction techniques against profile sampling, we find that the prediction achieves higher speedup and accuracy, especially when the design space is large. Moreover, we evaluate the accuracy of using CRD and PRD profile predictions to estimate multicore cache performance, especially MPKI. When combined with the existing problem scaling prediction, our techniques can predict shared LLC (private L2 cache) MPKI to within 12% (14%) of simulation across 1,728 (1,440) configurations using only 36 measured CRD (PRD) profiles. Lastly, we propose a new framework based on RD analysis to optimize multicore cache hierarchies. Our study not only reveals several new insights, but it also demonstrates that RD analysis can help computer architects improve multicore designs.

Reuse Distance Analysis for Large-Scale Chip Multiprocessors

by

Meng-Ju Wu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2012

Advisory Committee:
Professor Donald Yeung, Chair/Advisor
Professor Bruce Jacob
Professor Manoj Franklin
Professor Rajeev Barua
Professor Chau-Wen Tseng

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible. First I would like to thank my advisor, Professor Donald Yeung for his unwavering support as I worked at this challenging and fruitful topic. The thinking process which I learned from him is the best treasure of the entire research.

I would also like to thank Professor Bruce Jacob, Professor Manoj Franklin, Professor Rajeev Barua, and Professor Chau-Wen Tseng for agreeing to serve on my thesis committee and for sparing their invaluable time reviewing this manuscript.

My colleagues at the University of Maryland have enriched my graduate life in many ways and deserve a special mention. Hameed, Xuanhua, Wanli, Xu, Inseok, Minshu, and Cheng-Han gave me wonderful help and discussion. I would also like to thank my friends for enriching my graduate life in many ways.

I owe my deepest thanks to my family, my mother and father, my sister and brother-in-law, and my little nieces. Although they are in Taiwan and far away from me, their love, encouragement, and guidance have supported me throughout my whole life. Words cannot express the gratitude I owe them. And finally, to my dearest wife, Yung-Ching, whom I love very much. More than anything, she has continuously motivated me and given me confidence through this journey. Without her love and support, I could not complete my dissertation, and life will be tasteless.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In recent years, chip multiprocessors (CMPs) dominate design trends as chip manufacturers strive to achieve greater performance and power efficiency. CMPs with one hundred cores are already in the market, and CMPs with more than one hundred cores and more than one hundred MBs of on-chip cache will be available in the near future. On multicore processors, parallel programs can use multiple cores in parallel to solve problems more quickly. One key factor determining a multicore processor's performance and power consumption is how effectively programs can utilize the on-chip cache hierarchy.

Memory performance depends on the physical characteristics of the cache system and the parallel application's intra-thread locality and inter-thread interactions in the cache hierarchy. For example, data sharing may reduce the aggregate working set size in shared caches, decreasing the cache capacity pressure. However, shared caches have longer average access latency. In contrast, data sharing may cause replication and communication in private caches, reducing the effective cache capacity and inducing coherence misses. However, private caches keep data locally and have shorter average access latency.

To understand these complex effects, simulation is the de facto method for

studying multicore cache hierarchies [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. These studies simulate processors with varying *core count* and *cache capacity* to quantify how different designs impact memory performance. However, the number of configurations in terms of core count, cache hierarchy, and problem size is very large. Worse yet, detailed simulations will become more complex and time consuming as processors and problem sizes scale to the large-scale chip multiprocessor (LCMP) level. Hence, using detailed simulations to study this large design space becomes difficult due to the multi-dimensional nature of the design space.

To study future CMPs, computer architects need new tools to gain deeper insights into multicore memory performance. Reuse Distance (RD) analysis [12] is a good potential candidate to tackle this multi-dimensional design space problem. Recently, researchers have developed multicore RD analysis to analyze multicore cache performance for shared caches and private caches. To address interference and data sharing effects between threads in shared caches, *Concurrent Reuse Distance* (CRD) [13, 14, 15, 16, 17] uses a global stack to measure RD across thread-interleaved memory reference streams. On the other hand, to address data replication and communication in private caches, *Private-stack Reuse Distance* (PRD) [15, 16, 17] uses per-thread coherent stacks to measure RD separately for individual threads. For multicore processors, we can use CRD and PRD profiles together to evaluate an application's memory performance for different cache hierarchies quickly. But most importantly, it provides rich insights into how an application's inter-thread interactions impact its data locality.

A major problem with multicore RD analysis is that CRD and PRD profiles

are sensitive to how inter-thread memory references interleave. For example, the number of interleaved memory streams increases as core count scales. Hence, an application's CRD and PRD profiles are not valid at different core counts. Even at the same core count, the relative execution speed between threads may change across different cache sizes, and this may change the interleaving of memory references. As a result, the CRD and PRD profiles measured on one cache size may not be valid for another cache size. So, strictly speaking, CRD and PRD profiles are *architecture dependent*. Such profile's instability defeats the benefits of multicore RD analysis.

Previous multicore RD research has revolved around developing techniques for acquiring profiles and verifying accuracy. Researchers have investigated constructing multicore RD profiles by using trace-based analyses [13, 14] for shared caches. Unfortunately, these techniques are complex because they need to take into account all the possible ways that memory references can interleave. Moreover, these techniques usually require at-scale profiling. Hence, they are impractical for large core counts and problem sizes.

In this dissertation, we will show that the complexity of analyzing memory interleaving depends on how programs are parallelized. *Task-level parallelism* and *loop-level parallelism* are two of the major parallelization techniques. In task-level parallel programs, threads are often doing different computations, and they have different locality characteristics. When the cache size changes, the relative speed between threads may change, causing irregular memory interleaving and complex thread interference. In contrast, in a loop-based parallel program, threads from the same parallel loop are doing very similar computations. These threads have almost

identical locality characteristics. When the cache size changes, these threads all either speed up or slow down, but by the same amount. So roughly speaking, the interleaving does not change. In this case, CRD and PRD profiles are highly stable across different cache sizes and can provide accurate analysis. We also find that core count scaling makes CRD and PRD profiles shift *coherently* in a shape-preserving way. The coherent movement suggests predictability. When combined with the existing problem scaling prediction [20], we can study the entire design space from a small number of samples very quickly, and enable practical RD analysis for LCMP-scale systems.

In this work, we focus on loop-based parallel programs. Although this is one restriction of our work, loop-based parallel programs are pervasive in many domains, for example, scientific, multimedia, data-mining, and bioinformatics applications. A lot of data-parallel applications have symmetric threads. One of the most popular programming models, OpenMP, also provides a pragma to parallelize loops. In addition, loop-based parallel programs can provide large amounts of parallelism simply by increasing the problem size, so they are highly scalable. For future CMPs, loop-based parallel programs will be very important workloads. For these reasons, multicore RD analysis for loop-based parallel programs will be very valuable to multicore designers, compilers, and programmers.

1.2 Contributions

This dissertation presents a thorough investigation of multicore RD analysis. The challenges lie in developing an efficient multicore RD analysis framework to analyze the CRD and PRD profiles for different scaling dimensions (core count and problem size) and different cache hierarchies (multi-level private and shared caches). This dissertation addresses these challenges and makes the following six contributions.

(1) In-depth Analysis on CRD and PRD Profiles

We provide an in-depth analysis on inter-thread interactions in both shared and private caches, and we show how CRD and PRD profiles capture them. We isolate these different effects by creating several new locality profiles to analyze their relative contributions.

First, memory reference streams are interleaved in shared caches, and the interleaving degrades intra-thread's data locality. When data sharing happens, it can reduce the memory reference's reuse distance and improve data locality in shared caches. Because our benchmarks tend to share data across distant iterations, data sharing usually impacts CRD profiles at large RD values in our benchmarks. Depending on where data sharing happens, inter-thread shared memory references also tend to spread and distort the CRD profile. However, we find that this effect is not significant in our benchmarks.

Second, read-shared data causes replication in private caches, reducing the effective cache capacity. On the other hand, write-shared data causes invalidation

in private caches. While invalidations cause coherence misses for the reuses of victimized data blocks, they can also improve locality because the holes they leave behind can absorb stack demotions. PRD profiles can capture these effects. In CMPs, multiple private stacks contribute to increased cache capacity. To capture this effect, we compute the *scaled PRD*, or sPRD, which equals $T \times PRD$, where T is the number of threads. Because both CRD and sPRD reflect total cache capacity, we can compare the cache performance between shared and private caches across different sizes by comparing CRD and sPRD profiles directly.

Our analysis quantify these effects, and help researchers better understand how inter-thread interactions impact an application's memory behavior.

(2) The Impact of Core Count Scaling

We use RD analysis to study the impact of core count scaling on an application's memory behavior, showing how CRD and PRD profiles evolve at different core counts. For core count scaling, we find CRD profiles shift *coherently* to larger RD values in a shape-preserving way. Shifting slows down and eventually stops at a certain RD value, and we define this point as C_{core}. Core count scaling only impacts cache performance significantly below this stopping point in shared caches.

Core count scaling also causes sPRD profiles to shift to larger RD values in a shape-preserving way. However, replications and coherence misses also grow as core count scales. As a result, there is no C_{core} in sPRD profiles, and data locality degradation happens across all RD values.

(3) Architectural Implications

We also explore the architectural implications of our data sharing insights.

This dissertation defines C_{share} to be the cache capacity at which the data sharing of a given application becomes noticeable. Beyond this point, shared caches show locality advantage (lower cache misses) over private caches. We also find that the degree of data sharing is not a fixed characteristic of a given application, but rather is a function of RD value. So the selection between private and shared caches also depends on cache capacities.

When considering the scaling impact, we find that C_{core} shifts to larger RD values and C_{share} shifts to smaller RD values with core count scaling. This suggests that the cache capacity at which shared caches begin to outperform private caches decreases as core count scales. But this benefit must be weighted against the higher access latency of shared caches which also grows as core count scales.

Problem size scaling increases the working set size, and CRD and sPRD profiles shift to larger RD values. We also find that both C_{core} and C_{share} shift to larger RD values. As a result, problem size scaling may reduce the benefit of using shared caches at a fixed cache capacity.

(4)Profile Prediction

The CRD and PRD profiles of loop-based parallel programs show coherent shifting with core count scaling and problem size scaling, and we develop techniques to predict the coherent movement of CRD and PRD profiles. Reference groups [20] is previously used to predict a sequential program's RD profiles across problem size scaling. We employ this technique to predict CRD and PRD profiles across core count scaling. Because data sharing also causes spreading, we propose uniformly distributing the portion of CRD profiles, which is associated with shared references.

We investigate the prediction accuracy of CRD and PRD profiles under three scaling schemes, core count scaling, problem size scaling, and core-problem scaling. To evaluate the prediction accuracy between measured and predicted profiles, we use two metrics, $\text{RD}_{Accuracy}$ and $\text{RD\_CMC}_{Accuracy}$. The former represents the normalized absolute difference, and the latter reflects the difference in cache performance. The average $\text{RD}_{Accuracy}$ and $\text{RD\_CMC}_{Accuracy}$ for CRD (PRD) profiles are between 82.4% (80.7%) and 91.5% (96.3%). We also find that the prediction accuracy decreases as the prediction horizon increases.

Lastly, we compare our prediction technique against the RD sampling technique, which can also accelerate the acquisition of profiles. The prediction technique and the sampling technique have similar average accuracy. However, the sampling technique needs to collect profiles at every configuration. In contrast, the prediction technique can predict any configuration from a small number of measurements. The benefit of prediction becomes more significant for core-problem scaling. As a result, our prediction technique can outperform the RD sampling technique.

(5) Profile Stability and Cache Performance Validation via Simulation

We quantify the CRD and PRD profiles' dependence on cache capacity, and we also validate the cache performance provided by CRD and PRD profile predictions against detailed simulations. We use the M5 simulator to model tiled CMPs and simulate our benchmarks on processors with 2–256 cores. For shared last level caches (LLCs), we simulate the cache capacity from 4MB to 128MB. For private L2 caches, we simulate the per-core L2 cache capacity from 16KB to 256KB. In total, we simulate 3,168 configurations.

Two stability metrics, $\text{RD}_{Stability}$ and $\text{RD\_CMC}_{Stability}$, are used to evaluate profile stability. The average $\text{RD}_{Stability}$ and $\text{RD\_CMC}_{Stability}$ for CRD (PRD) profiles are 97.2% (99.97%) and 99.6% (99.89%), respectively. The results confirm that CRD and PRD profiles are minimally cache-capacity dependent in our loop-based parallel programs.

Lastly, our core count prediction techniques can predict shared LLC (private L2 cache) MPKI to within 10% (13%) of simulation across 1,728 (1,440) configurations using 72 measured CRD (PRD) profiles. When combined with the existing prediction technique for problem size scaling, we can predict shared LLC (private L2 cache) MPKI to within 12% (14%) of simulation using 36 measured CRD (PRD) profiles. The results show that our prediction technique can help explore a large design space efficiently.

(6) Multicore Cache Hierarchy Optimization

Lastly, we propose a novel framework for identifying optimal multicore cache hierarchies for loop-based parallel programs by using reuse distance analysis. Our framework can analyze and quantify the performance difference for different cache hierarchies easily, providing several new insights. In this work, we focus on tiled-CMPs.

The key to optimizing multicore cache hierarchies lies in *balancing the total on-chip and off-chip memory stalls*. To achieve good performance, the capacity of the last private cache above the last level cache must exceed the region in the PRD profile where significant data locality degradation happens. Shared LLCs can outperform private LLCs when *the total off-chip memory stall saved in shared LLCs*

is larger than the total on-chip memory stall saved in private LLCs. At the optimal LLC size, the average performance (AMAT) difference between private and shared LLCs can reach as high as 15%, but it is smaller than the performance difference caused by L2/LLC capacity partition (76% in shared LLCs, and 33% in private LLCs). This suggests that the physical data locality is very important for multicore cache systems.

1.3 Roadmap

The rest of this dissertation is organized as follows. Chapter 2 provides the background for our study on multicore RD analysis, and explains the methodology used to acquire CRD and PRD profiles. Chapter 3 discusses the impact of data sharing on CRD and PRD profiles by breaking down CRD and PRD profiles into several profiles to explain how different effects change the application's data locality. We also explore the architectural implications of CRD and PRD profiles across core count scaling and problem size scaling. The coherent movement in CRD and PRD profiles due to different scaling schemes suggests the predictability of profiles. Chapter 4 develops techniques to predict CRD and PRD profiles, and it evaluates the prediction accuracy. Then, Chapter 5 validates the profile stability, and it also demonstrates our technique's ability to accelerate cache performance evaluation. To study the multicore cache system design, Chapter 6 proposes a novel framework based on multicore RD analysis for studying cache hierarchy optimization. Chapter 7 compares our prediction technique against the RD sampling technique. Finally,

Chapters 8 lists the prior work related to this research, and Chapter 9 concludes this dissertation and suggests future research directions.

Chapter 2

Background and Methodology

This chapter describes the essential concept of multicore reuse distance (RD) analysis and the methodology used to acquire profiles. Section 2.1 introduces multicore reuse distance. Section 2.2 presents our modified Intel Pin tool, which we use to profile loop-based parallel programs. Then, we introduce the 9 benchmarks and the architecture-application design space used to drive this work.

2.1 Multicore Reuse Distance

In 1970, Mattson et al. [12] introduced reuse distance (RD) to model different storage configurations on virtual memory pages in one pass. Later, researchers applied RD analysis to study uniprocessor cache performance.

Reuse distance measures the number of unique data blocks referenced between two references to the same data block in the LRU stack. When a new data block appears in the memory reference stream, this data block is pushed onto the stack. When a previously-accessed data block appears, the stack is searched. The reuse distance is the depth between the referenced data block and the top of the stack. The histogram of RD values for all references in a program is the RD profile. For an LRU cache with capacity C, the number of cache misses is the sum of all references counts with reuse distance $\geq C$ in the RD profile. For sequential programs, RD

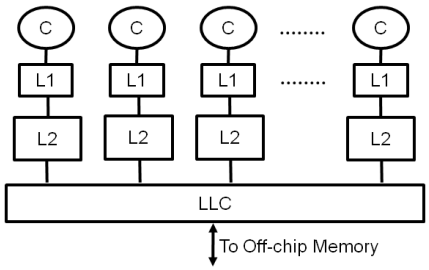


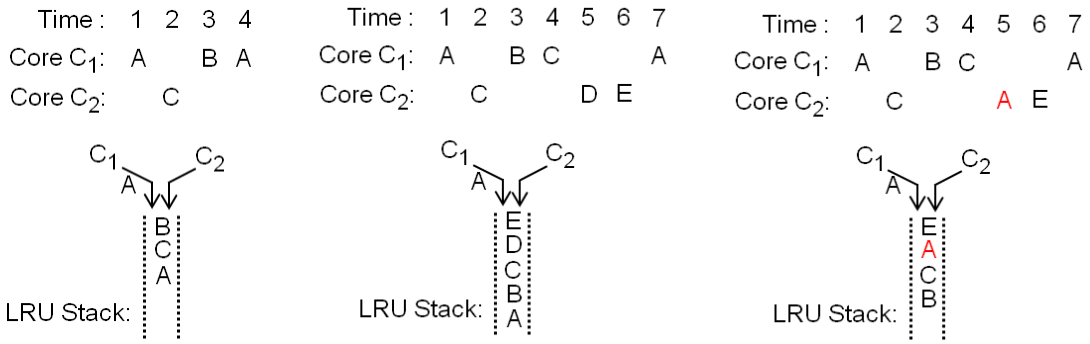
Figure 2.1: Multicore cache hierarchy.

profiles are *architecture independent*. They can be acquired on one machine, and then used to predict cache misses at different cache sizes without additional runs.

Multicore processors often contain both shared and private caches. For example, Figure 2.1 illustrates a typical multicore processor consisting of 2 levels of private cache backed by a shared last-level cache. Threads interact very differently in each type of cache, requiring separate locality profiles. For example, data sharing may reduce the aggregate working set size in shared caches, reducing cache capacity pressure. In contrast, data sharing may cause replication and communication across private caches, reducing the effective cache capacity and inducing coherence misses. To model shared caches and private caches, we use *Concurrent Reuse Distance (CRD)* and *Private-stack Reuse Distance (PRD)* profiles, respectively.

2.1.1 Concurrent Reuse Distance

RD analysis can be extended for shared caches by computing reuse distance across the interleaved memory streams from all cores on a single LRU stack–*i.e.*, the concurrent reuse distance (CRD) [13, 14, 15, 16, 17]. Data locality in shared caches is affected by several different inter-thread interactions. Figure 2.2 illustrates CRD



(a) Dilation. (b) Overlap. (c) Intercept.

Figure 2.2: Two interleaved memory reference streams, illustrating dilation, overlap, and intercept among inter-thread memory references in the shared cache.

for a sequence of interleaved memory references from two cores, showing dilation, overlap, and intercept among inter-thread memory references in the shared cache.

In Figure 2.2(a), Core 1 accesses data blocks A and B at time 1 and 3, while Core 2 accesses data block C at time 2. When Core 1 accesses A at time 4, Core 1's reuse of A has $RD = 1$, but its $CRD = 2$. In this case, CRD is larger than RD, because Core 2 brings in one unique reference, C. Hence, the interleaving causes *CRD dilation*.

In many multithreaded programs, threads share data. Data sharing can reduce dilation in two ways. First, data sharing can introduce *overlapping references*, which happens when data sharing occurs inside the reuse interval of referenced data. In Figure 2.2(b), both Core 1 and Core 2 access block C at time 2 and time 4. So there are only 4 unique references between the reuse of A, instead of 5, due to the overlap. Second, data sharing can introduce *intercepts*, which occur when data sharing happens on the reused data itself. For example, in Figure 2.2(c), Core 2 references A instead of D at time 5, which causes Core 1's reuse of A to exhibit

CRD = 1, so CRD actually becomes less than RD.

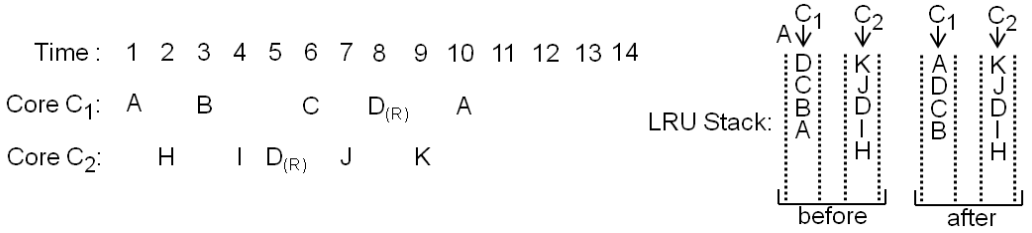
In Chapter 3, we investigate dilation, overlap, and intercept in CRD profiles. Then we study their effects as core count and problem size scale.

2.1.2 Private-stack Reuse Distance

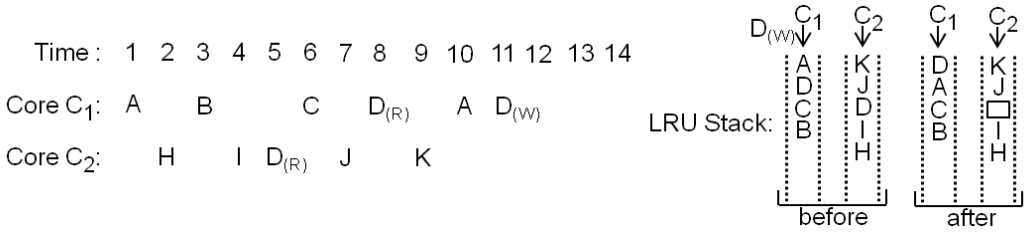
Private-stack Reuse Distance (PRD) profiles are measured by applying each thread's memory reference stream onto its own LRU stack while maintaining coherence across per-thread stacks [15, 16, 17]. To maintain data coherence in private caches, write invalidation is a common mechanism. In the absence of writes, there are not any inter-thread interactions across private stacks. For example, Figure 2.3(a) shows the PRD stacks from two cores before and after the memory access to A at time 10. Both Core 1 and Core 2 have read data block D by this time. Read-sharing causes duplication of D in the private stacks. Hence, replications reduce the effective cache capacity in private caches.

When a write happens, only one data block is kept in the private stacks, as the cache coherence protocol invalidates all other copies. In Figure 2.3(b), Core 1 writes D at time 11, and Core 1's reuse of D has PRD = RD = 1. Core 2's block D is invalidated. To prevent invalidations from promoting blocks further down the LRU stack, invalidated blocks become holes [15, 16, 17].

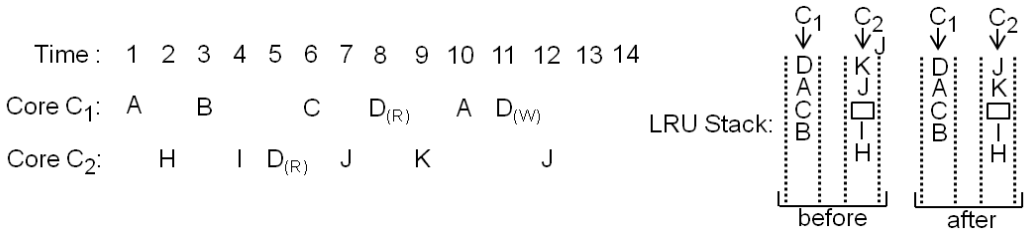
The depth of the hole is unaffected when referencing blocks above the hole. In Figure 2.3(c), Core 2 accesses block J at time 12. Block J moves to the top of the stack, and it pushes K down. The hole remains at the same position. However, when



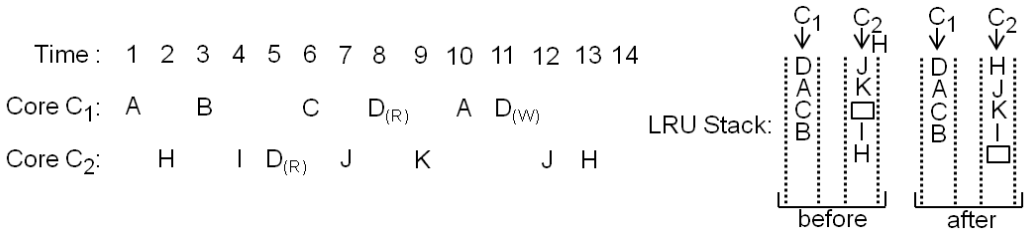
(a) Replication.



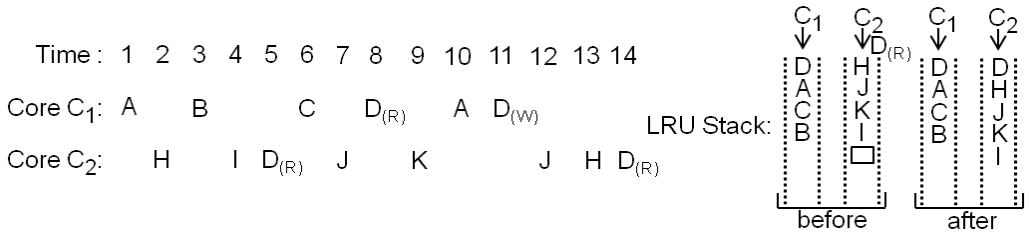
(b) Invalidation.



(c) Access to a block above a hole.



(d) Access to a block below a hole.



(e) Re-reference to an invalidated data.

Figure 2.3: Two memory reference streams, illustrating replication, invalidation, and hole in the private caches.

Core 2 accesses H, which is below the hole, H is brought to the top of the stack. Then the hole moves to the former depth of block H, as shown in Figure 2.3(d). So, blocks which are deeper than H remain at the same depths in the stack.

When a new block or an invalidated block is accessed, all the data blocks above the topmost hole are pushed down and fill the topmost hole. Figure 2.3(e) shows an example. When Core 2 re-references invalidated block D, it causes a miss in Core 2's private cache. Data block $I - H$ are pushed down and fill the hole.

Invalidations always cause the reuse of a victimized data block to be a cache miss, and these are known as coherence misses. However, invalidations may also improve data locality because the holes they leave behind eventually absorb stack demotions. For example, if Core 2 first accesses a new data block L instead of H in Figure 2.3(d), the hole will be filled, and the depth of H is still 4. Next when Core 2 accesses block H, the reuse of block H has $PRD = 4$, instead of 5. We call this effect *demotion absorption*. Hence, if victimized data blocks are not re-referenced frequently, invalidations may actually relieve capacity pressure and improve data locality.

In Chapter 3, we will investigate replication and invalidation in PRD profiles. Then we will study their effects as core count and problem size scale.

2.2 Methodology

To provide an in-depth analysis on how data sharing and interleaving impact CRD and PRD profiles for loop-based parallel programs, we develop a profiling tool

based on the Intel Pin infrastructure to acquire CRD and PRD profiles across 9 benchmarks running 4 different problem sizes on 2–256 cores. In this section, we first introduce our Pin-based tool. Then we present the benchmarks and the design space that are used in this research.

2.2.1 Pin-based Profiling Tool

Intel's Pin [21] is a dynamic binary instrumentation tool that can capture very detailed program behavior. The instrumented binary runs natively on the hardware, so it provides much higher performance and compatibility than simulators. Hence, we develop our own Pin tool to acquire CRD and PRD profiles.

One challenge in acquiring multicore RD profiles by using Pin is to ensure the accurate modeling of inter-thread interactions. We need to control the context switch in the OS scheduler to simulate simultaneous thread execution, which is faithful to how a CMP would execute the threads. Therefore, we adopt the fine-grain context switch method proposed by McCurdy and Fischer [22], as illustrated in Figure 2.4. In the McCurdy and Fischer's method, a centralized scheduler controls which thread is active. Only one thread can be active at a time. The other threads are waiting for the active signal from the scheduler. The active signal is passed in round-robin order, so the memory accesses are interleaved in a consistent order across all threads. The scheduler also simulates the synchronization mechanism of Pthreads.

When we acquire CRD and PRD profiles, we make several assumptions. First,

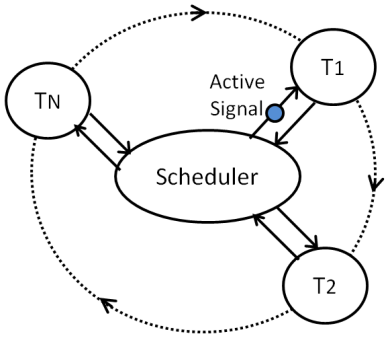


Figure 2.4: Thread interleaving mechanism.

our Pin tool performs functional execution only, context switching between threads every memory reference. Hence, the memory references from each thread are interleaved uniformly in time. Second, in our memory interleaving model, we don't simulate a particular cache hierarchy or CMP architecture. So, there are no timing-related interferences in CRD and PRD profiles. As we will show in our stability study (Section 5.2), this assumption of uniform memory interleaving is accurate enough to acquire profiles for loop-based parallel programs. Third, we also assume the application is the only load on the system. The OS does not interrupt threads. Finally, we assume 64-byte memory blocks.

2.2.2 Benchmarks

Table 2.1 lists our benchmarks used in this research: FFT, LU, Radix, Barnes, FMM, Water, and Ocean from the SPLASH2 suite [23], KMeans from MineBench [24], and BlackScholes from PARSEC [25]. For each benchmark, we employ 4 problem sizes, S1–S4 (2^{nd} column of Table 2.1). We run initialization code on a single core, optionally simulate the beginning of the parallel region, and then turn on CRD and

Table 2.1: Parallel benchmarks used in our study.

Benchmark	Problem Sizes(S1/S2/S3/S4)	Insts Profiled (M)(S1/S2/S3/S4)	Profiled Region
FFT	$2^{16}/2^{18}/2^{20}/2^{22}$ elements	29/129/560/2,420	whole program
LU	$256^2/512^2/1024^2/2048^2$ elements	43/344/2,752/22,007	whole program
RADIX	$2^{18}/2^{20}/2^{22}/2^{24}$ keys	53/211/843/3,372	whole program
Barnes	$2^{13}/2^{15}/2^{17}/2^{19}$ particles	214/1,015/4,438/19,145	1 timestep
FMM	$2^{13}/2^{15}/2^{17}/2^{19}$ particles	235/1,006/4,109/16,570	1 timestep
Ocean	$130^2/258^2/514^2/1026^2$ grid	30/107/420/1,636	1 timestep
Water	$10^3/16^3/25^3/40^3$ molecules	43/143/553/2,099	1 timestep
KMeans	$2^{16}/2^{18}/2^{20}/2^{22}$ objects, 18 features	186/742/2,967/11,874	1 timestep
BlackScholes	$2^{16}/2^{18}/2^{20}/2^{22}$ options	60/242/967/3,867	1 timestep

PRD profiling and continue parallel region simulation for some number of instructions (3^{rd} column of Table 2.1). In FFT, LU, and Radix, profiles are acquired for the entire program. For the other benchmarks, profiles are acquired for only 1 timestep of the algorithm, so we skip the 1^{st} timestep and profile the 2^{nd} timestep.

2.2.3 Architecture-Application Design Space

Processor scaling defines a design space consisting of multicore processors with varied core counts and cache organizations with different capacities. These are architecture design parameters. As processors scale to large core counts and cache capacities, problem size scales, too. Hence, our work also considers the problem size as an independent parameter that can be varied as well. The number of threads and problem sizes are application parameters. Together, these scaling dimensions of architecture and application form a multi-dimensional designed space, as illustrated in Figure 2.5. We call this architecture-application design space ($AADS$).

In this research, each benchmark has 4 problem sizes running on 8 core counts

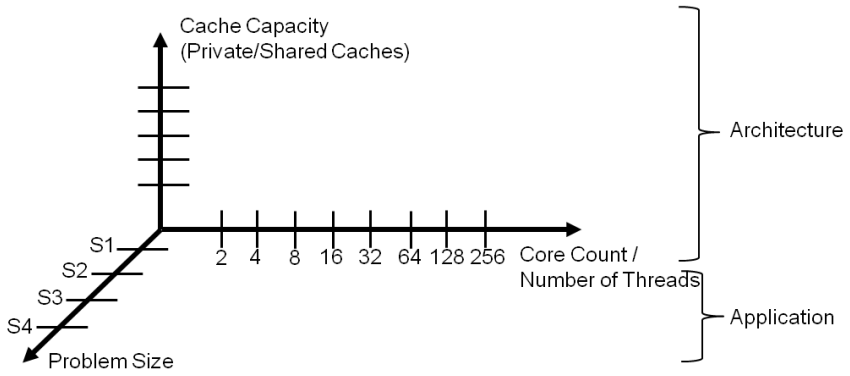


Figure 2.5: Multi-dimensional architecture-application design space (AADS).

from 2 to 256 cores. We also study private and shared caches with varying cache capacities. By comparing CRD (PRD) profiles along any axis, we can characterize profile sensitivity to the corresponding type of scaling.

In our profile prediction study, each benchmark has 32 configurations, and we have a total of 288 configurations across our 9 benchmarks. When we evaluate the accuracy of using CRD and PRD profile predictions to estimate the multicore cache performance (MPKI), we simulate 6 different shared LLC sizes and 5 different private L2 cache sizes. In this case, the design space has a total of 3,168 configurations.

Chapter 3

Multicore Reuse Distance Analysis

The memory behavior on multicore cache systems is the result of intra-thread data locality and inter-thread interferences. In Section 2.1, we review different thread interactions in private and shared caches, and we show how CRD and PRD capture them. In Section 3.1, we further separately quantify these effects by creating several new locality profiles that isolate these thread interactions. This analysis provides rich information about how inter-thread interactions impact an application's memory behavior in private and shared caches.

In Section 3.2–Section 3.4, we study three sources of inter-thread interaction perturbation. The first one is cache capacity scaling; for this, we present our interaction insights at a fixed core count and problem size. The second one is core count scaling, which increases the number of interleaving memory reference streams. The third one is problem size scaling, which increases the memory footprint.

The impact of core count scaling and problem size scaling on CRD and PRD profiles has implications for multicore cache performance. In Section 3.5, we identify two important cache capacities, C_{core} and C_{share}. Then we study their architectural implications.

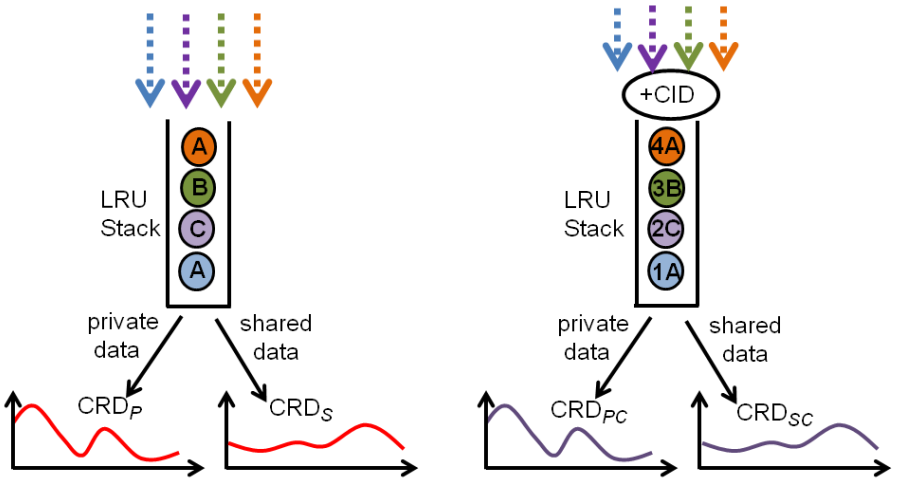
3.1 Quantifying Thread Interactions

For shared caches, inter-thread interactions cause dilation, overlap, and intercept in CRD profiles. On the other hand, for private caches, inter-thread interactions cause replication and invalidation in PRD profiles. To study inter-thread interactions, we isolate these different effects by creating several new locality profiles. To further separate the different locality characteristics of each parallel region in a program, our Pin tool records profiles in between every pair of barrier calls–*i.e.*, per parallel region. Although there might be several parallel loops in the same parallel region, per-parallel region profiling is sufficient for our study.

3.1.1 CRD profiles

Within each parallel region, we acquire CRD profiles for mostly private data and mostly shared data separately. We call the former profiles "private-date CRD (CRD_P) profiles", and we call the latter profiles "shared-data CRD (CRD_S) profiles". We employ a single global LRU stack for computing CRD_P and CRD_S, as illustrated in Figure 3.1(a).

To separate private and shared data blocks, we record each memory block's CRD values separately and the number of times the block is referenced by each core. In our benchmarks, because individual memory blocks tend to exhibit a small number of distinct CRD values, this bookkeeping does not increase storage appreciably. After a parallel region completes, we use a fixed threshold to determine each memory block's sharing status. If a single core is responsible for 90% or more of



(a) CRD_P and CRD_S profiles. (b) CRD_{PC} and CRD_{SC} profiles.

Figure 3.1: Acquiring CRD_P, CRD_S, CRD_{PC}, and CRD_{SC} profiles.

a memory block's references, the block is private; otherwise, it is shared. Then we accumulate all memory blocks' CRD counts into either the CRD_P profile or CRD_S profile based on each block's sharing status.

As mentioned in section 2.1.1, data sharing introduces overlapping references and reduces the dilation. Although the CRD_P profile only has the mostly private data, data sharing still occurs in between data reuses. This is because we measure CRD values from the same stack. Hence, the CRD_P profile represents the combined effect of dilation and overlap, and the amount of intercepts is small. The CRD_S profile also captures data sharing that happens on the reuse data itself. As a result, the CRD_S profile not only contains the dilation and overlap effects, but it also has the intercept effect.

Next, we isolate the sharing-based interactions. We maintain a second global LRU stack, and we prepend every memory block's address with the ID of the core (CID) that performs this memory access, as illustrated in Figure 3.1(b). We call

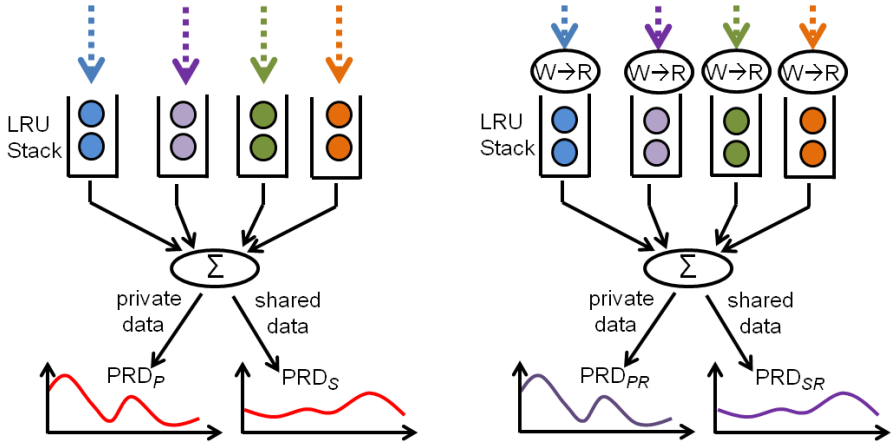
the profiles acquired on this CID-extended stack CRD_{PC} and CRD_{SC} profiles. In these two profiles, inter-thread references are always unique. Comparing CRD_{PC} and CRD_{P} profiles shows the impact of overlap. Similarly, the effect of intercepts due to shared data is also removed. Comparing CRD_{SC} and CRD_{S} profiles shows the combined impact of overlap and intercept.

3.1.2 PRD profiles

For PRD profiles, each core has its own private LRU stack, and the coherent mechanism mentioned in Section 2.1.2 is implemented. We acquire PRD profiles for mostly private data and mostly shared data separately within each parallel region. We call the former profiles "private-data PRD (PRD_{P}) profiles", and we call the latter profiles "shared-data PRD (PRD_{S}) profiles". After a parallel region completes, we sum up these per-thread PRD_{P} profiles to create a single PRD_{P} profile, and we sum up these per-thread PRD_{S} profiles to create a single PRD_{S} profile, as illustrated in Figure 3.2(a). These two profiles represent overall per-thread memory behavior for mostly private data and mostly shared data.

As mentioned in section 2.1.2, read-shared data causes replications, and write-shared data causes invalidations in private stacks. Both PRD_{P} and PRD_{S} profiles contain the combined effect of replication and invalidation because we measure PRD values on the same per-core stack. The re-reference of invalidated data causes cache misses. These "coherence misses" appear at the infinite PRD value.

Next, we remove write-sharing to isolate hole-related interactions. This is done



(a) PRD$_P$ and PRD$_S$ profiles. (b) PRD$_{PR}$ and PRD$_{SR}$ profiles.

Figure 3.2: Acquiring PRD$_P$, PRD$_S$, PRD$_{PR}$, and PRD$_{SR}$ profiles.

by converting writes to reads, as illustrated in Figure 3.2(b). The only effect left in profiles is replication. Profiles acquired on read-conversion stacks are called PRD$_{PR}$ and PRD$_{SR}$ profiles. Comparing PRD$_P$ with PRD$_{PR}$ profiles shows the absorption impact due to holes. Comparing PRD$_S$ and PRD$_{SR}$ profiles shows the impact of holes and coherence misses.

3.2 Thread Interactions Analysis at a Fixed Core Count and Problem Size

This section applies the isolation techniques introduced in Section 3.1 to the study of inter-thread interactions. We analyze two specific benchmarks, FFT and Barnes. Although each benchmark has different interactions, all parallel regions exhibit very similar behavior. The insights gathered from FFT and Barnes can generally represent inter-thread interactions for our benchmarks.

Although PRD profiles are based on per-core stacks, the multiple private stacks

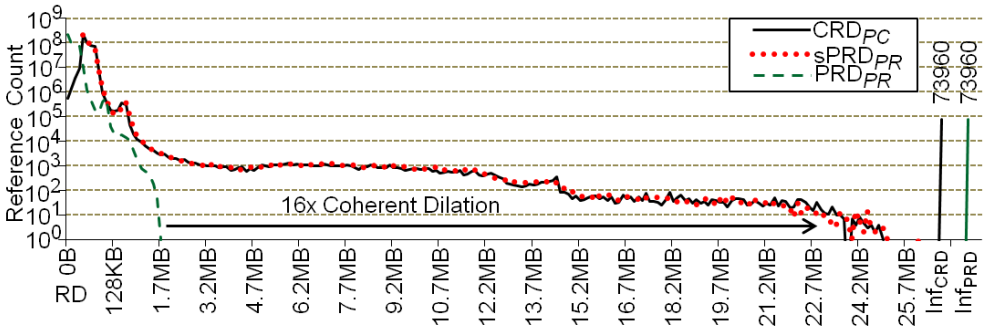
still contribute to increased cache capacity on multicore processors. To capture this effect, we compute the *scaled PRD*, or sPRD, which equals $T \times PRD$, where T is the number of threads. Because both CRD and sPRD reflect total cache capacity, we can compare the cache performance between shared and private caches across different cache sizes by comparing CRD and sPRD profiles directly.

Figure 3.3 and Figure 3.4 shows different CRD and PRD profiles for the most important parallel region in Barnes and FFT running on 16 cores at the S2 problem size. In each graph, the Y-axis is the reference count in log10 scale, and the X-axis is the RD value in terms of cache capacity. This is done by multiplying RD values by the cache block size, 64 bytes. In this study, for each profile, reference counts from multiple adjacent RD values are summed into a single RD bin, and plotted as a single Y value. For capacities 0–128KB, bin size grows logarithmically; beyond 128KB, all bins are 128KB each.

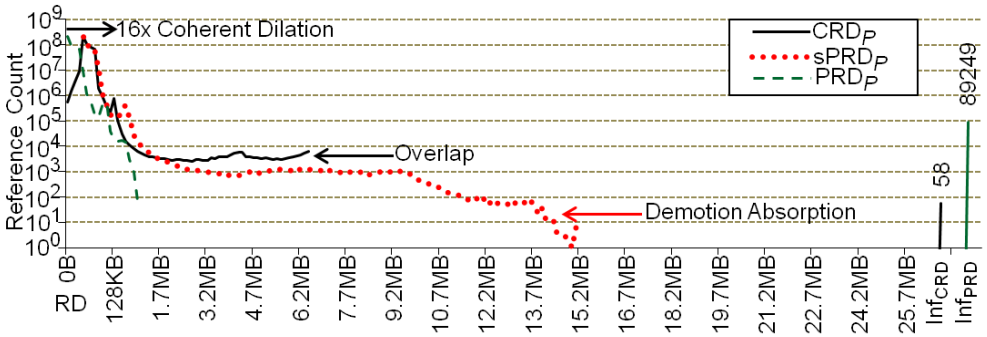
3.2.1 Private-data Profiles

Figure 3.3(a) and Figure 3.4(a) plot CRD_{PC} and sPRD_{PR} profiles along with PRD_{PR} profile. As described in Section 3.1, there are no sharing-induced interactions in CRD_{PC} and sPRD_{PR} profiles. Comparing CRD_{PC} and PRD_{PR} profiles shows the dilation effect, and comparing sPRD_{PR} and PRD_{PR} profiles shows the scaling effect.

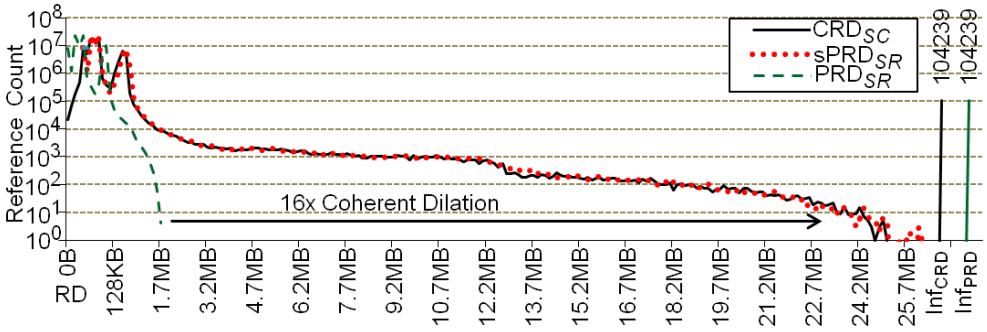
In Figure 3.3(a) and Figure 3.4(a), the sPRD_{PR} profile is a 16x scaling of the PRD_{PR} profile. This is because sPRD profiles are the scaled versions of PRD



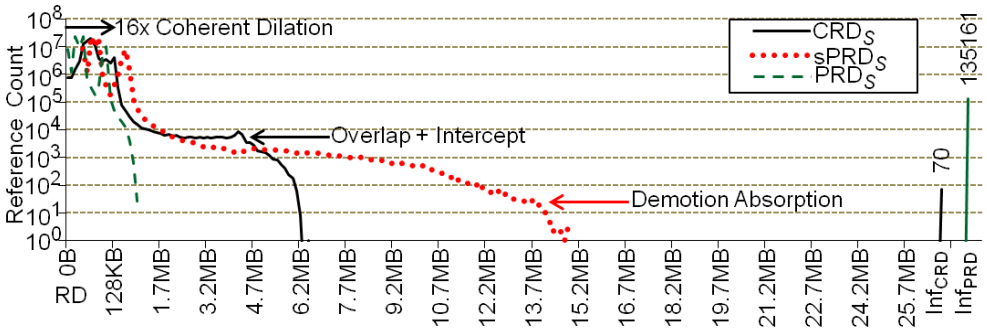
(a) Private-data profiles showing dilation and scaling.



(b) Private-data profiles showing overlap and demotion absorption.

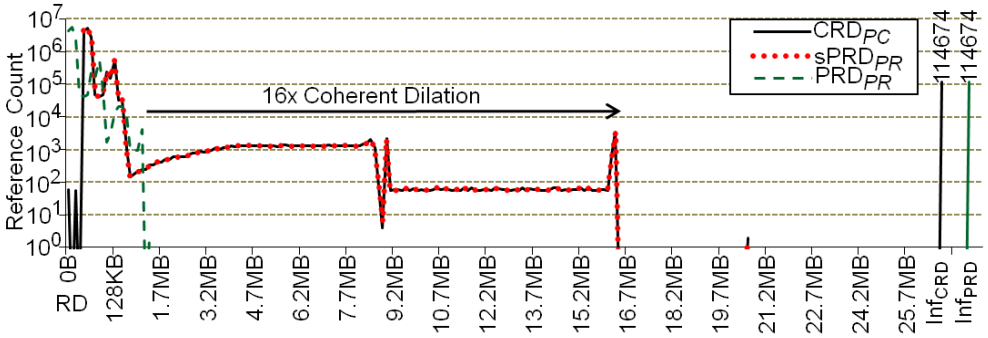


(c) Shared-data profiles showing dilation and scaling.

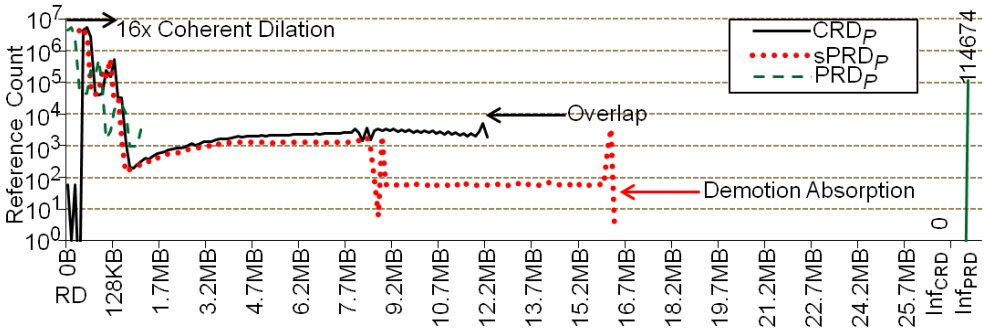


(d) Shared-data profiles showing overlap+intercept and demotion absorption.

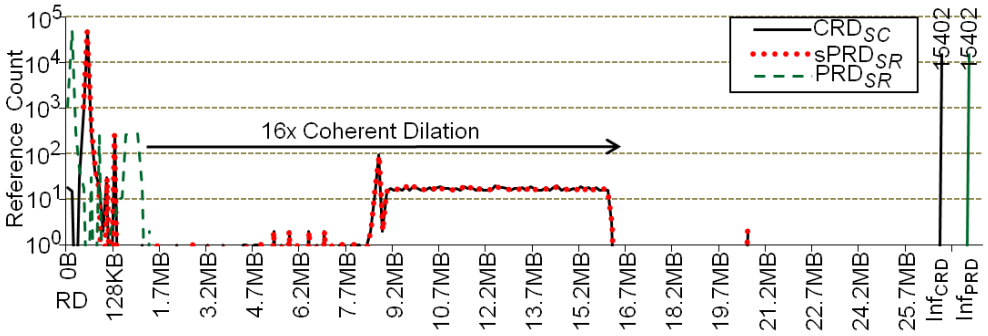
Figure 3.3: Barnes' locality profiles for the most important parallel region running on 16 cores at the S2 problem size.



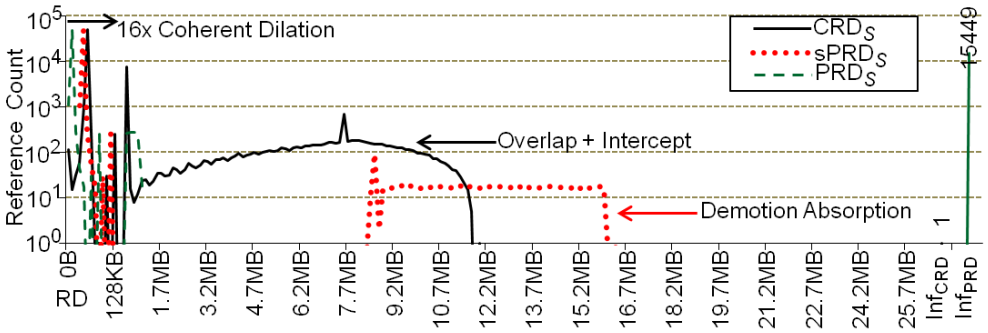
(a) Private-data profiles showing dilation and scaling.



(b) Private-data profiles showing overlap and demotion absorption.



(c) Shared-data profiles showing dilation and scaling.



(d) Shared-data profiles showing overlap+intercept and demotion absorption.

Figure 3.4: FFT's locality profiles for the most important parallel region running on 16 cores at the S2 problem size.

profiles. An interesting observation is that the CRD_{PC} profile is almost identical to the $sPRD_{PR}$ profile, and the CRD_{PC} profile is also a 16x scaling of the PRD_{PR} profile. This is because symmetric threads are interleaved systematically in the same parallel region. In the shared cache, the intra-thread data reuse at a particular RD is interleaved by the same amount of RD from each of the other simultaneous threads. This effect is called *dilation*. In this example, the dilation is by exactly a factor of 16x. As a result, scaling and dilation both shift the PRD_{PR} profile in a shape-preserving way and degrade data locality at the same rate, *i.e.*, linear with the number of threads.

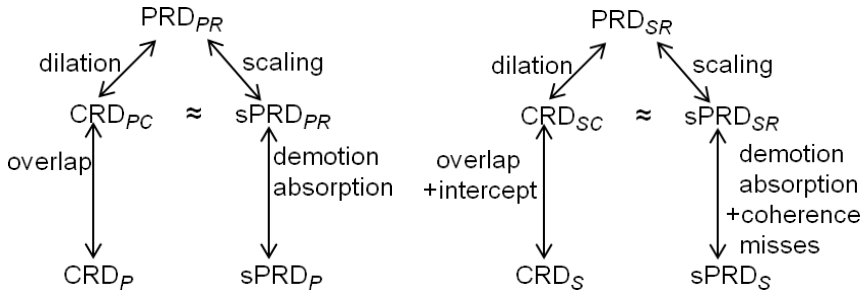
When there is no data sharing, shared and private caches show the same data locality behavior. However, when data sharing happens, shared and private caches have different sharing-related interactions. Figure 3.3(b) and Figure 3.4(b) illustrates the CRD_P, $sPRD_P$, and PRD_P profiles of Barnes and FFT, respectively. As described in Section3.1, comparing CRD_{PC} and CRD_P profiles shows the impact of overlapping references, and comparing $sPRD_{PR}$ (PRD_{PR}) and $sPRD_P$ (PRD_P) profiles shows the impact of invalidated references.

CRD_P profiles terminate before CRD_{PC} profiles. As discussed in Section 3.1, data sharing introduces overlap that reduces dilation in CRD profiles, and the amount of reduction depends on the degree of data sharing. CRD_P and CRD_{PC} profiles are almost identical at small RD values. As RD value increases, the CRD_P profile exhibits less shift. In our benchmarks, programmers tend to share data across distant loop iterations, so data sharing tends to happen at large reuse windows only. As a result, overlapping references rarely happen in small reuse distance windows

for CRD profiles.

Data sharing introduces *demotion absorption* in sPRD profiles. At small RD values, sPRD_P and sPRD_{PR} profiles are almost identical. This is because invalidated references rarely happen in small reuse windows for sPRD profiles. As RD value increases, sPRD_P profiles exhibit less shift and terminate before sPRD_{PR} profiles. When there are few invalidations as in FFT, sPRD_{PR} and sPRD_P are practically identical, even at large RD values. However, when there are more invalidations, as in Barnes, the holes reduce the shift significantly.

Lastly, the amount of contraction in CRD_P and sPRD_P profiles varies with reuse distance. Because the contraction comes from the inter-thread interactions of sharing data, its presence or absence along CRD_P and sPRD_P profiles permits assessing the degree of data sharing as a function of reuse distance. As illustrated in Figure 3.3(b) and Figure 3.4(b), CRD_P and sPRD_P profiles are almost identical at small RD values. Then data sharing begins to affect CRD_P and sPRD_P profiles. The contraction increases as RD value grows, and finally causes the different termination of CRD_P and sPRD_P profiles. CRD_P profiles end earlier than sPRD_P profiles. Although invalidations create holes and reduce the reuse distance, this absorption effect is smaller than the overlap effect. The is because write-shared data makes up only a portion of the total shared data, and replications caused by read-shared data also degrades data locality in private caches. Figure 3.5(a) illustrates the relationship between profiles for private data.



(a) Private data. (b) Shared data.

Figure 3.5: Quantifying individual thread interaction effects.

3.2.2 Shared-data Profiles

Figure 3.3(c)-(d) and Figure 3.4(c)-(d) plot the shared-data profiles of Barnes and FFT. The shared-data profiles exhibit behavior very similar to the corresponding private-data profiles. First, the dilation and scaling are equivalent for parallel loops in the absence of data sharing. As a result, CRD_{SC} and sPRD_{SR} profiles are almost identical, and both show the coherent shift by a factor of 16x in a shape preserving way with respect to the PRD_{SR} profile. The CRD_S profile has the effect of overlap, and the sPRD_S profile has the effect of demotion absorption. Both profiles show contraction, but the CRD_S profile shrinks more than the sPRD_S profile. The reasons are described in Section 3.2.1.

Figure 3.3(d) and Figure 3.4(d) also show the effect of intercepts in CRD_S profiles, and the effect of invalidations in sPRD_S profiles. As described in Section 2.1.1, the intercept splits intra-thread reuse windows, with the resulting CRD value depending on the intercepted location. Because intercepts can happen randomly at any location, the CRD values of intercepted data blocks can be any value between 0 and the max CRD value. In our benchmarks, because data sharing usually happens

at large reuse windows, intercept tends to spread the reference counts at large RD values in CRD_S profiles. This effect is visible clearly in FFT.

Invalidations create holes in private stacks, and the consequent references to the already-invalidated blocks have infinite reuse distance. So the increasing cache misses at infinite reuse distance show the cache performance degradation due to coherence misses. Another important observation is that the holes have the same impact on sPRD_S and sPRD_P profiles, because holes reduce the depth for both shared and private data in stacks. Figure 3.5(b) summarizes the relationship between profiles for shared data.

Lastly, in our benchmarks, we find private-data profiles dominate shared-data profiles. For example, the amount of private references is 6x and 259x more than the amount of shared references in Barnes and FFT, respectively. As a result, dilation and overlap in CRD profiles along with scaling and demotion absorption in sPRD profiles determine overall shared/private cache performance.

3.3 Thread Interactions Analysis for Core Count Scaling

When core count increases but problem size stays fixed (*i.e.*, strong scaling), core count scaling reduces each thread's working set size. More threads also increase inter-thread interactions. Figure 3.6 depicts a simple example which parallelizes a vector operation for P cores. In the sequential program, each cache block contains four data elements, so there are a total of $M/4$ cache blocks. Each cache block is referenced four times (re-referenced three times) before the inner for-loop advances

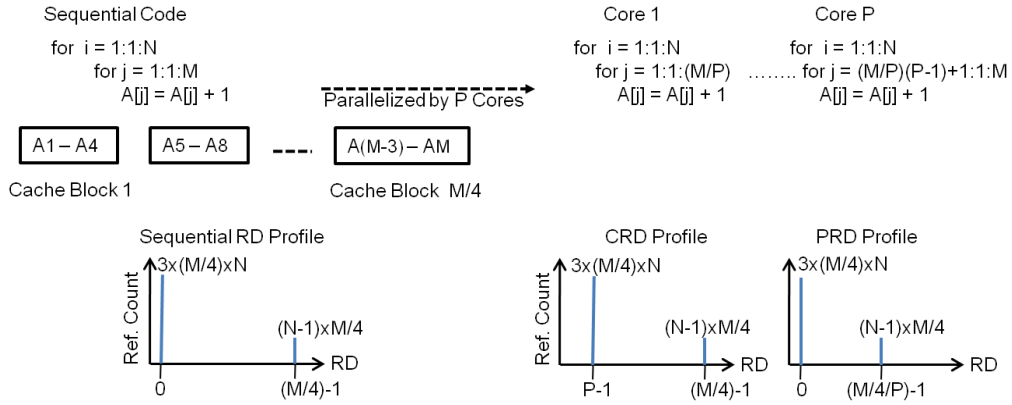


Figure 3.6: A simple example showing how CRD and PRD shift with core count scaling. Each cache block contains 4 elements.

to the next cache block. The outer for-loop re-accesses each cache block with reuse distance $(M/4) - 1$, and there are $N - 1$ re-references for each block.

In the parallel program, the inner loop is partitioned into P chunks, and each core has M/P elements. For the CRD profile, the uniform interleaving causes the RD value of re-references at the inner-loop to move to $P - 1$. However, the re-references at the outer-loop stay at the same RD value, $(M/4) - 1$. This is because core count scaling does not increase the total amount of global data, so the theoretical max RD value doesn't change. As a result, when core count increases, the references at small RD values move to larger RD values. However, the CRD profiles of different core counts eventually end at the same RD value.

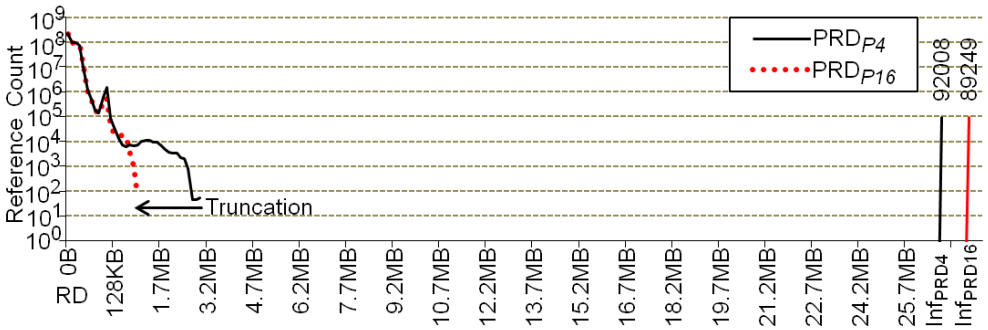
In contrast, PRD profiles truncate as core count increases. The references at small RD values do not move due to the absence of interleaving. However, the re-references at the outer-loop move to smaller RD values due to the reduction of per-thread working set size. In this example, the max RD value of the PRD profile moves from $(M/4) - 1$ to $(M/4/P)/ - 1$. As a result, when core count increases, the PRD

profile truncates. This simple example shows the major inter-thread interactions as core count scales. In the following sections, we conduct a detailed study to understand how core count scaling impacts CRD and PRD profiles.

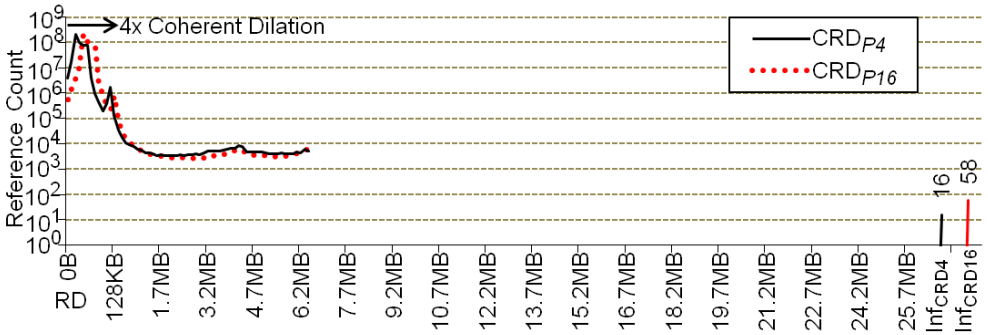
3.3.1 Private-data Profiles

Figure 3.7 and Figure 3.8 illustrate the private-data profiles for the most important parallel region in Barnes and FFT running on 4 cores and 16 cores at the S2 problem size. First, we compare PRD_{P4} and PRD_{P16} profiles to study the per-thread locality impact due to core count scaling. In Figure 3.7(a) and Figure 3.8(a), PRD_{P4} and PRD_{P16} profiles exhibit very similar shapes because threads on 4 and 16 cores execute the same code. At small RD values, PRD_{P4} and PRD_{P16} profiles are almost identical. This is because this region reflects memory references executed within contemporaneous computation. So core count scaling doesn't affect the locality. Then PRD_{P4} and PRD_{P16} profiles split at a certain RD value, and finally the PRD_{P16} profile ends earlier than the PRD_{P4} profile due to the reduction of per-thread working set size.

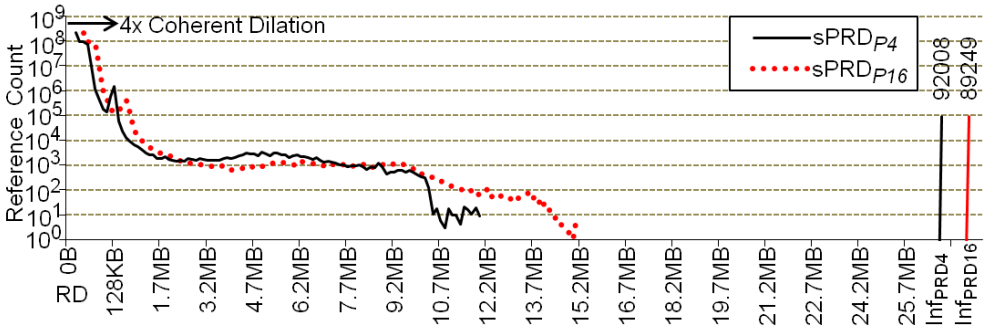
This truncation, along with overlapping references at large RD values, almost perfectly cancel the dilation due to core count scaling in CRD profiles, as illustrated in Figure 3.7(b) and Figure 3.8(b). Because symmetric threads are interleaved systematically, the CRD_{P16} profile is not only a coherent shift of the PRD_{P16} profile, but it is also a coherent shift of the PRD_{P4} profile and the CRD_{P4} profile. At small RD values, the CRD_{P16} profile coherently scales the CRD_{P4} profile by a factor of



(a) PRD$_P$ on 4 and 16 cores.



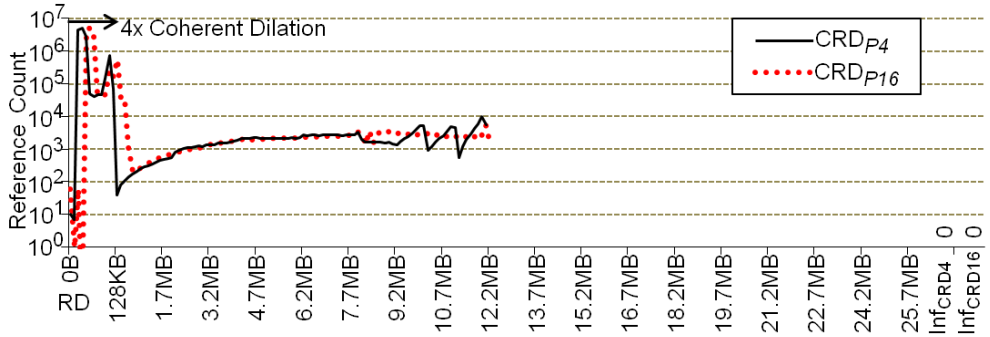
(b) CRD$_P$ on 4 and 16 cores.



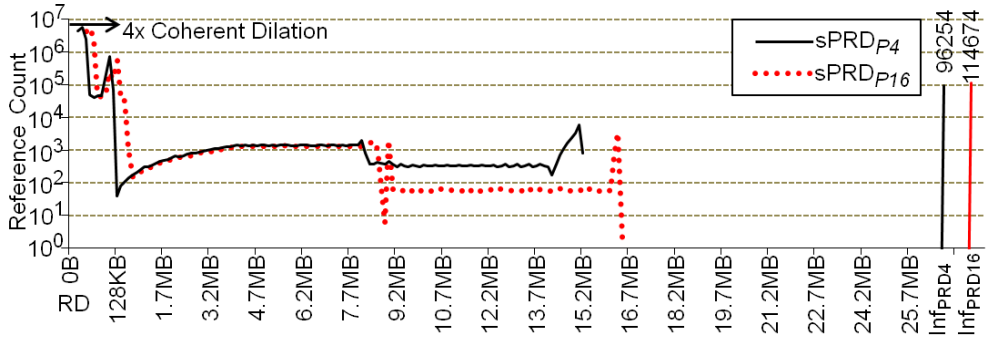
(c) sPRD$_P$ on 4 and 16 cores.

Figure 3.7: Barnes' private-data locality profiles running on 4 cores and 16 cores at the S2 problem size.

(a) PRD$_P$ on 4 and 16 cores.



(b) CRD$_P$ on 4 and 16 cores.



(c) sPRD$_P$ on 4 and 16 cores.

Figure 3.8: FFT's private-data locality profiles running on 4 cores and 16 cores at the S2 problem size.

4x. At large RD values, shifting slows down and eventually stops due to the effects of truncation and overlap. So CRD_{P16} and CRD_{P4} profiles merge, and end at about the same RD value. This makes sense: because core count scaling does not change the amount of global data, the theoretical maximum RD value is roughly the same. This analysis shows core count scaling degrades data locality for shared caches, but its impact is limited to small capacities. We will discuss this further in Section 3.5.1.

Lastly, Figure 3.7(c) and Figure 3.8(c) show the 4- and 16-core $sPRD_P$ profiles. Because the $sPRD_P$ profile is a scaled version of PRD_P profile, the $sPRD_P$ profile shifts to larger RD values with respect to the PRD_P profile. Again, because the PRD_{P4} and PRD_{P16} profiles are almost identical at small RD values, the $sPRD_{P16}$ profile exhibits the coherent shift by a factor of 4x compared to the $sPRD_{P4}$ profile. At large RD values, the truncation and demotion absorption reduce the effect of scaling, but the $sPRD_{P16}$ profile still maintains some shifting relative to the $sPRD_{P4}$ profile due to the smaller degree of contraction caused by demotion absorption compared to overlap. So, like CRD_P profiles, $sPRD_P$ profiles also shift non-uniformly. However, our analysis shows that core count scaling degrades data locality in private caches more than in shared caches, since core count scaling affects $sPRD_P$ profile across a larger range of cache capacities.

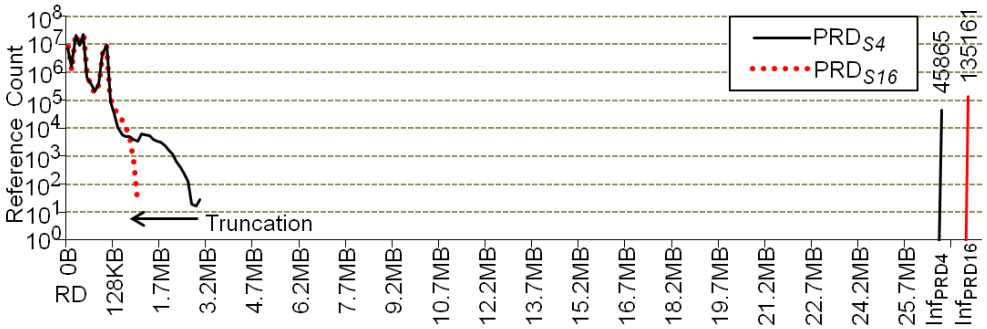
3.3.2 Shared-data Profiles

Core count scaling also impacts shared-data profiles. Figure 3.9 and Figure 3.10 plot Barnes' and FFT's shared-data profiles at 4 cores and 16 cores. In

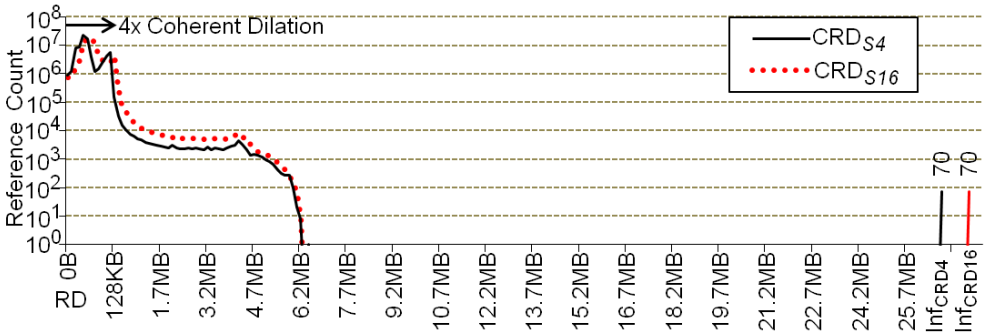
Figure 3.9(a) and Figure 3.10(a), PRD_{S4} and PRD_{S16} profiles exhibit very similar shapes. At small RD values, PRD_{S4} and PRD_{S16} profiles are almost identical. Then PRD_{S4} and PRD_{S16} profiles split at a certain RD value, and finally the PRD_{S16} profile ends earlier than the PRD_{S4} profile due to the truncation of per-thread working set size. As a result, when we scale the PRD_S profile across core counts, the $sPRD_{S16}$ profile exhibits the coherent shift by a factor of 4x compared to the $sPRD_{S4}$ profile at small RD values. At large RD values, the truncation reduces the $sPRD_{S16}$ profile's scaling, but the $sPRD_{S16}$ profile still maintains some shifting relative to the $sPRD_{S4}$ profile, as illustrated in Figure 3.9(c) and Figure 3.10(c). In addition, core count scaling also leads to a higher number of replications and invalidations. In Barnes, the total reference counts of $sPRD_P$ and $sPRD_S$ profiles at the infinite RD value increase from 137,873 to 224,410 as core count scales from 4 to 16.

The intercept effect in CRD_S profiles is more complicated. Figure 3.9(b) and Figure 3.10(b) plot Barnes' and FFT's CRD_S profiles at 4 and 16 cores, and show how the effect of intercept changes with core count scaling and applications. First, at small RD values, the CRD_{S16} profile exhibits the coherent shift by a factor of 4x compared to the CRD_{S4} profile. This is because that data sharing tends to occur across distant loop interactions. So the effect of intercepts, like overlap, rarely appears within small reuse windows. As a result, when there are few intercepts, CRD_S profiles scale like CRD_P profiles.

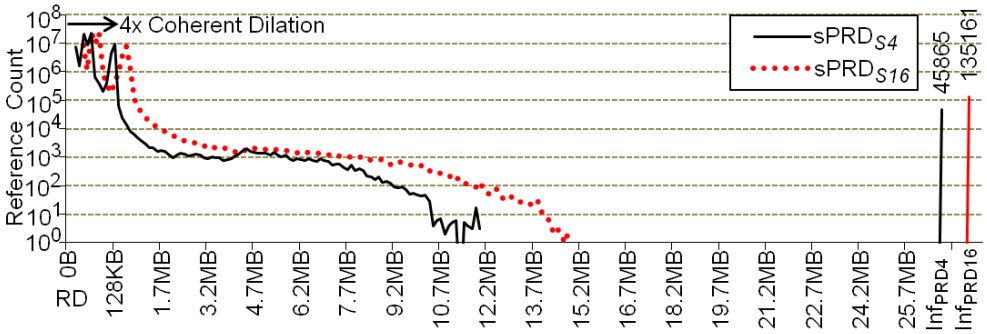
And second, at large RD values, intercepts happen more often. As described in Section 2.1.1, intercepts induce spreading and change CRD_S profiles. However, the spreading depends on where intercepts appear within intra-thread reuse windows



(a) PRD$_S$ running on 4 and 16 cores.

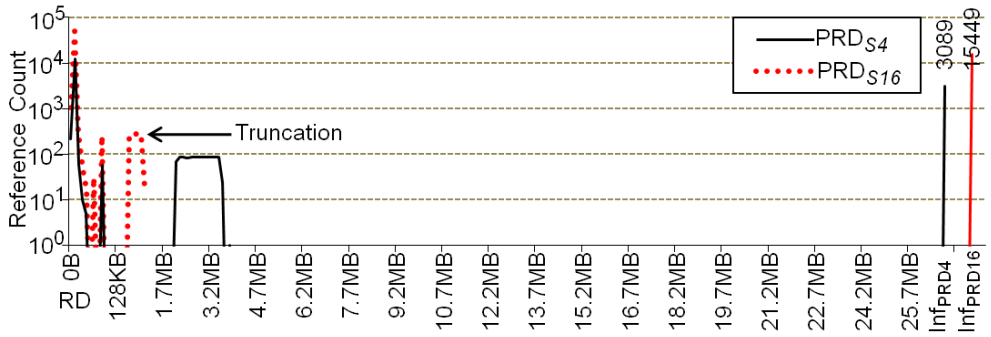


(b) CRD$_S$ running on 4 and 16 cores.

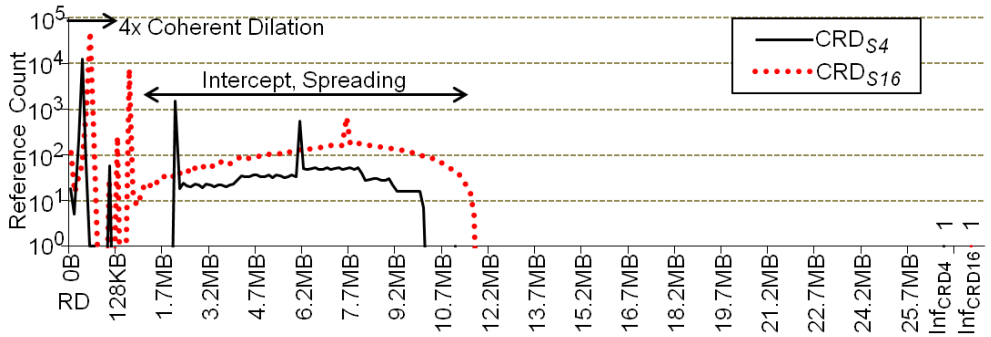


(c) sPRD$_S$ running on 4 and 16 cores.

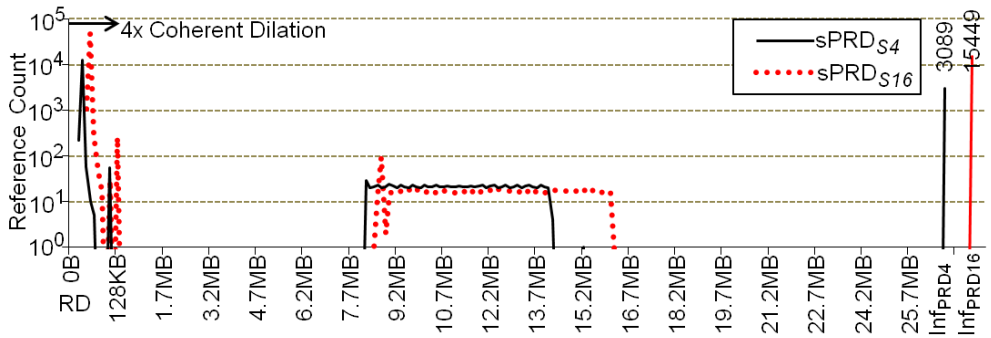
Figure 3.9: Barnes' shared-data locality profiles running on 4 cores and 16 cores at the S2 problem size.



(a) PRD_S running on 4 and 16 cores.



(b) CRD_S running on 4 and 16 cores.



(c) $sPRD_S$ running on 4 and 16 cores.

Figure 3.10: FFT's shared-data locality profiles running on 4 cores and 16 cores at the S2 problem size.

and on the frequency of intercepts. In Figure 3.9(b), intercepts in Barnes don't cause significant spreading. The major effects of core count scaling are dilation and overlap. In contrast, as illustrated in Figure 3.10(b), intercepts in FFT cause a significant spreading. This spreading stretches CRD_{S4} profiles toward both smaller and larger RD values. Although intercepts in shared caches may cause more complicated shifting on CRD_S profiles, CRD_S profiles contain fewer memory references than CRD_P profiles in our benchmarks. As a result, while the exact percentage is application dependent, we find CRD_P profiles always dominate in our benchmarks.

3.4 Thread Interactions Analysis for Problem Size Scaling

Problem size scaling at a particular core count increases each thread's working set size. Hence, the total number of references increases, and the reuse distance profile shifts to larger RD values. Figure 3.11 uses the same example as in Figure 3.6 to explain these two effects. When vector length increases from M to M', the total references increase by a factor of M'/M. For CRD profiles, the uniform interleaving causes the RD value of re-references at the inner-loop to remain at $P-1$, but the RD values of re-references at the outer-loop moves to $(M'/4)-1$. For PRD profiles, the RD value of re-references at the inner-loop remains at 0, but the RD value of the re-references at the outer-loop moves to $(M'/4/P)-1$. Hence, the problem size scaling at a particular core count does not move the references at small RD values because these references' locality are insensitive to the input data. However, the problem size scaling increases the reuse window at large RD values.

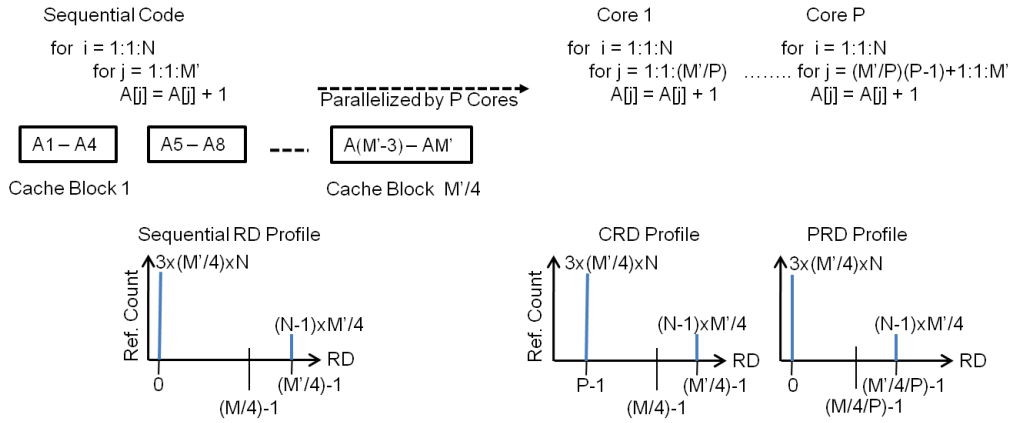
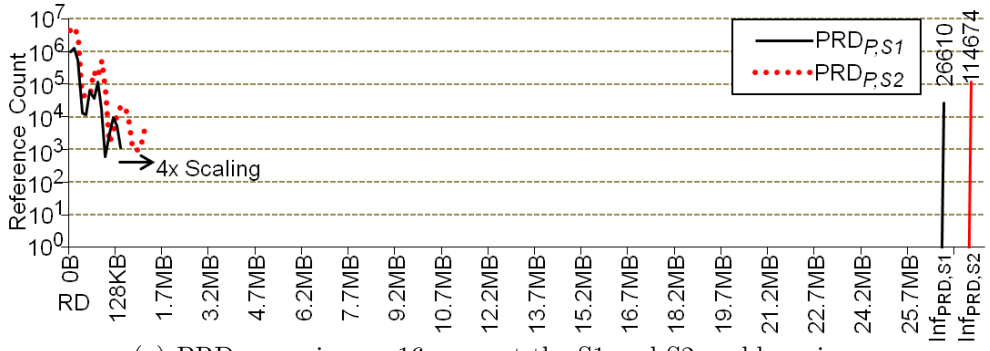


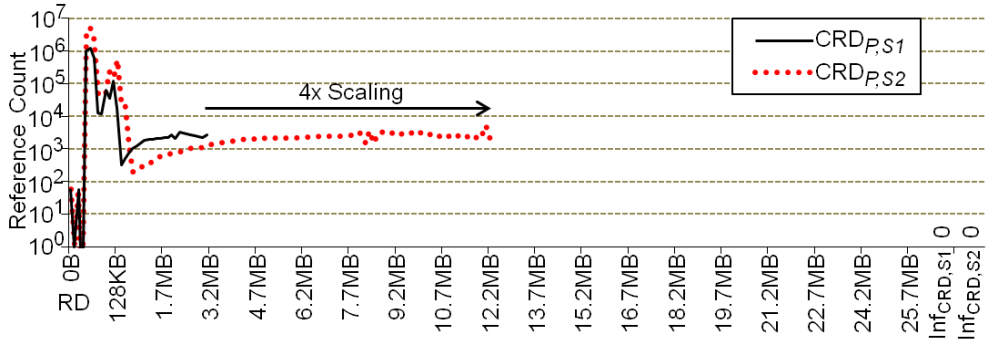
Figure 3.11: A simple example showing how CRD and PRD shift with problem size scaling. Each cache block contains 4 elements.

Figure 3.12 and Figure 3.13 show the private-data and shared-data profiles of FFT's most important parallel region running on 16 cores at the S1 and S2 problem sizes. Figure 3.12(a) plots PRD_P profiles at the S1 and S2 problem sizes. $\text{PRD}_{P,S1}$ and $\text{PRD}_{P,S2}$ profiles show two major effects due to problem size scaling. First, $\text{PRD}_{P,S1}$ and $\text{PRD}_{P,S2}$ profiles have similar shapes, but the $\text{PRD}_{P,S2}$ profile has higher reference counts. Second, problem size scaling causes the $\text{PRD}_{P,S2}$ profile to end at a large RD value. This is because the memory footprint increases, and the max RD value increases by about a factor of 4x. As a result, in problem size scaling, the profile shift along the X-axis occurs at large RD values. The reason shifting stops below a certain RD is because these references are often associated with computations that do not scale with problem size. Problem size scaling causes the same impact on CRD_P and sPRD_P profiles, as illustrated in Figure 3.12(b) and Figure 3.12(c).

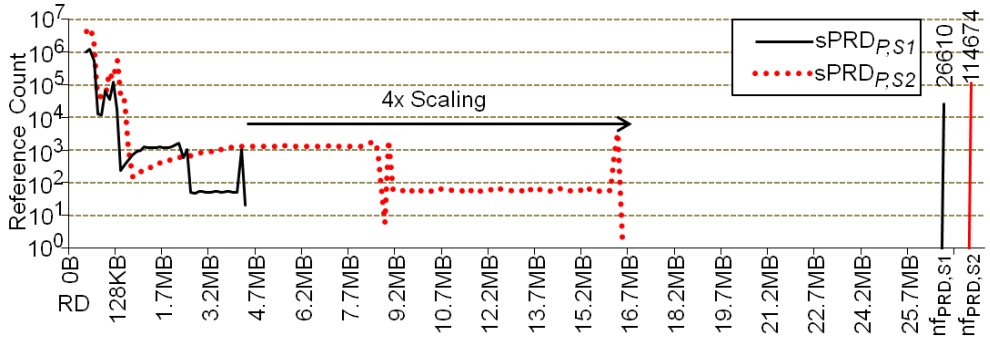
Figure 3.13 plots the PRD_S, sPRD_S, and CRD_S profiles at the S1 and S2 problem sizes. For shared data, problem size scaling induces the same stretching



(a) PRD_P running on 16 cores at the S1 and S2 problem sizes.

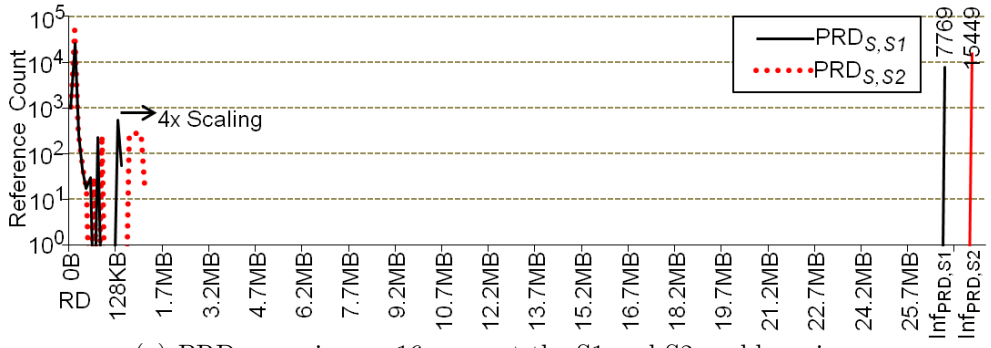


(b) CRD_P running on 16 cores at the S1 and S2 problem sizes.

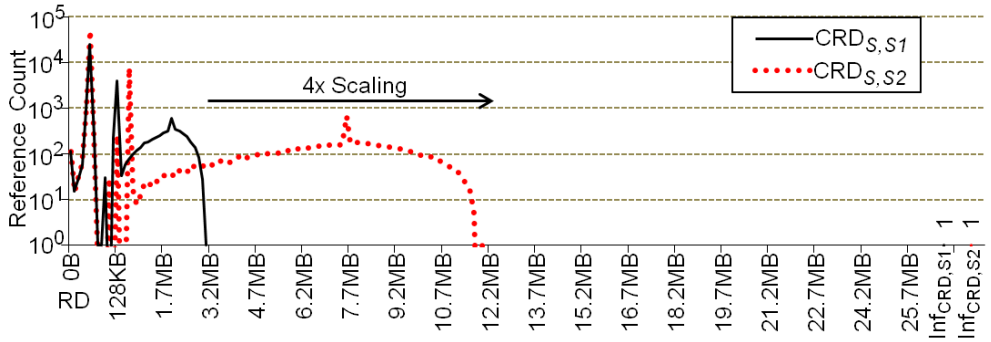


(c) sPRD_P running on 16 cores at the S1 and S2 problem sizes.

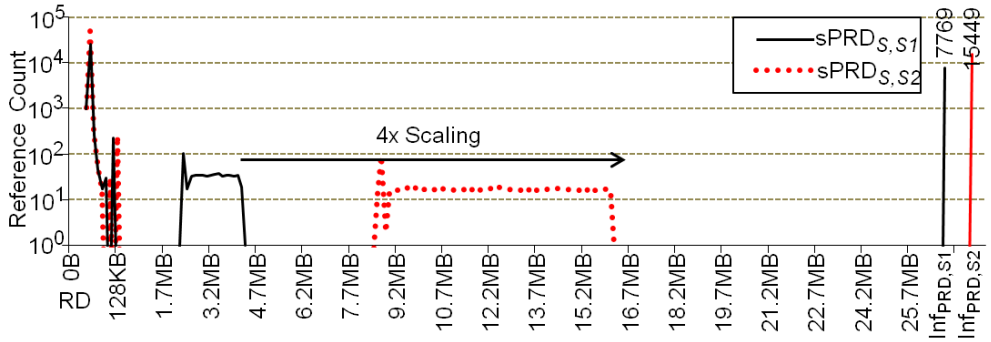
Figure 3.12: FFT's private-data locality profiles running on 16 cores at the S1 and S2 problem sizes.



(a) PRD$_S$ running on 16 cores at the S1 and S2 problem sizes.



(b) CRD$_S$ running on 16 cores at the S1 and S2 problem sizes.



(c) sPRD$_S$ running on 16 cores at the S1 and S2 problem sizes.

Figure 3.13: FFT's shared-data locality profiles running on 16 cores at the S1 and S2 problem sizes.

behavior. As a result, problem size scaling causes less complicated movement compared to core count scaling.

3.5 Architectural Implications

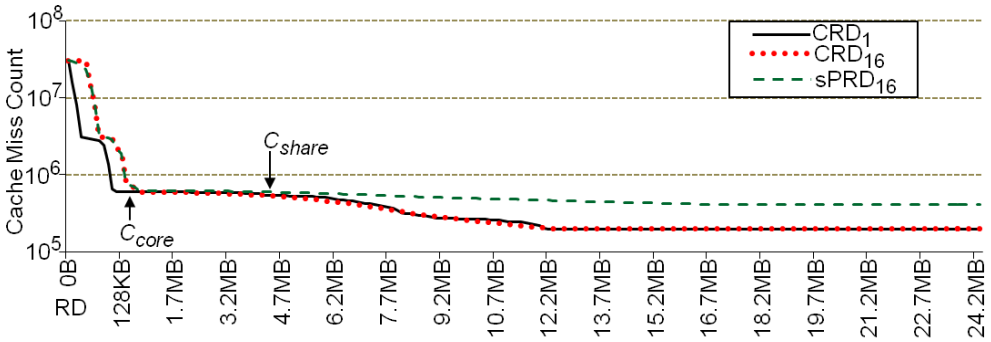
The impact of core count scaling and problem size scaling on CRD and PRD profiles has implications for multicore cache performance. To illustrate this, we compare cache miss count (CMC) profiles derived from CRD and sPRD profiles. The number of cache misses incurred at capacity i in a CRD and sPRD profile are defined in Equation 3.1, where N is the number of bins.

$$\begin{aligned} CRD\_CMC[i] &= \sum_{j=i}^{N-1} CRD[j] + CRD[Inf] \\ sPRD\_CMC[i] &= \sum_{j=i}^{N-1} sPRD[j] + sPRD[Inf] \end{aligned} \tag{3.1}$$

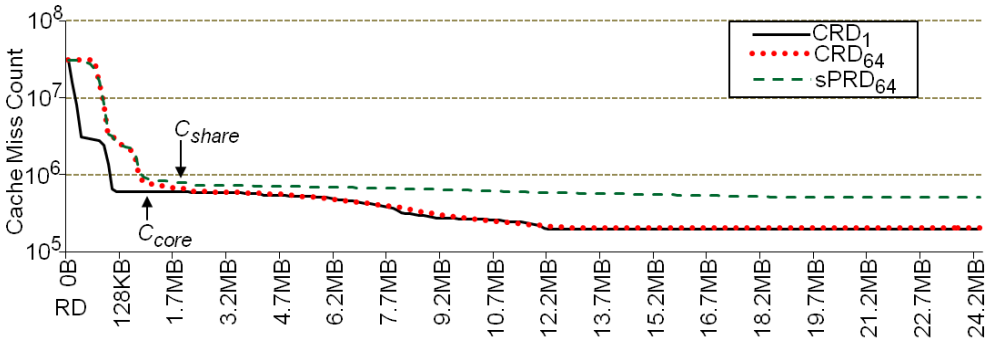
In this section, we first characterize how core count scaling (*i.e.*, strong scaling) impacts performance of shared caches and private caches. Then we extend the study to problem size scaling, and core-problem scaling (*i.e.*, weak scaling).

3.5.1 Core Count Scaling

As described in Section 3.3, the data locality degradation within a shared cache is limited to smaller cache capacities. This implies core count scaling has very little impact on cache performance when the shared cache is beyond a certain capacity. To illustrate, Figure 3.14(a) shows the whole-program CRD_CMC profiles for the FFT benchmark running the S2 problem size on 1 and 16 cores. Because CRD profiles



(a) CRD_CMC and SPRD_CMC profiles running on 1 core and 16 cores.



(b) CRD_CMC and SPRD_CMC profiles running on 1 core and 64 cores.

Figure 3.14: FFT's CMC profiles running on 1, 16, and 64 cores at the S2 problem size.

eventually stop shifting, their associated CMC profiles merge at a certain point. In this study, we call this point "C_{core}." As Figure 3.14(a) shows, C_{core} delineates cache-miss impact. At RD $< C_{core}$, cache misses increase significantly with core count scaling, but cache misses do not increase much when RD $> C_{core}$. In other words, *core count scaling degrades locality, but its impact is confined to smaller RD values*. This implies caches smaller than C_{core} will incur large cache-miss increases with core count scaling, but caches bigger than C_{core} will not. Because the shifting region grows as core count scales, C_{core} grows as core count scales, too. Figure 3.14(b) shows that the C_{core} grows from 210KB to 688KB when scaling from 16 to 64 cores.

In contrast, core count scaling degrades the data locality of private caches across all cache capacities. As a result, there exists a gap between the CRD_CMC and sPRD_CMC profiles that represents the difference between shared and private cache performance which is a function of cache capacity. To illustrate, Figure 3.14(a) plots sPRD_CMC profiles on top of CRD_CMC profiles. Figure 3.14(a) shows that private and shared caches incur very similar cache misses at small cache capacities. As described in Section 3.2, this is because there is little data sharing in this region. The effects of dilation and scaling are very similar, so CRD and sPRD profiles are almost identical. At larger capacities, the overlap effect in CRD profiles cause more contraction than the demotion absorption effect in sPRD profiles. Private caches also have cache misses due to replications and invalidations. As a result, CRD_CMC and sPRD_CMC profiles begin to diverge at a certain cache capacity. Beyond this capacity, shared caches begin to show an advantage over private caches in terms of cache misses. We call this point "C_{share}."

Because core count scaling increases the amount of replication and invalidation, the gap between SPRD_CMC and CRD_CMC profiles indeed increases when scaling from 16 to 64 cores. In addition, core count scaling also moves the sharing point, C_{share}, to smaller RD values. Comparing Figure 3.14(a) and Figure 3.14(b), we see that as SPRD_CMC increases relative to CRD_CMC, C_{share} moves to the smaller RD value. This makes sense. Because core count scaling distributes the same amount of work across more cores, the data sharing frequency is likely to increase, too.

We measured C_{core} and C_{share} across the entire architecture-application design space (AADS),which is illustrated in Figure 2.5. This was done as follows. For

every core count and problem size in the AADS, we derive the CMC profiles for 1–256 cores. At a given cache capacity, we define ΔM to be the ratio of cache-miss counts between the P- and 1-core CMC profiles. We first compute ΔM at CRD = $\frac{maxbin}{2}$, well beyond C_{core} where the CMC profiles have almost merged. We call this ΔM_{merged}. Then, we identify the cache capacity closest to $\frac{maxbin}{2}$ where ΔM =1.5 $\times\Delta M_{merged}$–*i.e.*, the tail-end of shifting where very large ΔM transition to ΔM_{merged}. This capacity is C_{core}. Then, we compute ΔM at every cache capacity between 1MB and C_{core}, recording the average and maximum values. These are the average and maximum cache-miss increases between 1MB and C_{core}, ΔM_a and ΔM_m, respectively. Lastly, we also quantify C_{share}. We begin from the end of CRD_CMC and sPRD_CMC profiles, and we trace these two profiles backward until we reach the point where CRD_CMC and sPRD_CMC are within 10%. This capacity is C_{share}.

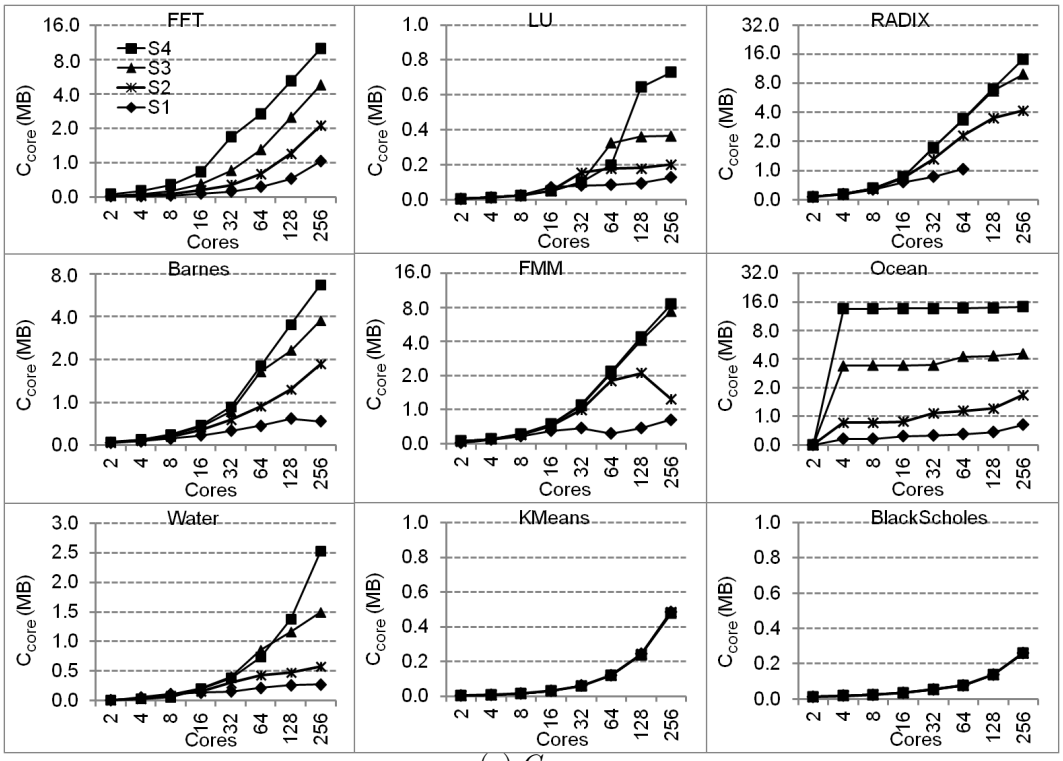
Figure 3.15 reports C_{core} and C_{share} across all of our benchmarks. Each graph in Figure 3.15(a) reports C_{core} when scaling from 2 cores to 256 cores for a particular benchmark at 4 different problem sizes. One result from Figure 3.15(a) is that C_{core} indeed increases with core count scaling at each problem size. Table 3.1 reports C_{core} for 256 cores. As this data shows, C_{core} varies between 131.0KB and 13.2MB. On average, C_{core} is between 529.7KB and 6.1MB for different problem sizes. These results show that the impact of core count scaling is confined to smaller shared cache sizes, usually $<$ 16MB. The larger shared cache sizes beyond C_{core} will not experience significant cache-miss increases due to core count scaling. C_{core} is particularly small for LU, KMeans, and BlackScholes, never exceeding 746.8KB. The working sets for

these benchmarks are extremely small and fit inside a 1 MB cache size. For programs with such good locality, the profile shift due to core count scaling is minimal. So, core count scaling never significantly impacts the cache misses of reasonable shared-cache sizes in these programs.

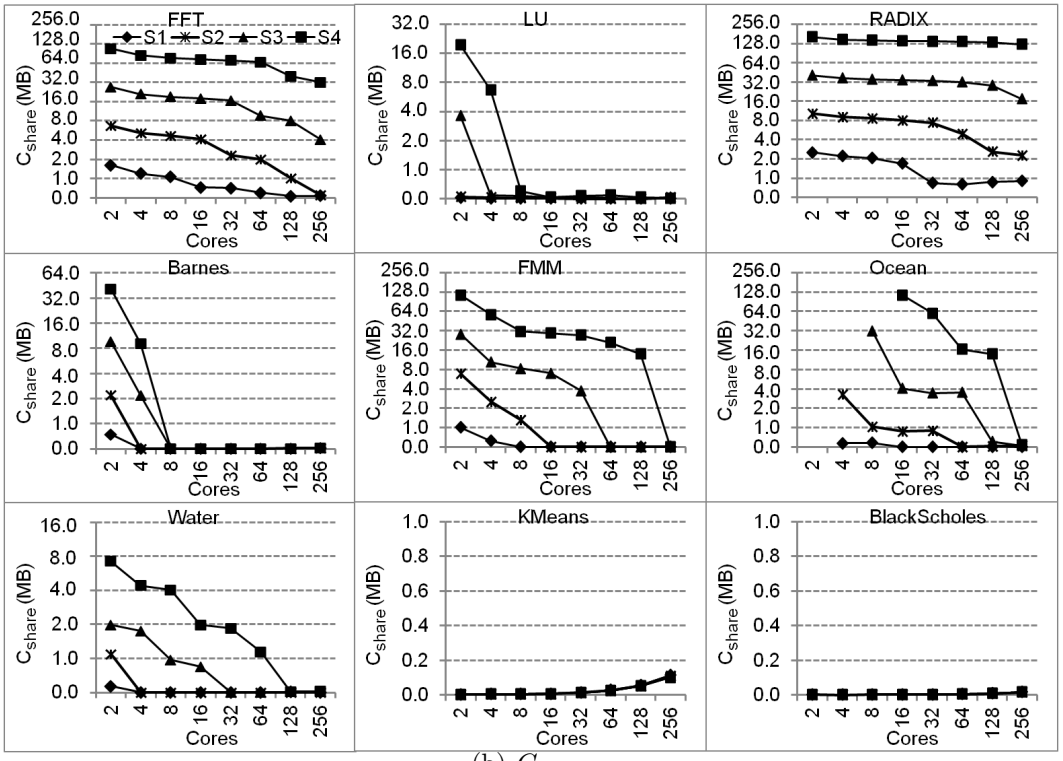
Table 3.1 reports ΔM_a and ΔM_m. Results are only presented for cases where $C_{core} >$ 1MB. As Table 3.1 shows, ΔM_a varies between 1.2 and 5.1, while ΔM_m varies between 1.7 and 8.8. On average, ΔM_a (ΔM_m) is between 2.5 (3.2) and 3.4 (4.5) across different problem sizes. These results show core count scaling can increase cache misses significantly for cache sizes below C_{core}.

Figure 3.15(b) reports C_{share} for 2 to 256 cores on the S1 to S4 problem sizes. The result confirms that core count scaling reduces C_{share} in general. Although in some benchmarks, we see that core count scaling increases C_{share}, these cases happen when C_{share} is very small, below 128KB. Table 3.1 reports C_{share} for 256 cores. As this data shows, C_{share} varies between 0.4KB and 122.9MB. On average, C_{share} is between 133.0KB and 17.3MB for different problem sizes. This result shows, for different benchmarks, that the impact of data sharing begins at different cache capacities. As a result, each benchmark has different sharing characteristics which impacts the multicore cache hierarchy optimization. A detailed discussion of this is in Chapter 6.

Lastly, Figure 3.16 reports CRD_{max} and $sPRD_{max}$ across our AADS. CRD_{max} is roughly constant across 2-256 cores. On the other hand, $sPRD_{max}$ grows as core count scales. Most importantly, $sPRD_{max}$ is always larger than CRD_{max}. This confirms that the overlap effect in CRD profiles causes more contraction than



(a) C_{core}

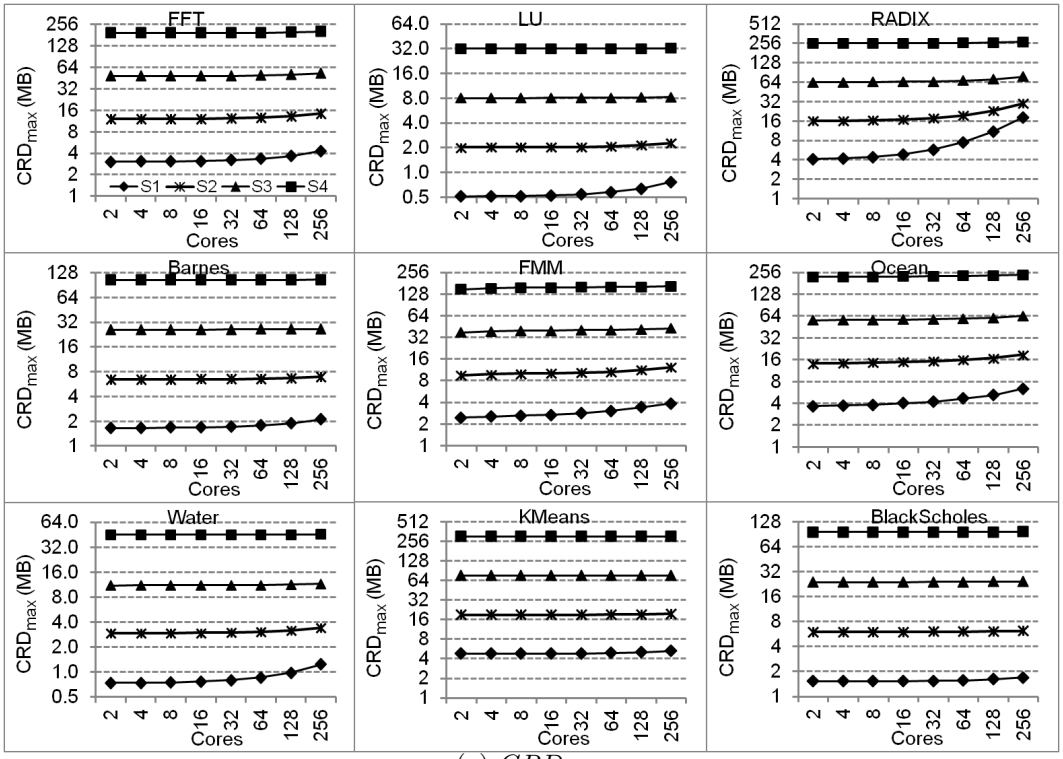


(b) C_{share}

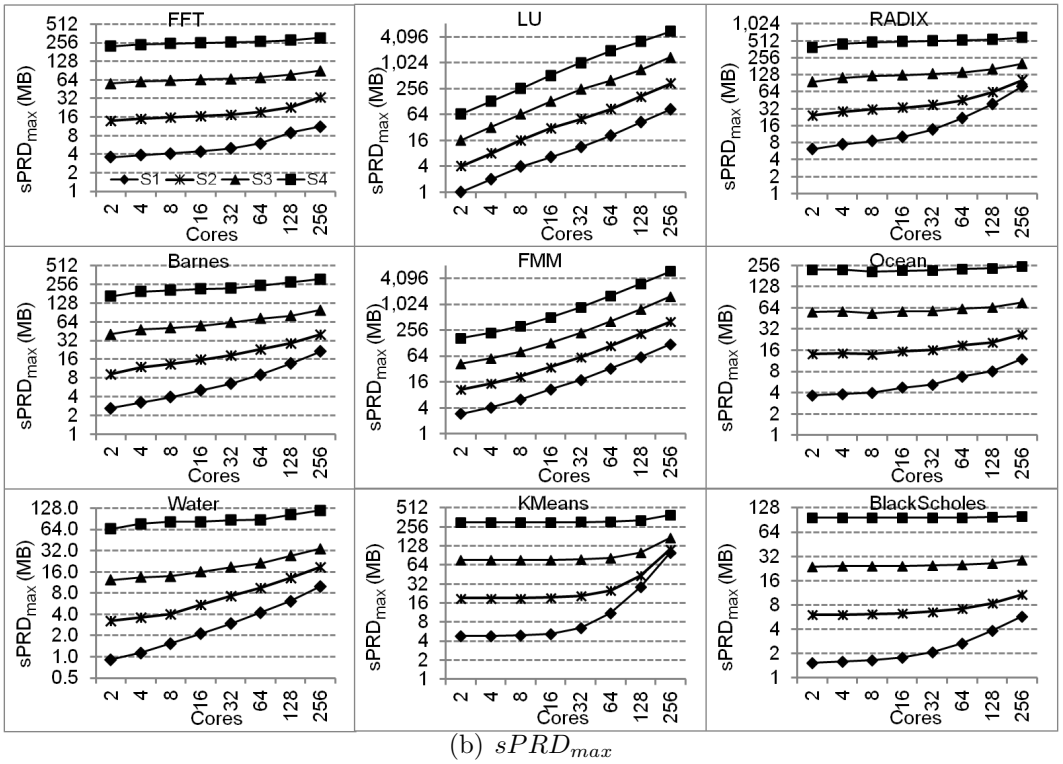
Figure 3.15: C_{core} and C_{share} across core counts and problem sizes.

Table 3.1: C_{core}, C_{share}, CRD_{max}, $sPRD_{max}$, ΔM_a, and ΔM_m for our benchmarks.

Benchmark	S1	S2	S3	S4	S1	S2	S3	S4
	C_{core}				C_{share}			
FFT	1.0MB	2.2MB	5.0MB	10.5MB	108.2KB	168.1KB	4.1MB	32.1MB
LU	131.0KB	207.8KB	374.1KB	746.8KB	0.4KB	53.1KB	51.0KB	22.8KB
RADIX	-	4.2MB	10.5MB	13.2MB	881.3KB	2.4MB	17.9MB	122.9MB
Barnes	573.5KB	1.8MB	3.6MB	5.9MB	32.7KB	33.9KB	34.9KB	35.2KB
FMM	715.9KB	1.3MB	7.1MB	8.7MB	18.2KB	22.2KB	22.6KB	22.7KB
Ocean	715.6KB	1.5MB	4.8MB	13.0MB	15.9KB	72.6KB	86.7KB	128.0KB
Water	283.0KB	592.3KB	1.5MB	2.5MB	8.3KB	8.9KB	19.3KB	36.3KB
KMeans	495.9KB	495.8KB	494.9KB	489.2KB	115.6KB	112.6KB	104.5KB	102.9KB
BlackS.	266.2KB	266.6KB	266.8KB	266.8KB	16.0KB	16.0KB	16.0KB	16.0KB
Average	529.7KB	1.4MB	3.7MB	6.1MB	133.0KB	326.5KB	2.5MB	17.3MB
Benchmark	S1	S2	S3	S4	S1	S2	S3	S4
	CRD_{max}				$sPRD_{max}$			
FFT	4.3MB	14.3MB	52.3MB	200.3MB	11.2MB	33.9MB	90.1MB	305.9MB
LU	785.8KB	2.3MB	8.3MB	32.4MB	82.8MB	335.0MB	1.3GB	5.4GB
RADIX	18.3MB	30.3MB	78.3MB	270.3MB	77.8MB	102.2MB	198.2MB	580.9MB
Barnes	2.1MB	6.9MB	26.5MB	105.3MB	21.3MB	40.3MB	98.7MB	309.2MB
FMM	3.9MB	12.2MB	42.7MB	163.0MB	121.4MB	408.3MB	1.5GB	5.8GB
Ocean	6.4MB	18.7MB	63.9MB	237.2MB	11.9MB	26.9MB	75.6MB	249.6MB
Water	1.2MB	3.4MB	11.5MB	45.5MB	9.9MB	18.7MB	34.2MB	119.2MB
KMeans	5.3MB	19.5MB	76.5MB	304.5MB	99.6MB	113.9MB	170.9MB	398.9MB
BlackS.	1.7MB	6.2MB	24.2MB	96.2MB	5.8MB	10.8MB	28.7MB	100.7MB
Average	4.9MB	12.6MB	42.7MB	161.6MB	49.1MB	121.1MB	397.4MB	1.5GB
	ΔM_a				ΔM_m			
FFT	3.0	3.4	3.3	3.5	3.2	3.7	4.0	4.4
LU	-	-	-	-	-	-	-	-
RADIX	-	5.1	3.0	2.6	-	7.2	6.0	5.6
Barnes	-	3.3	3.6	3.7	-	5.2	7.3	8.8
FMM	-	2.3	2.0	2.1	-	2.6	2.8	3.0
Ocean	-	2.8	1.5	1.2	-	3.6	1.9	1.7
Water	-	-	1.7	1.8	-	-	2.0	2.1
KMeans	-	-	-	-	-	-	-	-
BlackS.	-	-	-	-	-	-	-	-
Average	3.0	3.4	2.5	2.5	3.2	4.5	4.0	4.3



(a) CRD_{max}



(b) $sPRD_{max}$

Figure 3.16: CRD_{max} and $sPRD_{max}$ across core counts and problem sizes.

the demotion absorption effect in sPRD profiles. Table 3.1 reports CRD_{max} and $sPRD_{max}$ for 256 cores. In LU and FMM, $sPRD_{max}$ can reach as high as 5.4GB and 5.8GB.

3.5.2 Problem Size Scaling

As described in Section 3.4, CRD and sPRD profiles shift to larger RD values with problem size scaling due to the increased memory footprint. Figure 3.16 reports CRD_{max} and $sPRD_{max}$ at different problem sizes. The results show CRD_{max} indeed increases by roughly 4x with each problem size increment–*i.e.*, linearly with problem size. $sPRD_{max}$ increases at a sub-linear rate at large core counts. Most importantly, $sPRD_{max}$ is always larger than CRD_{max}. Table 3.1 reports CRD_{max} and $sPRD_{max}$ for each benchmark and problem size on 256 cores. CRD_{max} varies between 785.8KB and 304.5MB. On average, CRD_{max} is between 4.9MB and 161.6MB for different problem sizes. $sPRD_{max}$ varies between 5.8MB and 5.8GB. On average, $sPRD_{max}$ is between 49.1MB and 1.5GB for different problem sizes. This result confirms that data locality degradation affects private caches across a large range of cache capacities.

As Figure 3.15 shows, C_{core} and C_{share} generally increase with problem size scaling. For benchmarks which have very good locality (*i.e.*, KMeans and BlackScholes), problem size scaling has little impact on C_{core} and C_{share}. Table 3.1 reports C_{core} and C_{share} for each benchmark and problem size on 256 cores. On average, C_{core} increases from 529.7KB to 6.1MB and C_{share} increases from 133.0KB to 17.3MB as

problem size scales from S1 to S4. Table 3.1 shows C_{core} increases at a sub-linear rate, roughly as the square root of problem size for our benchmarks. In contrast, C_{share} may increase at a super-linear rate. Hence, for a fixed cache capacity, problem size scaling may reduce the benefit of using shared caches.

3.5.3 Core-Problem Scaling

When core count and problem size scale together, the shifting region associated with core count scaling will itself shift to larger RD values due to problem size scaling. Hence, continued problem size scaling beyond S4 would increase C_{core} beyond the 746.8KB–13.2MB in Table 3.1. Assuming the same rate of increase at larger problems, we see that another 64x increase in problem size would cause C_{core} in many of our benchmarks to grow to 64–128MB. With modest increases in problem size, core count scaling will impact much larger cache capacities, not just those below 16MB.

Although core count scaling reduces C_{share}, problem size scaling increases C_{share}. In general, the combined effect causes C_{share} to increase when we scale core count and problem size together, as illustrated in Figure 3.15(b). As a result, weak scaling may reduce the benefit of using shared caches in multicore processors, and we confirm this in Chapter 6.

Chapter 4

Multicore Reuse Distance Profile Prediction

In Chapter 3, the coherent movement in CRD and PRD profiles suggests the predictability of profiles. This chapter studies techniques for predicting CRD and PRD profiles across core count scaling, problem size scaling, and core-problem scaling. First, we describe our techniques and introduce the evaluation methodology. Then, we present results.

4.1 Prediction Techniques

Section 3.3 shows that CRD_P and CRD_S profiles change differently across core count scaling, so we predict them separately. Based on our insights, we employ two techniques. We use reference groups [20] for CRD_P profile prediction, and we employ a uniform spread model for CRD_S profiles prediction. Section 3.3 shows demotion absorption causes the same effect on PRD_P and PRD_S profiles, so we use reference groups to predict the coherent shift in both PRD_P and PRD_S profiles.

In Section3.4, we find that problem size scaling also causes CRD and PRD profiles to shift coherently. We employ the same technique (reference groups) to predict CRD and PRD profiles at different problem sizes.

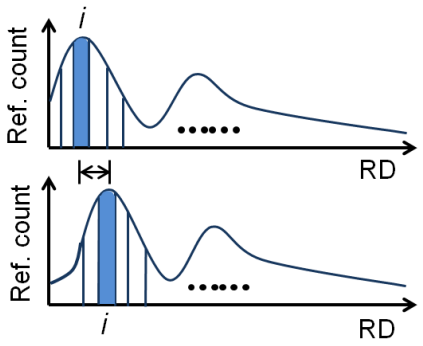


Figure 4.1: Detecting alignment and shifting using reference groups.

4.1.1 Coherent Shift

For sequential programs, Zhong *et al* [20] found that RD profiles exhibit coherent shift due to problem size scaling. They proposed *reference groups* to predict the coherent shift at different problem sizes. We extend their technique to predict CRD and PRD profiles across core count scaling.

Figure 4.1 illustrates Zhong's technique. Zhong divides RD profiles into groups along the RD axis, with each group containing an equal fraction of the program's total references. Reference groups are *aligned* via association: the i^{th} group in the first profile is aligned with the i^{th} group in the second profile. Aligned reference groups are assumed to shift together with their own shifting rates. Zhong's technique employs the pattern function, $p(x) = x^k$, and allows 5 shift rates: constant ($k = 0$), cube root ($k = \frac{1}{3}$), square root ($k = \frac{1}{2}$), cube-root squared ($k = \frac{2}{3}$), and linear ($k = 1$). The group shift rate cannot be greater than linear because reuse distance cannot increase by more than the number of unique memory references, which is proportional to problem size.

For each pair of reference groups, i, the shift is measured and compared against

each allowed shift rate. Let S1 and S2 be the problem sizes of first and second RD profiles. d_{1i} and d_{2i} are the average RD values at reference group i of first and second RD profiles. The shift rate with the closest match (*i.e.*, there exists a $p(x)$ such that $\frac{p(S1)}{p(S2)}$ is closest to $\frac{d_{1i}}{d_{2i}}$) is assigned to the reference group. After solving Equation 4.1 for each reference group, we can predict RD profiles for different problem sizes. Each reference group is shifted by its shift rate and desired problem scaling factor.

$$\begin{aligned} d_{1i} &= c_i + e_i \times p_i(S_1) \\ d_{2i} &= c_i + e_i \times p_i(S_2) \end{aligned} \tag{4.1}$$

We apply Zhong's technique to predict core count scaling as follows. To predict CRD profiles (either CRD_P or CRD_S), we use the 2- and 4-core CRD profiles as samples to predict the CRD profiles at the remaining core counts. These measured profiles are divided into 200,000 groups, each containing an equal fraction (0.0005%) of the profile's references. While Zhong originally divided each profile into 1,000 reference groups, we find the increased resolution provides better accuracy for core count scaling. We detect the inter-group shift as discussed above, but instead of multiplying this shift rate by the problem scaling factor, we multiply it by the core count scaling factor. We also increase the granularity of the pattern function (Equation 4.2). We use reference groups to predict CRD_P profiles at larger core count from measured CRD_P profiles. We also use reference groups to predict CRD_S profiles. We call the predicted profiles CRD_{Sshift}. Then we combine the CRD_{Sshift} profile with spread prediction in the next section to derive CRD_S profiles.

$$p(x) = x^k, k = 0, 0.01, 0.02, ..., 0.99, 1.00 \tag{4.2}$$

For PRD profiles, we use the same technique to predict profiles at larger core counts from the 2- and 4-core PRD profiles. However, because PRD profiles shift to smaller RD value as core count scales, we need to change the pattern function to support this truncation behavior. We use Equation 4.3 as the new pattern function to predict the coherent shift in PRD profiles.

$$p(x) = \frac{1}{x^k}, k = 0, 0.01, 0.02, ..., 0.99, 1.00 \tag{4.3}$$

For problem size scaling, we employ the same technique and use the patter function in Equation 4.2 to predict both CRD and PRD profiles. We also divide each profile into 200,000 reference groups.

4.1.2 Spread

Section 3.3.2 shows that intercepts spread CRD_S profiles, with individual reuses moving to CRD values between 0 and $P \times RD$, where P is the core count. Although the actual distribution within this range is application dependent, we make the simplifying assumption that references are spread *uniformly* across the range. To predict spread, we sample the CRD_S profile at 4 cores (the same sample used in shift prediction), and uniformly distribute the reference counts ($CRD_{S\_4core}[k] \times Pd[k]$)

at each CRD between 0 and $min(k \times \frac{core\_count}{4}, C_{max})$, where k is a particular CRD value, and $Pd[k] = \frac{k}{C_{max}}$ (C_{max} is the CRD profile's maximum CRD value). We call this prediction $\text{CRD}_{Sspread}$. Then, we predict the CRD_S profile as follows:

$$CRD_S[k] = (1 - Pd[k]) \times CRD_{Sshift}[k] + CRD_{Sspread}[k]$$

This predicts CRD_S by averaging CRD_{Sshift} and $\text{CRD}_{Sspread}$, weighting the former more heavily at small CRD values (where intercepts happen rarely) and the latter more heavily at large CRD values (where intercepts happen often).

4.2 Prediction Methodology

Machine scaling defines a design space consisting of multicore processors with varying core counts and cache capacities. When processors scale to the LCMP level, they will also execute larger problems. So, it is very important to understand the impact of problem size scaling. Our work also considers problem size as an independent parameter that can be varied as well. Figure 2.5 illustrates our architecture-application design space (AADS). In our study, we acquire CRD and PRD profiles at every core count and problem size as illustrated in Figure 4.2. For each benchmark, we have 32 configurations, and we acquire CRD and PRD profiles for each configuration. By comparing measured and predicted profiles along any axis, we can compute the prediction accuracy to the corresponding type of scaling. In this study, we employ two metrics, $\text{RD}_{Accuracy}$ and $\text{RD\_CMC}_{Accuracy}$ to quantify the prediction accuracy.

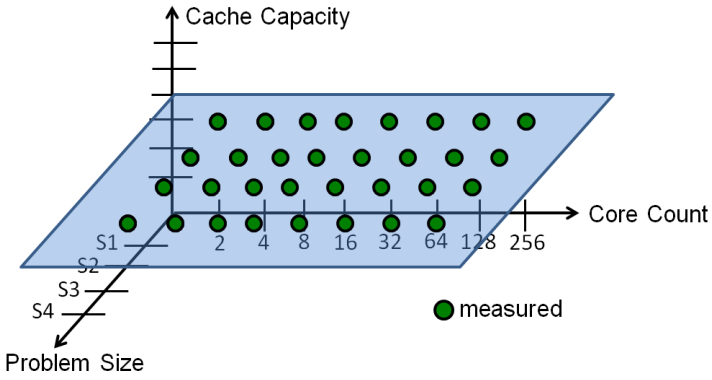
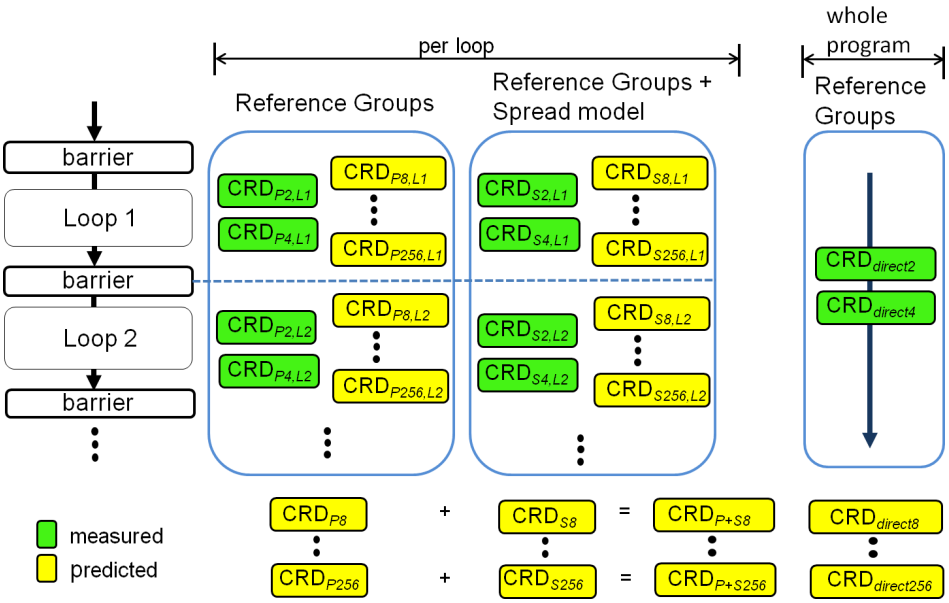


Figure 4.2: Design space across 8 core counts and 4 problem sizes.

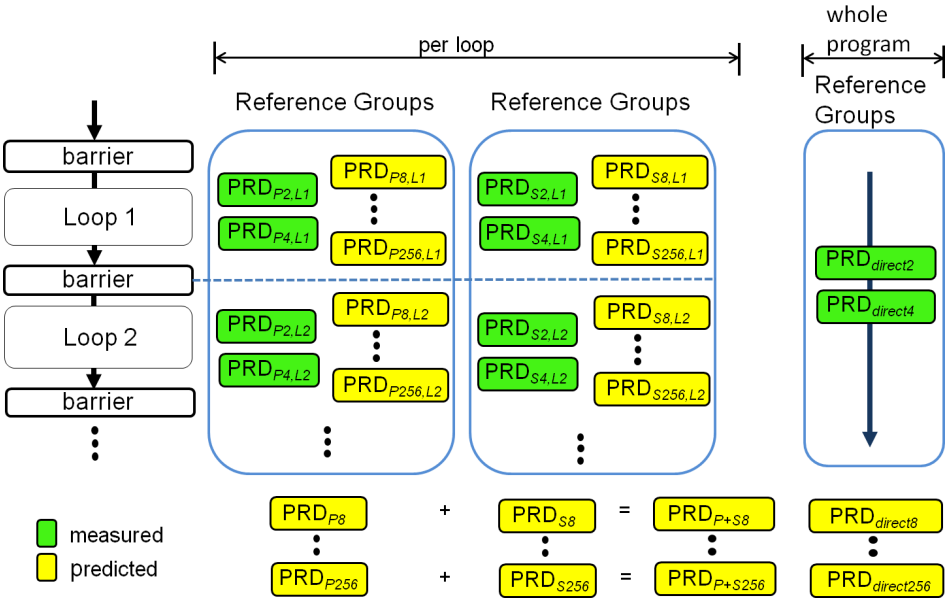
4.2.1 Acquiring Profiles

We use the in-house built Pin tool to acquire CRD and PRD profiles for our study. Figure 4.3(a) shows how we acquire and predict CRD profiles. First, we acquire the per-parallel region CRD_P and CRD_S profiles, as described in Section 3.1.1, for 2- and 4-core executions. Then, at each parallel region, we use the techniques from Section 4.1 to predict the CRD_P and CRD_S profiles for 8–256 cores from the 2- and 4-core samples. For CRD_P, we use reference groups to predict the coherent movement. For CRD_S, we further include the spread model to predict intercept effects. After the program finishes, we sum up all the per-loop predicted profiles to get program-wide CRD_P and CRD_S profiles at different core counts. Then we sum CRD_P and CRD_S profiles together to get the whole-program CRD profile. We call this CRD_{P+S}.

In our benchmarks, CRD_P profiles dominate CRD_S profiles. This implies that predicting coherent shift alone may be sufficient in many cases. In addition to predicting CRD_P and CRD_S profiles separately, we also employ whole-program CRD profile prediction. We use Pin to acquire the whole program CRD profiles



(a) Acquiring and predicting CRD_P, CRD_S, CRD_{P+S}, and CRD_{direct} profiles.



(b) Acquiring and predicting PRD_P, PRD_S, PRD_{P+S}, and PRD_{direct} profiles.

Figure 4.3: Acquiring and predicting profiles.

at 2- and 4- cores. Then, we use reference groups to predict the whole-program profiles for 8–256 cores directly from the measured whole-program profiles. We call these profiles CRD_{direct}. The advantage of this approach is that it does not require profiling individual parallel regions.

Figure 4.3(b) shows how we acquire and predict PRD profiles. As described in Section 3.3, demotion absorption causes the same impact on PRD_P and PRD_S profiles. At each parallel region, we use reference groups to predict the PRD_P and PRD_S profiles for 8–256 cores from the 2- and 4-core samples. After the program finishes, we sum up all the per-loop predicted profiles to get program-wide PRD_P and PRD_S profiles at different core counts. Then we sum PRD_P and PRD_S together to get the whole-program PRD profile, PRD_{P+S}. We also acquire the whole program PRD profiles at 2- and 4- cores, and we use reference groups to predict the whole-program profiles for 8–256 cores directly from the measured whole-program profiles. We call these profiles PRD_{direct}.

4.2.2 Accuracy Metrics

We use two metrics, $\text{RD}_{Accuracy}$ and $\text{RD\_CMC}_{Accuracy}$, to assess prediction accuracy. The first metric is similar to metrics used in previous work [26, 14]. $\text{RD}_{Accuracy}$ is defined in Equation 4.4, where N is the number of bins. $\text{RD}_{Accuracy}$ is $1 - \frac{E}{2}$, where E is the sum of the normalized absolute differences between every pair of reference counts from a predicted and measured RD profile. E can be at most 200%, so $\text{RD}_{Accuracy}$ is between 0–100%. The $\text{RD}_{Accuracy}$ for predicted CRD

and PRD profiles are $\text{CRD}_{Accuracy}$ and $\text{PRD}_{Accuracy}$.

$$RD_{Accuracy} = 1 - \frac{1}{2}\sum_{i=0}^{N-1}\frac{|RD_{measured}[i] - RD_{predicted}[i]|}{total\ references} \tag{4.4}$$

The second metric is $\text{RD\_CMC}_{Accuracy}$. CMC (cache-miss count) accuracy is computed from CMC profiles, which present the number of cache misses predicted by a RD profile at each of its cache capacities. We compute $\text{RD\_CMC}_{Accuracy}$ by averaging the error between pairs of RD values from the entire predicted and measured CMC profiles as specified in Equation 4.5.

$$RD\_CMC_{Accuracy} = 1 - \frac{1}{N}\sum_{i=0}^{N-1}\frac{|RD\_CMC_{measured}[i] - RD\_CMC_{predicted}[i]|}{RD\_CMC_{measured}[i]} \tag{4.5}$$

$\text{RD\_CMC}_{Accuracy}$ reflects cache performance. Because $\text{RD}_{Accuracy}$ is an absolute metric, it more heavily weights error at the first few RD values where reference counts are enormous but which occur well below small cache capacities. In contrast, $\text{RD\_CMC}_{Accuracy}$ equally weights error across CMC profiles. So $\text{RD\_CMC}_{Accuracy}$ can reflect the cache performance more fairly. The $\text{RD\_CMC}_{Accuracy}$ for predicted CRD and PRD profiles are $\text{CRD\_CMC}_{Accuracy}$ and $\text{PRD\_CMC}_{Accuracy}$.

4.3 Prediction Accuracy Results for Core Count Scaling

To evaluate the prediction accuracy of core count scaling for each benchmark and problem size, we use the measured profiles at 2 and 4 cores to predict the profiles for 8–256 cores, yielding predicted profiles for 24 configurations per benchmark.

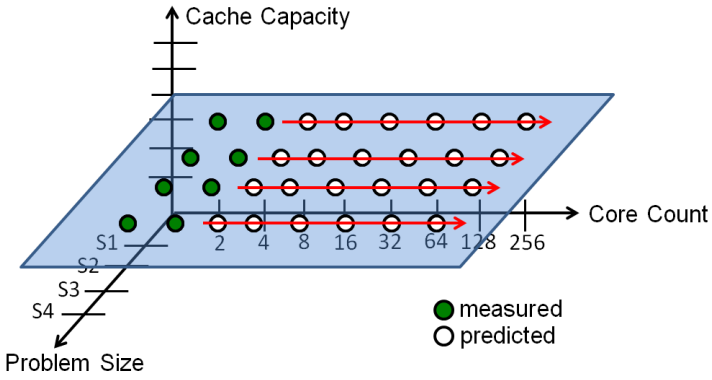


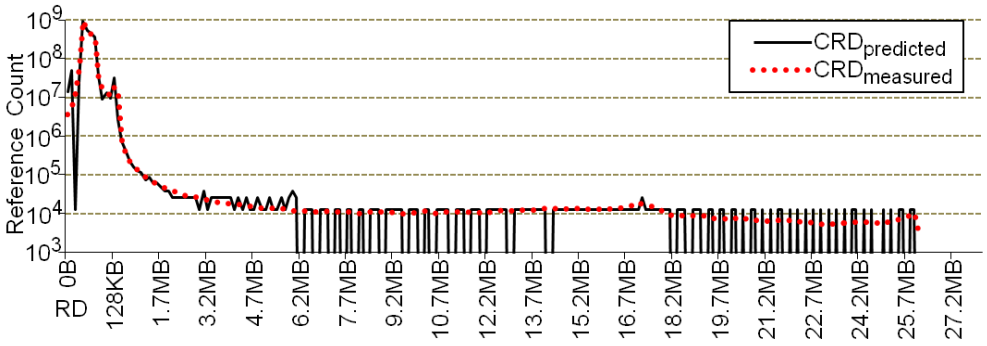
Figure 4.4: Profile prediction for core count scaling.

Figure 4.4 illustrates the measured and predicted points.

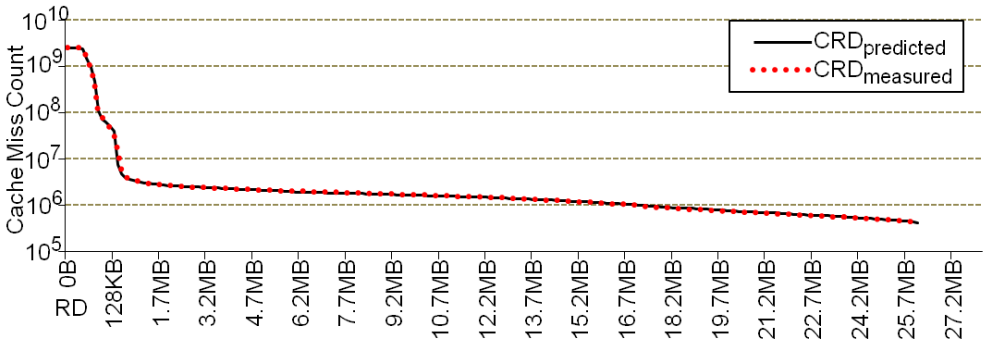
4.3.1 CRD Profiles

Figure 4.5 compares Barnes' measured CRD_{direct} profiles (dotted lines) with predicted CRD_{direct} profiles (solid lines) running on 16 and 64 cores at the S3 problem size. In Figure 4.5(a) and Figure 4.5(c), predicted CRD and measured CRD profiles are very similar. However, the predicted CRD profiles show saw-tooth oscillation at large RD values. This is because a reference group collects the reference counts across several bins into one group when the reference counts are small. After we shift the reference group to the new RD value, these references have the same RD value. So we lose some detailed information. However, reference groups can still capture the major shifting behavior for core count scaling. As a result, in Figure 4.5(a) and Figure 4.5(c), the $\text{CRD}_{Accuracy}$ of 16 cores and 64 cores are 95.0% and 90.7%, respectively.

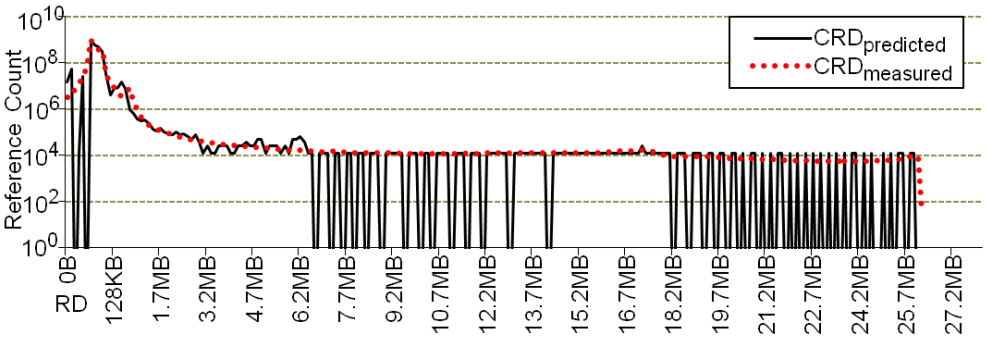
For CMC profiles, we also predict the compulsory misses which have infinite reuse distance. In our prediction, we assume that compulsory misses grow pro-



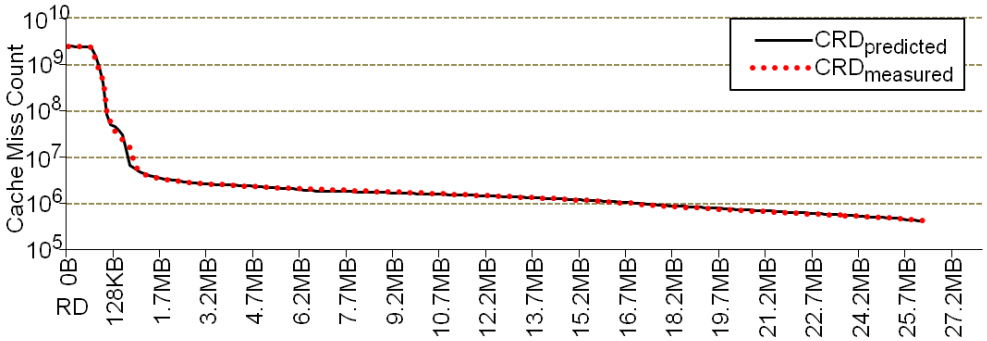
(a) Measured and predicted CRD profiles on 16 cores at the S3 problem size.



(b) Measured and predicted CRD_CMC profiles on 16 cores at the S3 problem size.



(c) Measured and predicted CRD profiles on 64 cores at the S3 problem size.



(d) Measured and predicted CRD_CMC profiles on 64 cores at the S3 problem size.

Figure 4.5: Examples for measured and predicted CRD_{direct} profiles with core count scaling.

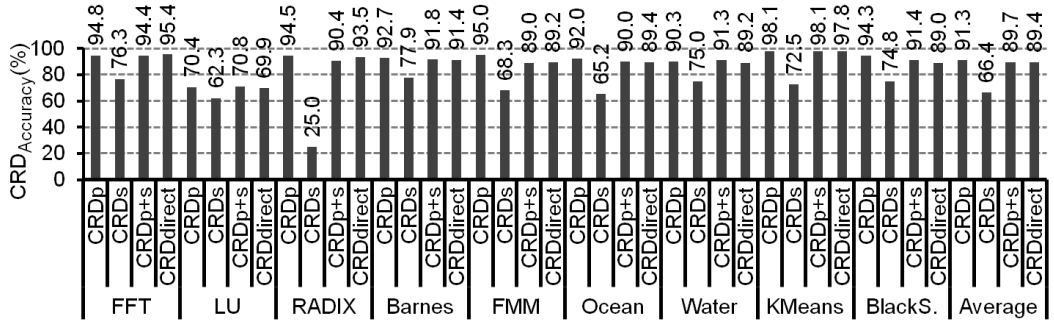
portionally with respect to core count. Figure 4.5(b) and Figure 4.5(d) show the corresponding CRD_CMC profiles running on 16 and 64 cores. Although reference groups lose some detailed information at large RD values in the predicted CRD profile, integration makes this impact insignificant. As a result, the CRD_CMC$_{Accuracy}$ of 16 cores and 64 cores are 97.8% and 96.5%, respectively.

Figure 4.6 presents our full CRD profile prediction results. In Figure 4.6(a), the "CRD$_P$" ("CRD$_S$") bars show results for predicting CRD$_P$ (CRD$_S$) profiles separately. For each benchmark, problem size, and core count, we sum all predicted per-parallel region CRD$_P$ (CRD$_S$) profiles into a single CRD$_P$ (CRD$_S$) profile. Then, we compare this against the measured aggregate CRD$_P$ (CRD$_S$) profile. Each bar in Figure 4.6(a) reports the average CRD$_{Accuracy}$ achieved over the 24 predictions per benchmark. The rightmost bars report the average across all benchmarks.

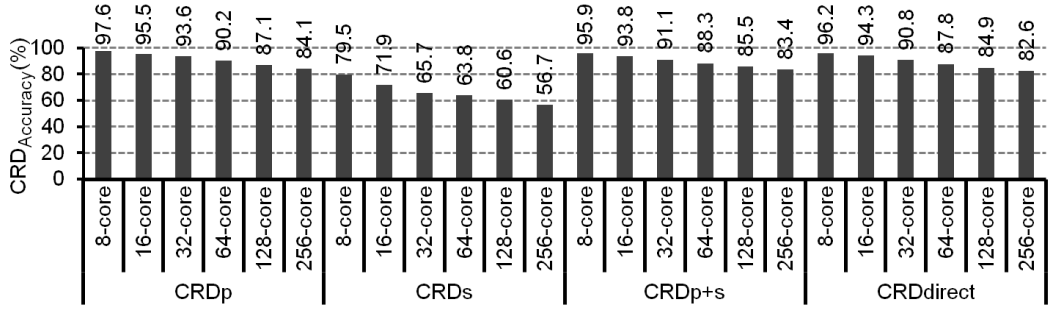
As Figure 4.6(a) shows, CRD$_P$ profiles are predicted with high accuracy. For all benchmarks except LU, CRD$_P$ accuracy is between 90.3% and 98.1%. For LU, CRD$_P$ accuracy is 70.4%. Across all benchmarks, the average CRD$_P$ accuracy is 91.3%. CRD$_P$ profiles exhibit coherent shift across core count scaling which reference groups can effectively predict.

LU is the only benchmark with lower CRD$_P$ accuracy. In LU, blocking is performed to improve cache locality, but for S1 and S2 problem sizes, the default blocking factor does not create enough parallelism to keep more than 32 cores busy. This introduces error when predicting large core counts.

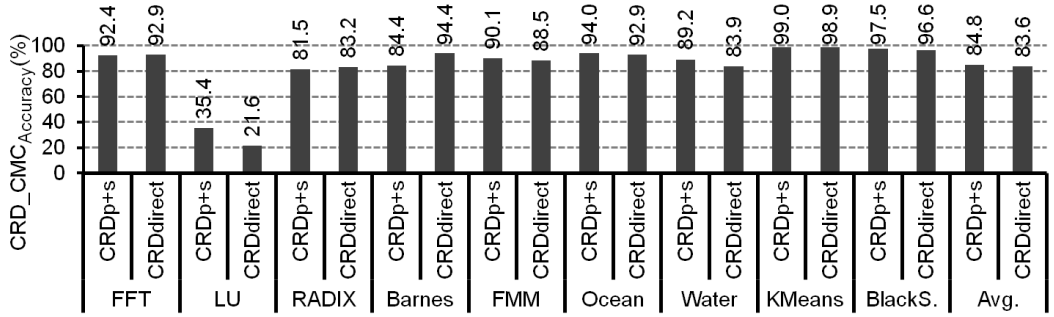
Compared to CRD$_P$ profiles, CRD$_S$ profiles are predicted with lower accuracy. In Figure 4.6(a), CRD$_S$ accuracy is between 25.0% and 77.9%. Across all bench-



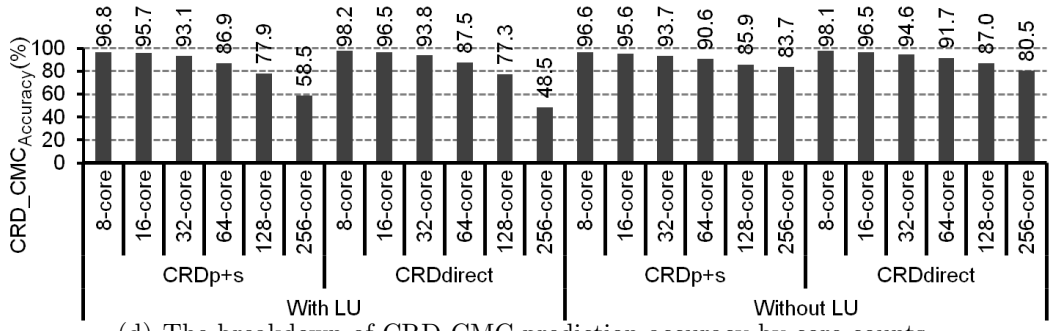
(a) CRD accuracy of predicted CRD_P and CRD_S profiles, and indirectly and directly predicted whole-program CRD profiles.



(b) The breakdown of CRD prediction accuracy by core counts.



(c) CRD_CMC accuracy of indirectly and directly predicted whole-program CRD profiles.



(d) The breakdown of CRD_CMC prediction accuracy by core counts.

Figure 4.6: CRD profile prediction accuracy for core count scaling.

marks, the average CRD_S accuracy is only 66.4%. CRD_S profiles suffer poor spread prediction. While intercepts induce spreading in the range we expect, the actual distribution across this range is highly application dependent. Unfortunately, our simple uniform spread model does not capture general behavior, leading to lower prediction accuracy.

Although CRD_S profiles are predicted with lower accuracy, the impact on overall prediction accuracy is minimal. In Figure 4.6(a), the bars labeled "CRD_{P+S}" report the average $CRD_{Accuracy}$ for whole-program CRD profiles predicted by combining CRD_P and CRD_S predictions. For all benchmarks except LU, CRD_{P+S} accuracy is between 89.0% and 98.1% (for LU, it is 70.8%). The average CRD_{P+S} accuracy for all benchmarks is 89.7%. These results confirm that CRD_P dominates CRD_S. So, predicting CRD_P profiles effectively leads to accurate whole-program CRD profile prediction.

In our benchmarks, there is usually one parallel region that dominates the whole program, so one would expect that predicting whole-program CRD profiles directly to be the same as (and hence, achieve similar accuracy compared to) predicting CRD_{P+S} profiles. This is in fact the case. The last set of bars in Figure 4.6(a), labeled "CRD_{direct}," report the average $CRD_{Accuracy}$ for direct whole-program CRD profiles prediction. Figure 4.6(a) shows CRD_{direct} is very similar to CRD_{P+S}. On average, CRD_{direct} accuracy is 89.4%, compared to 89.7% for CRD_{P+S}. The results in Figure 4.6(a) demonstrate the pervasiveness of coherent shift across our benchmarks, and confirm the accuracy of reference groups for this type of profile movement. Figure 4.6(b) shows the accuracy breakdown by different core counts.

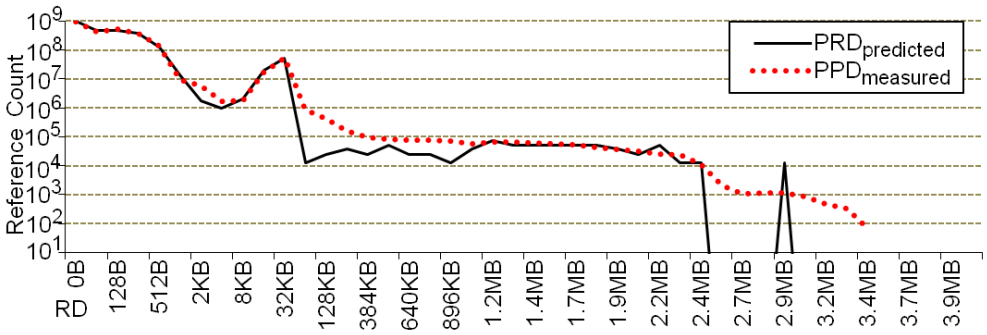
Because we use 2 cores and 4 cores to predict the larger core counts, the prediction accuracy degrades when the predicted point is farther away from the measured points.

Finally, Figure 4.6(c) illustrates the whole-program prediction results using the CRD_CMC$_{Accuracy}$ metric. CRD$_{P+S}$ and CRD$_{direct}$ have similar CRD_CMC$_{Accuracy}$. They are between 81.5%–99.0% for 8 benchmarks, and are roughly 35.4% for LU. On average, CRD$_{P+S}$ and CRD$_{direct}$ achieve a 91.0% and 91.4% accuracy, respectively, without LU, and 84.8% and 83.6% accuracy, respectively, for all benchmarks. This result suggests that our predicted CRD profiles can provide good cache-miss predictions. Figure 4.6(d) breaks down CRD_CMC$_{Accuracy}$ at different core counts. Again, the larger error happens at large core counts.

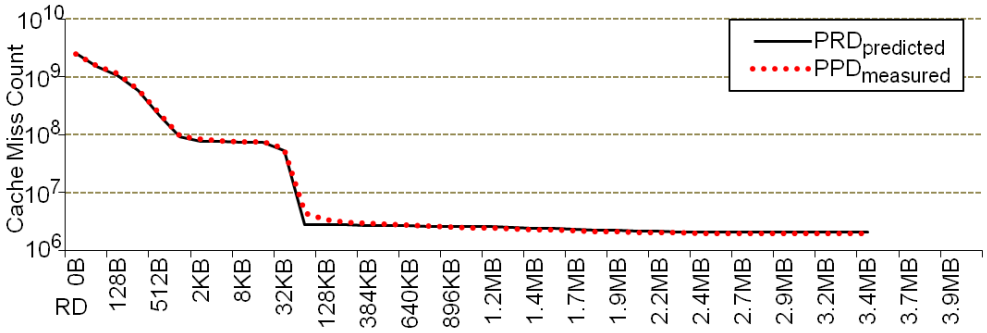
4.3.2 PRD Profiles

Figure 4.7 compares Barnes' measured PRD$_{direct}$ profile (dotted lines) with predicted PRD$_{direct}$ profile (solid lines) running on 16 and 64 cores at the S3 problem size. In Figure 4.7(a) and Figure 4.7(c), predicted PRD profiles capture the major shifting behavior for core count scaling, but predicted PRD profiles also lose some detailed information at large RD values due to insufficient resolution. Overall, predicted PRD profiles and measured PRD profiles have very similar shapes. As a result, in Figure 4.7(a) and Figure 4.7(c), the PRD$_{Accuracy}$ of 16 cores and 64 cores are 95.6% and 95.0%, respectively.

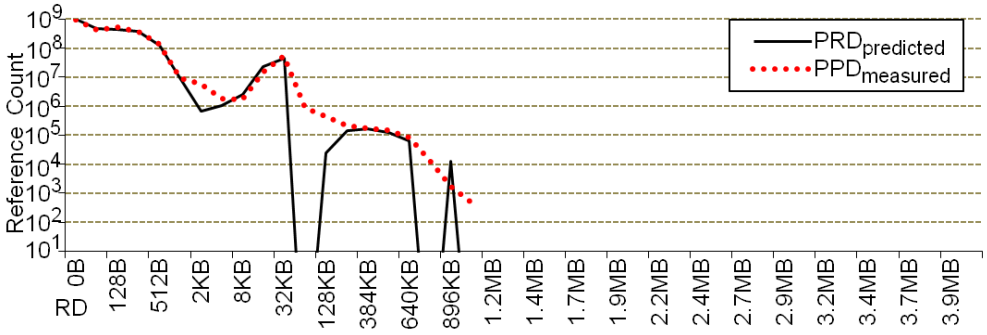
For CMC profiles, we also predict the compulsory misses and coherence misses,



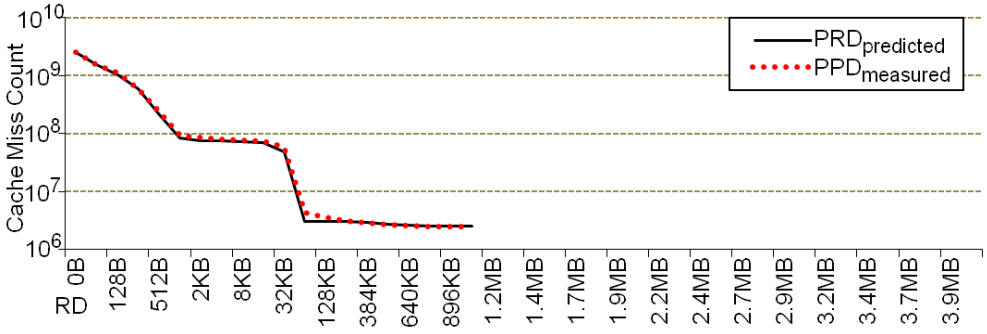
(a) Measured and predicted PRD profiles on 16 cores at the S3 problem size.



(b) Measured and predicted PRD_CMC profiles on 16 cores at the S3 problem size.



(c) Measured and predicted PRD profiles on 64 cores at the S3 problem size.



(d) Measured and predicted PRD_CMC profiles on 64 cores at the S3 problem size.

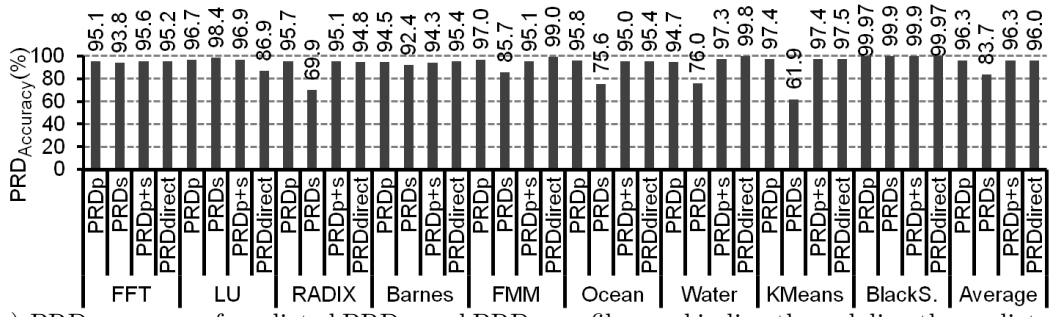
Figure 4.7: Examples for measured and predicted PRD$_{direct}$ profiles with core count scaling.

which have infinite reuse distance. We assume that these cache misses grow proportionally with respect to core count. Figure 4.7(b) and Figure 4.7(d) show the corresponding PRD_CMC profiles at 16 and 64 cores. Although reference groups lose some detailed information at some RD values in predicted PRD profiles, integration makes this impact insignificant. As a result, the PRD_CMC$_{Accuracy}$ of 16 cores and 64 cores are 94.1% and 93.6%, respectively.

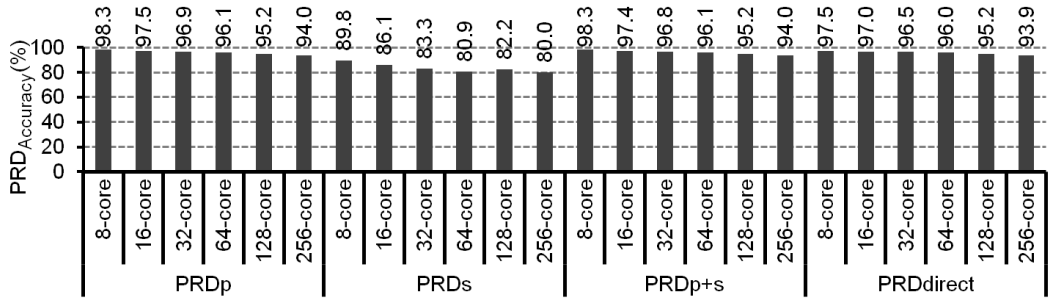
Figure 4.8 presents our PRD profile prediction results under core count scaling. In Figure 4.8(a), each benchmark has four bars, "PRD$_P$", "PRD$_S$", "PRD$_{P+S}$", and "PRD$_{direct}$". Each bar reports the average PRD$_{Accuracy}$ achieved over the 24 predictions per benchmark. The rightmost bars report the average accuracy across all benchmarks.

As Figure 4.8(a) shows, PRD$_P$ profiles are predicted with high accuracy, between 94.5% and 99.97%. Across all benchmarks, the average PRD$_P$ accuracy is 96.3%. PRD$_P$ profiles exhibit coherent shift across core count scaling, which reference groups can effectively predict. Compared to PRD$_P$ profiles, PRD$_S$ profiles are predicted with lower accuracy. In Figure 4.8(a), PRD$_S$ accuracy is between 61.9% and 99.9%. Across all benchmarks, the average PRD$_S$ accuracy is only 83.7%.

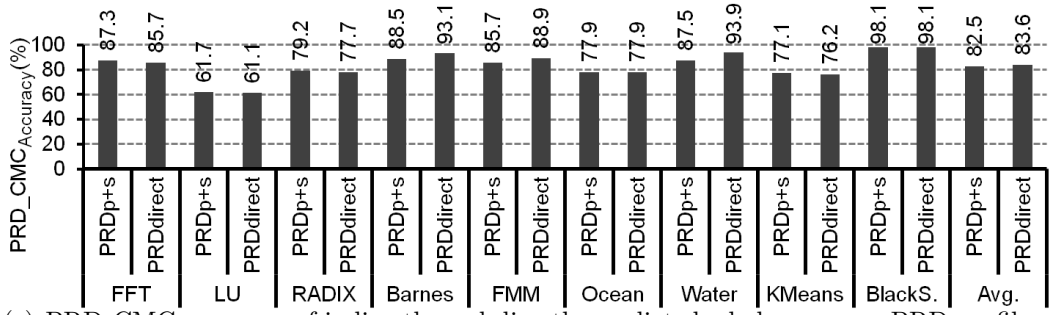
Although PRD$_S$ profiles are predicted with lower accuracy, the impact on overall prediction accuracy is minimal. In Figure 4.8(a), the bars labeled "PRD$_{P+S}$" illustrate the average PRD accuracy for whole-program PRD profiles predicted by combining PRD$_P$ and PRD$_S$ predictions. For all benchmarks, PRD$_{P+S}$ accuracy is between 94.3% and 99.9%. The average PRD$_{P+S}$ accuracy for all benchmarks is 96.3%. These results confirm that PRD$_P$ profiles dominate PRD$_S$ profiles.



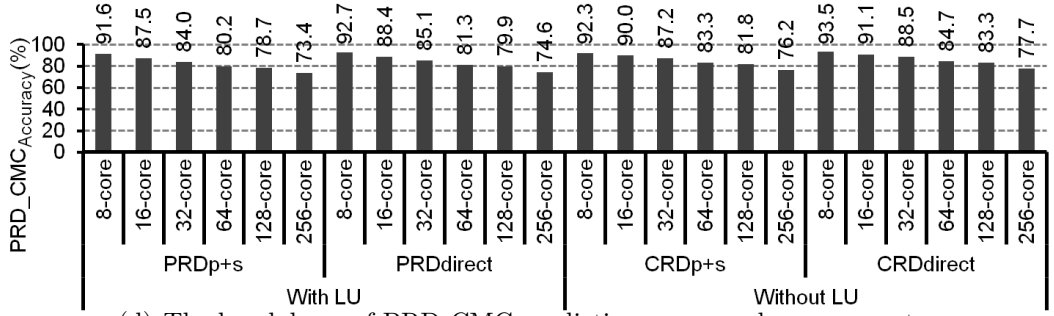
(a) PRD accuracy of predicted PRD_P and PRD_S profiles, and indirectly and directly predicted whole-program PRD profiles.



(b) The breakdown of PRD prediction accuracy by core counts.



(c) PRD_CMC accuracy of indirectly and directly predicted whole-program PRD profiles.



(d) The breakdown of PRD_CMC prediction accuracy by core counts.

Figure 4.8: PRD profile prediction accuracy for core count scaling.

The last set of bars in Figure 4.8(a), labeled "PRD_{direct}," report the average $\text{PRD}_{Accuracy}$ for direct whole-program PRD profile prediction. Figure 4.8(a) shows that PRD_{direct} is very similar to PRD_{P+S}. On average, PRD_{direct} accuracy is 96.0%, compared to 96.3% for PRD_{P+S}. The results in Figure 4.8(a) demonstrate the pervasiveness of coherent shift across our benchmarks, and confirm the accuracy of reference groups for this type of profile movement. Figure 4.8(b) shows the accuracy breakdown by different core counts. Because we use 2 cores and 4 cores to predict the larger core counts, the prediction accuracy degrades as the predicted point is farther away from the measured points.

Finally, Figure 4.8(c) illustrates the whole-program prediction results using the $\text{PRD\_CMC}_{Accuracy}$ metric. PRD_{P+S} and PRD_{direct} have similar $\text{PRD\_CMC}_{Accuracy}$, between 76.2%–98.1% for 8 benchmarks, and are roughly 61.1% for LU. On average, PRD_{P+S} and PRD_{direct} achieve a 85.2% and 86.5% accuracy, respectively, without LU, and 82.5% and 83.6% accuracy, respectively, for all benchmarks. This result suggests that our predicted PRD profiles can also provide good cache-miss predictions. Figure 4.8(d) breaks down $\text{PRD\_CMC}_{Accuracy}$ by different core counts. Again, the larger error happens at large core counts.

Comparing the prediction accuracy of CRD and PRD profiles, our prediction techniques can predict CRD and PRD profiles with similar accuracy for core count scaling. However, at large core counts, $\text{PRD\_CMC}_{Accuracy}$ is less accurate than $\text{CRD\_CMC}_{Accuracy}$. This is because we also need to predict compulsory misses and coherence misses for PRD profiles, and this induces higher error.

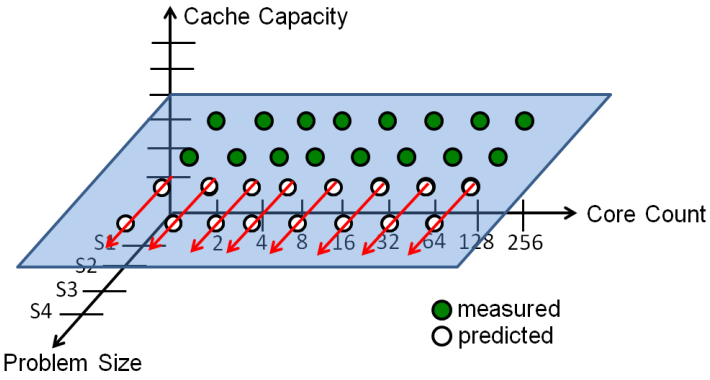


Figure 4.9: Profile prediction for problem size scaling.

4.4 Prediction Accuracy Results for Problem Size Scaling

To study the prediction accuracy of problem size scaling for each benchmark and core count, we use the measured profiles at the S1 and S2 problem sizes to predict the profiles for the S3 and S4 problem sizes, yielding predicted profiles for 16 configurations per benchmark. Figure 4.9 illustrates the measured and predicted points.

4.4.1 CRD Profiles

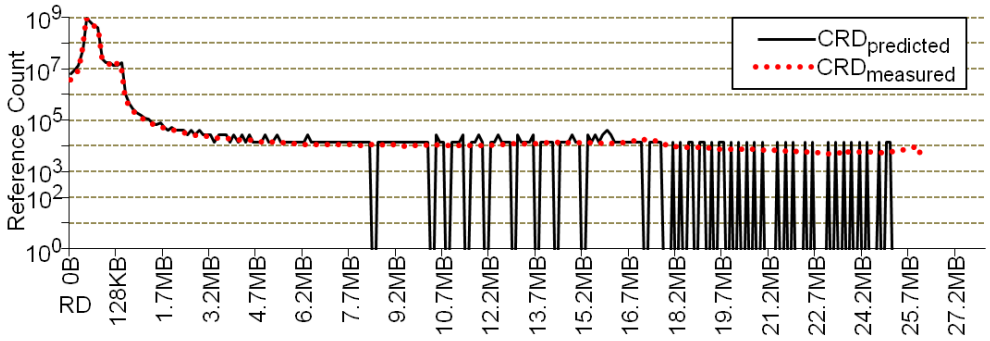
Figure 4.10 compares Barnes' measured CRD_{direct} profiles (dotted lines) with predicted CRD_{direct} profiles (solid lines) running on 16 cores at the S3 and S4 problem sizes. In Figure 4.10(a) and Figure 4.10(c), predicted CRD profiles capture the major shifting behavior for problem size scaling at small RD values. At large RD values, the CRD profile usually has a long tail with small reference counts. Problem size scaling causes the profile shift to larger RD values, and the region which contains small reference counts becomes longer. Because we use a fixed number of

reference groups (200,000), the resolution in the long tail decreases as problem size scales. In predicted profiles, some bins have 0 reference counts, and some bins have higher reference counts than the measured reference counts. However, predicted CRD profiles and measured CRD profiles still have very similar shapes. As a result, in Figure 4.10(a) and Figure 4.10(c), the $\text{CRD}_{Accuracy}$ of 16 cores at the S3 and S4 problem size are 95.2% and 93.1%, respectively.

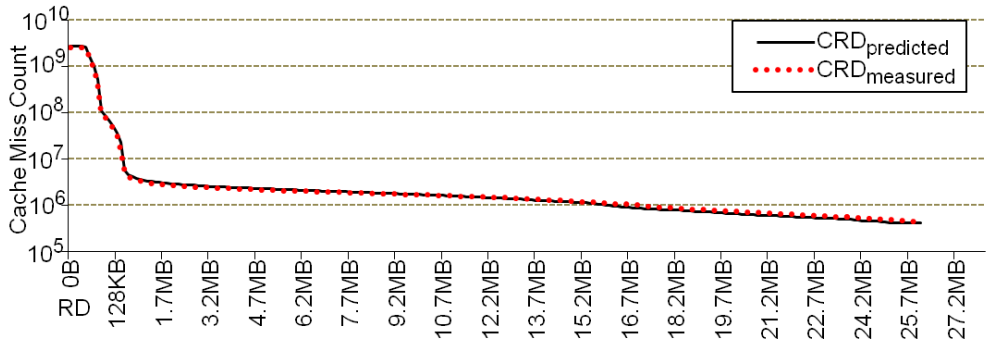
Figure 4.10(b) and Figure 4.10(d) show the corresponding CRD_CMC profiles at the S3 and S4 problem sizes. The resolution of reference groups causes some distortion in CRD profiles, but integration reduces the impact of distortion. We also assume that compulsory misses grow proportionally as problem size scales. As a result, the $\text{CRD\_CMC}_{Accuracy}$ of the S3 and S4 problem sizes is 92.5% and 85.0%, respectively. The predicted and measured profiles are also very similar for problem size scaling.

Figure 4.11 presents our full CRD profile prediction results. Each bar illustrates the average $\text{CRD}_{Accuracy}$ achieved over the 16 predictions per benchmark. The rightmost bars report the average accuracy across all benchmarks. As Figure 4.11(a) shows, CRD_P profiles are predicted with high accuracy, between 79.8% and 99.7%. Across all benchmarks, average CRD_P accuracy is 90.7%. CRD_P profiles exhibit the coherent shift across problem size scaling, which reference groups can effectively predict.

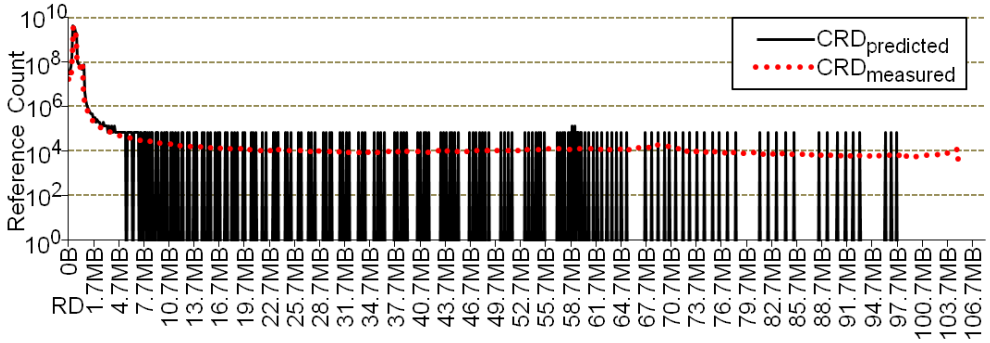
Compared to CRD_P profiles, CRD_S profiles are predicted with lower accuracy. In Figure 4.11(a), CRD_S accuracy is between 72.7% and 94.1%. Across all benchmarks, the average CRD_S accuracy is 85.4%. CRD_S profiles for problem size scaling



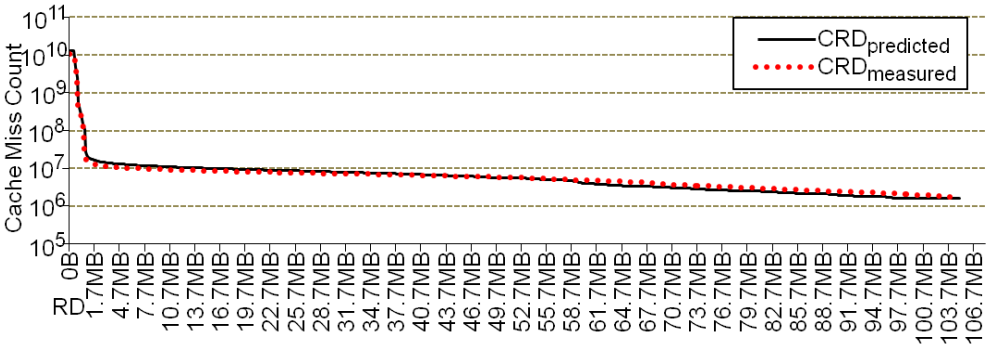
(a) Measured and predicted CRD profiles on 16 cores at the S3 problem size.



(b) Measured and predicted CRD_CMC profiles on 16 cores at the S3 problem size.

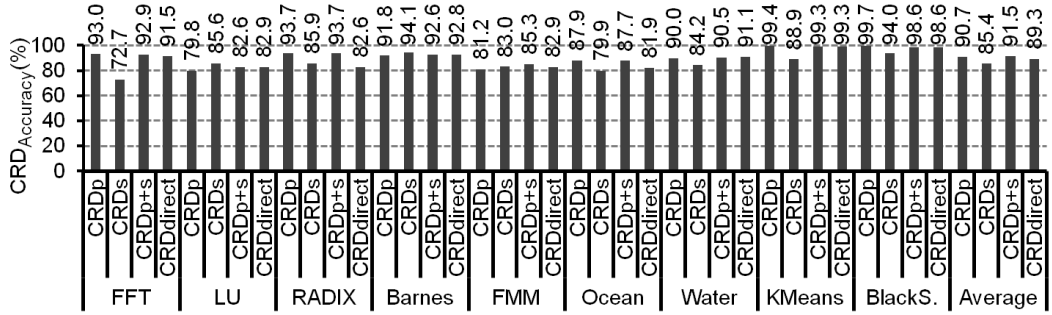


(c) Measured and predicted CRD profiles on 16 cores at the S4 problem size.

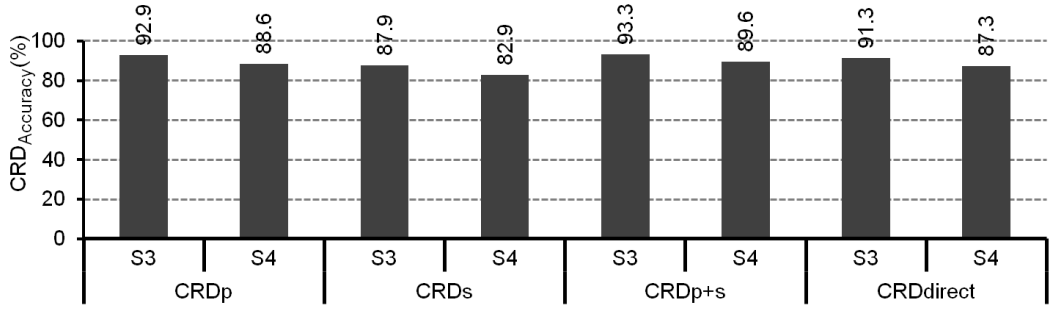


(d) Measured and predicted CRD_CMC profiles on 16 cores at the S4 problem size.

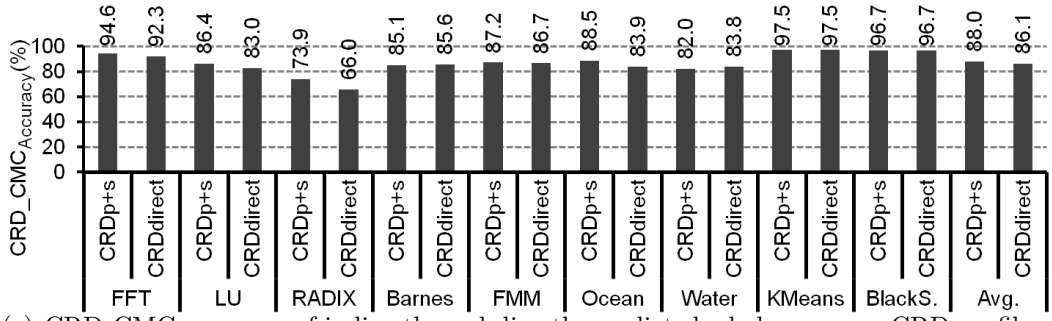
Figure 4.10: Examples for measured and predicted CRD_{direct} profiles with problem size scaling.



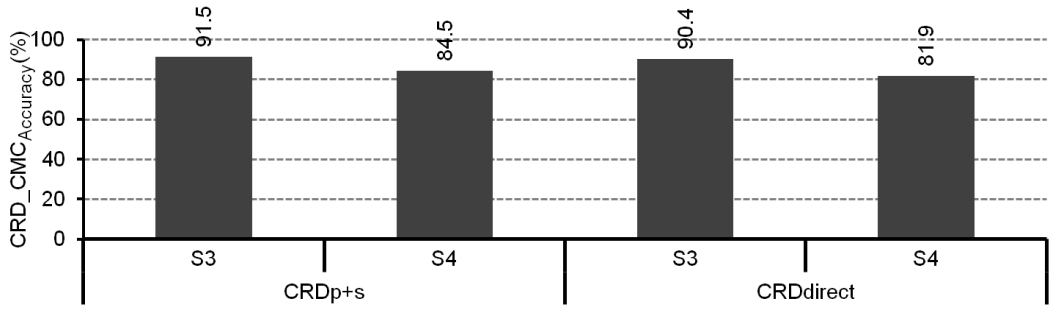
(a) CRD accuracy of predicted CRD_P and CRD_S profiles, and indirectly and directly predicted whole-program CRD profiles.



(b) The breakdown of CRD prediction accuracy by problem sizes.



(c) CRD_CMC accuracy of indirectly and directly predicted whole-program CRD profiles.



(d) The breakdown of CRD_CMC prediction accuracy by problem sizes.

Figure 4.11: CRD profile prediction accuracy for problem size scaling.

has better prediction accuracy than core count scaling. This is because shared-data profiles show the coherent shift for problem size scaling, as described in Section 3.4.

In Figure 4.11(a), the bars labeled "CRD_{P+S}" show the average $\text{CRD}_{Accuracy}$ for whole-program CRD profiles predicted by combining CRD_P and CRD_S predictions. For all benchmarks, CRD_{P+S} accuracy is between 82.6% and 99.3%. The average CRD_{P+S} accuracy for all benchmarks is 91.5%. The last set of bars in Figure 4.11(a), labeled "CRD_{direct}," show the average $\text{CRD}_{Accuracy}$ for direct whole-program CRD profile prediction. Figure 4.11(a) shows that CRD_{direct} is very similar to CRD_{P+S}. On average, CRD_{direct} accuracy is 89.3%, compared to 91.5% for CRD_{P+S}.

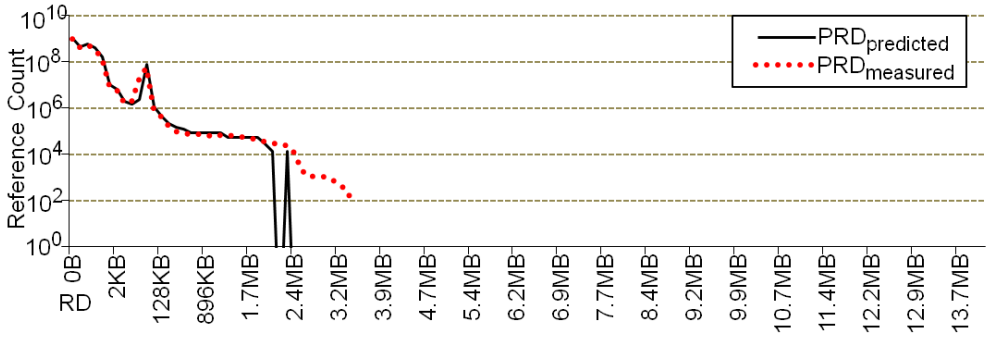
Finally, Figure 4.11(c) illustrates the whole-program prediction results using the $\text{CRD\_CMC}_{Accuracy}$ metric. Qualitatively, the $\text{CRD}_{Accuracy}$ and $\text{CRD\_CMC}_{Accuracy}$ results are the same. CRD_{P+S} and CRD_{direct} have similar $\text{CRD\_CMC}_{Accuracy}$, between 82.0%–97.5% for 8 benchmarks, and are roughly 70% for RADIX. In RADIX, the per-thread private data is large. For large core counts, reference groups detect small shift at the S1 and S2 problem sizes. However, the global data at the S3 and S4 problem sizes is large compared to per-thread private data, and the profiles do have large shift. So RADIX has low prediction accuracy. On average, CRD_{P+S} and CRD_{direct} achieve an 88.0% and 86.1% accuracy, respectively. Figure 4.11(b) and Figure 4.11(d) breaks down $\text{CRD}_{Accuracy}$ and $\text{CRD\_CMC}_{Accuracy}$ by different problem sizes. Again, the error increases as problem size scales.

4.4.2 PRD Profiles

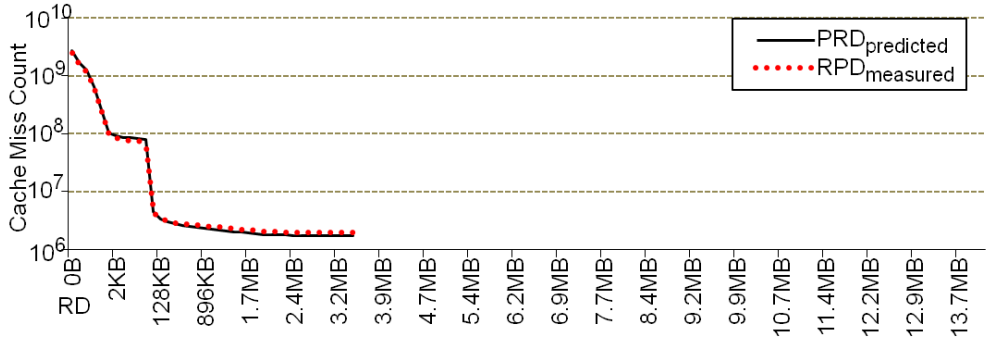
Figure 4.12 compares Barnes' measured PRD_{direct} profiles (dotted lines) with predicted PRD_{direct} profiles (solid lines) running on 16 cores at the S3 and S4 problem sizes. In Figure 4.12(a) and Figure 4.12(c), predicted PRD profiles capture the major shifting behavior for problem size scaling. At large RD values, predicted PRD profiles show distortion due to insufficient resolution. However, the region where distortion happens has very small reference counts compared to total reference counts. It is less important to predict this region with high accuracy. In Figure 4.12(a) and Figure 4.12(c), the $PRD_{Accuracy}$ of 16 cores at the S3 and S4 problem sizes is 99.9% and 68.9%, respectively.

For CMC profiles, we also predict compulsory misses and coherence misses, which have infinite reuse distance. In our prediction, we assume the cache misses at infinite reuse distance grow proportionally as problem size scales. Figure 4.12(b) and Figure 4.12(d) show the corresponding PRD_CMC profiles on 16 cores at the S3 and S4 problem sizes. Although reference groups lose some detailed information at some RD values in predicted PRD profiles, integration reduces the impact. As a result, the $PRD\_CMC_{Accuracy}$ of the S3 and S4 problem sizes is 89.8% and 79.7%, respectively. Although the error is 20% at S4, the predicted PRD_CMC profile still captures the trend of the application's cache performance.

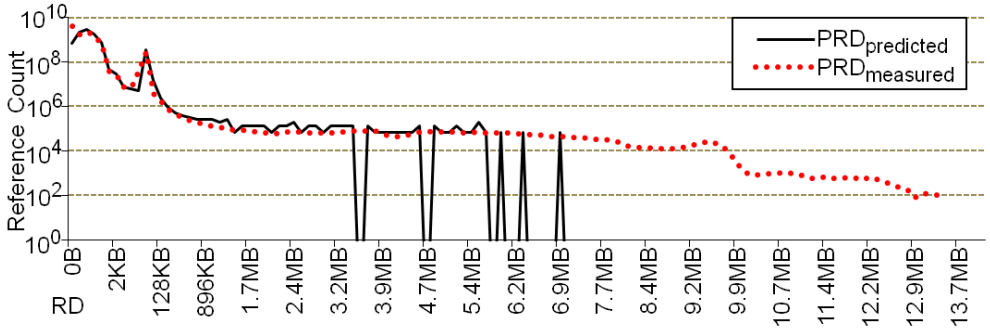
As Figure 4.13(a) shows, PRD_P profiles are predicted with high accuracy, between 81.7% and 99.9%. Across all benchmarks, the average PRD_P accuracy is 94.0%. Compared to PRD_P profiles, PRD_S profiles are predicted with slightly lower



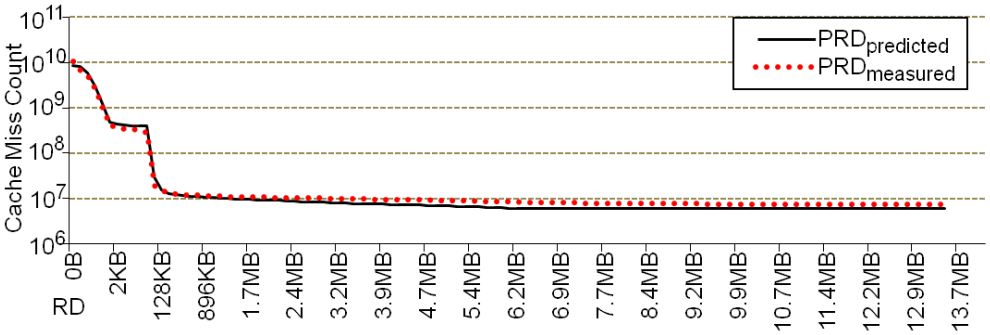
(a) Measured and predicted PRD profiles on 16 cores at the S3 problem size.



(b) Measured and predicted PRD_CMC profiles on 16 cores at the S3 problem size.

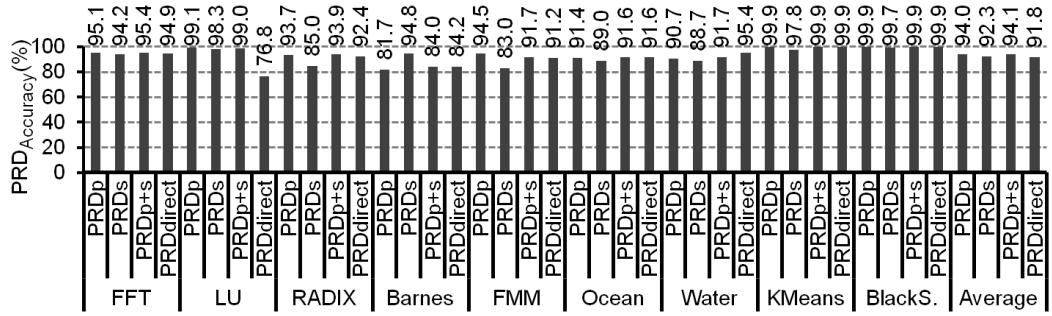


(c) Measured and predicted PRD profiles on 16 cores at the S4 problem size.

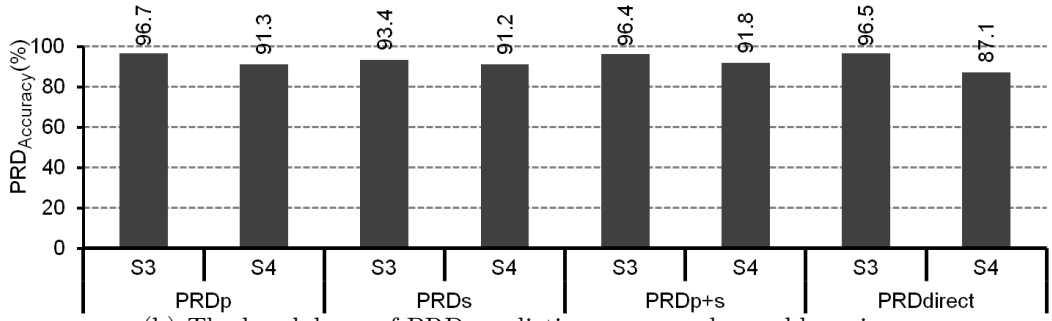


(d) Measured and predicted PRD_CMC profiles on 16 cores at the S4 problem size.

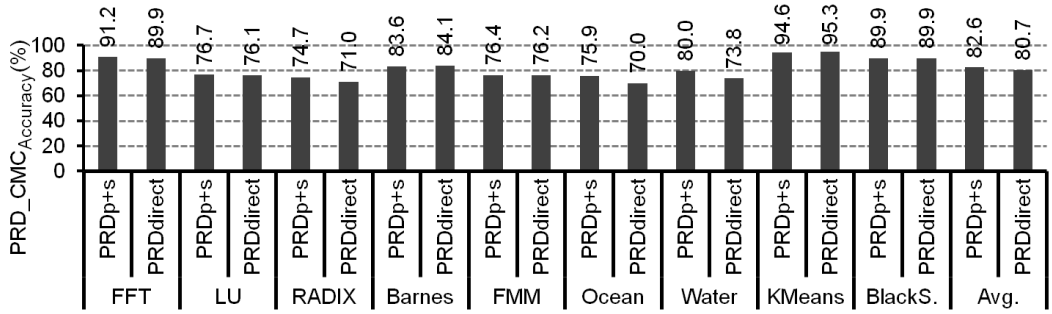
Figure 4.12: Examples for measured and predicted PRD$_{direct}$ profiles with problem size scaling.



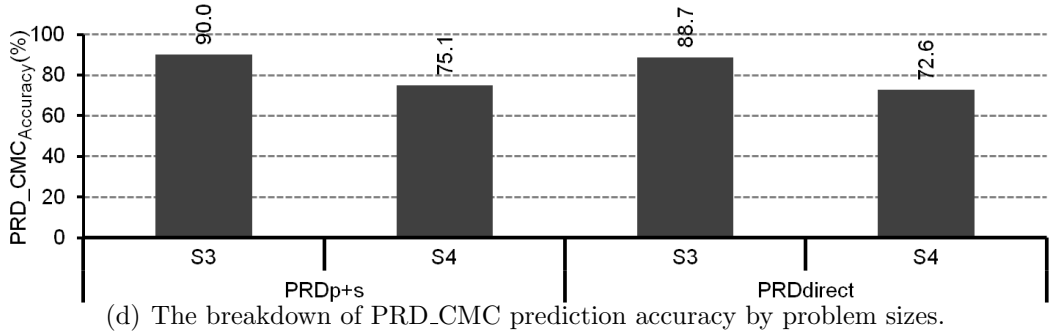
(a) PRD accuracy of predicted PRD_P and PRD_S profiles, and indirectly and directly predicted whole-program PRD profiles.



(b) The breakdown of PRD prediction accuracy by problem sizes.



(c) PRD_CMC accuracy of indirectly and directly predicted whole-program PRD profiles.



(d) The breakdown of PRD_CMC prediction accuracy by problem sizes.

Figure 4.13: PRD profile prediction accuracy for problem size scaling.

accuracy. PRD_S accuracy is between 83.0% and 99.7%. Across all benchmarks, the average PRD_S accuracy is 92.3%. PRD_{P+S} accuracy is between 84.0% and 99.9%. The average PRD_{P+S} accuracy for all benchmarks is 94.1%. The last set of bars, PRD_{direct}, shows PRD_{direct} is less accurate than PRD_{P+S}. On average, PRD_{direct} accuracy is 91.8%.

Finally, Figure 4.13(c) illustrates the whole-program prediction results using the $\text{PRD\_CMC}_{Accuracy}$ metric. PRD_{P+S} and PRD_{direct} have similar CMC accuracy. They are between 70.0%–95.3%. On average, PRD_{P+S} and PRD_{direct} achieve an 82.6% and 80.7% accuracy, respectively. Figure 4.13(b) and Figure 4.13(d) breaks down the $\text{PRD}_{Accuracy}$ and $\text{PRD\_CMC}_{Accuracy}$ by different problem sizes. Again, the error increases as problem size scales.

Comparing the prediction accuracy of CRD and PRD profiles, the reference groups technique can predict CRD and PRD profiles with similar accuracy for problem size scaling. However, at large problem sizes, $\text{PRD\_CMC}_{Accuracy}$ is less accurate than $\text{CRD\_CMC}_{Accuracy}$. This is because we need to predict compulsory misses and coherence misses for PRD profiles. The growing rate of these misses varies at different configurations, and it is not very regular. The reference groups technique also under-predicts the shift of PRD profiles at the S4 problem size. Hence, $\text{PRD\_CMC}_{Accuracy}$ has a higher error.

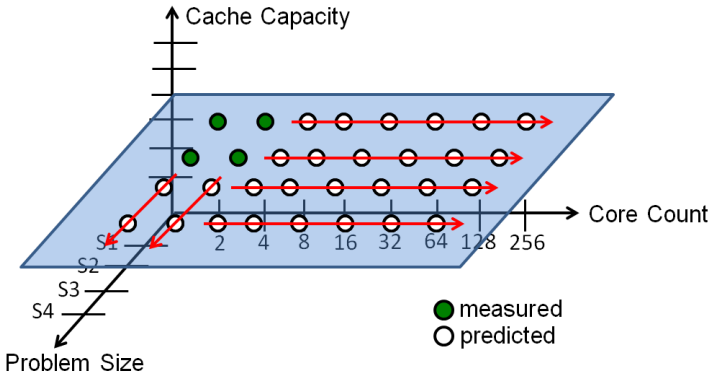


Figure 4.14: Profile prediction for core-problem scaling.

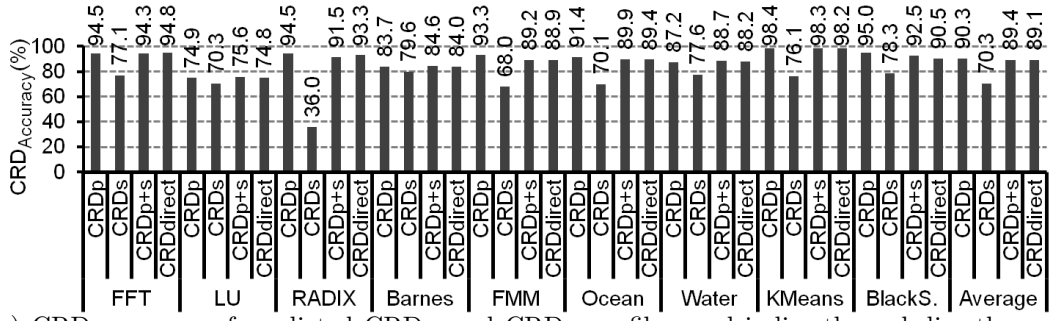
4.5 Prediction Accuracy Results for Core-Problem Scaling

Lastly, we combine core count prediction and problem scaling prediction together. At the S1 and S2 problem sizes, we used the measured profiles at 2 and 4 cores to predict the profiles at the S3 and S4 problem sizes. Then from these 8 profiles, we use core count prediction techniques to predict the other 24 profiles. This technique predicts 28 profiles from only 4 profiles. Figure 4.14 illustrates the measured and predicted points.

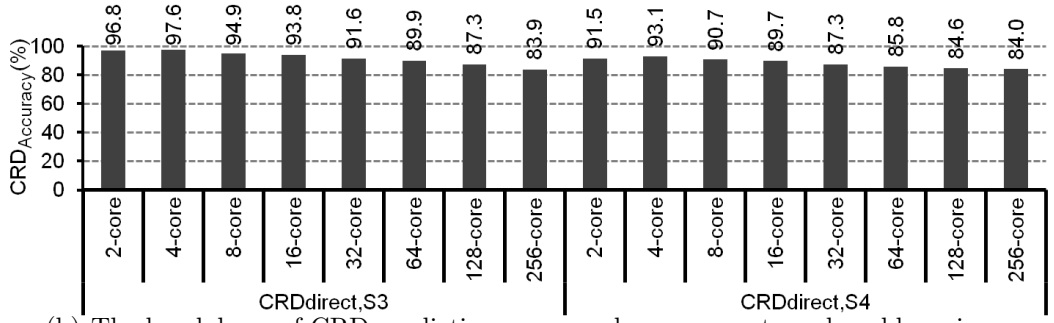
4.5.1 CRD Profiles

Figure 4.15 presents our full CRD profile prediction results for scaling core count and problem size together. Each bar reports the average CRD accuracy achieved over the 28 predictions per benchmark. The rightmost bars report the average accuracy across all benchmarks.

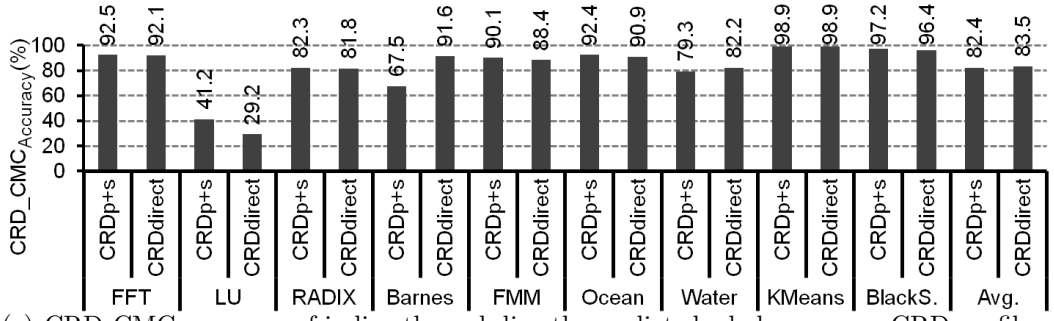
As Figure 4.15(a) shows, CRD_P profiles are predicted with high accuracy. For all benchmarks, CRD_P accuracy is between 74.9% and 98.4%. Across all bench-



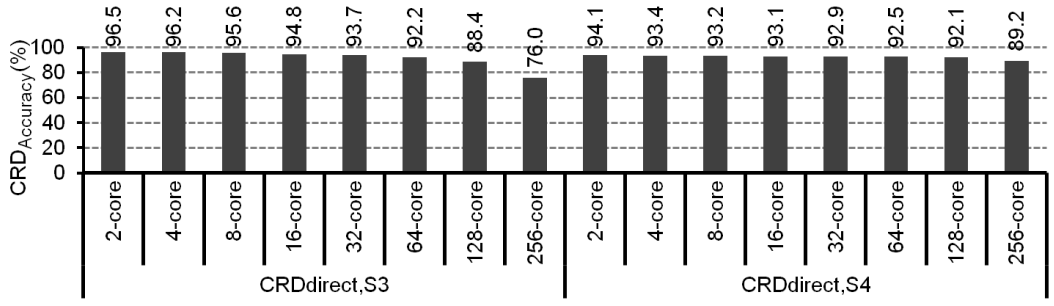
(a) CRD accuracy of predicted CRD_P and CRD_S profiles, and indirectly and directly predicted whole-program CRD profiles.



(b) The breakdown of CRD prediction accuracy by core counts and problem sizes.



(c) CRD_CMC accuracy of indirectly and directly predicted whole-program CRD profiles.



(d) The breakdown of CRD_CMC prediction accuracy by core counts and problem sizes.

Figure 4.15: CRD profile prediction accuracy for scaling core count and problem size together.

marks, average CRD_P accuracy is 90.3%. CRD_P profiles exhibit coherent shift across problem size and core count scaling which reference groups can effectively predict. Compared to CRD_P profiles, CRD_S profiles are predicted with lower accuracy. In Figure 4.15(a), CRD_S accuracy is between 36.0% and 79.6%. Across all benchmarks, the average CRD_S accuracy is only 70.3%.

In Figure 4.15(a), the bars labeled "CRD_{P+S}" show the average $\text{CRD}_{Accuracy}$ for whole-program CRD profiles predicted by combining CRD_P and CRD_S predictions. For all benchmarks, CRD_{P+S} accuracy is between 75.6% and 98.3%. The average CRD_{P+S} accuracy for all benchmarks is 89.4%. The last set of bars in Figure 4.15(a), labeled "CRD_{direct}," report the average CRD accuracy for direct whole-program CRD profile prediction. Figure 4.15(a) shows CRD_{direct} is very similar to CRD_{P+S}. On average, CRD_{direct} accuracy is 89.1%, compared to 89.4% for CRD_{P+S}.

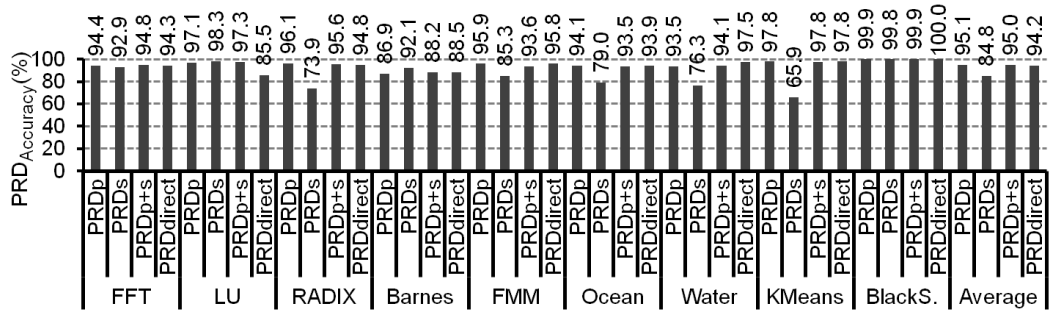
Finally, Figure 4.15(c) illustrates the whole-program prediction results using the $\text{CRD\_CMC}_{Accuracy}$ metric. CRD_{P+S} and CRD_{direct} have similar accuracy, between 67.5%–98.9% for 8 benchmarks, and roughly 30% for LU. LU has low prediction accuracy due to core count scaling. On average, CRD_{P+S} and CRD_{direct} achieve a 87.5% and 90.3% accuracy, respectively, without LU, and 82.4% and 83.5% accuracy, respectively, for all benchmarks. Figure 4.15(b) and Figure 4.15(d) break down the $\text{CRD}_{Accuracy}$ and $\text{CRD\_CMC}_{Accuracy}$ by different core counts and problem sizes. The prediction accuracy degrades as the prediction horizon increases.

4.5.2 PRD Profiles

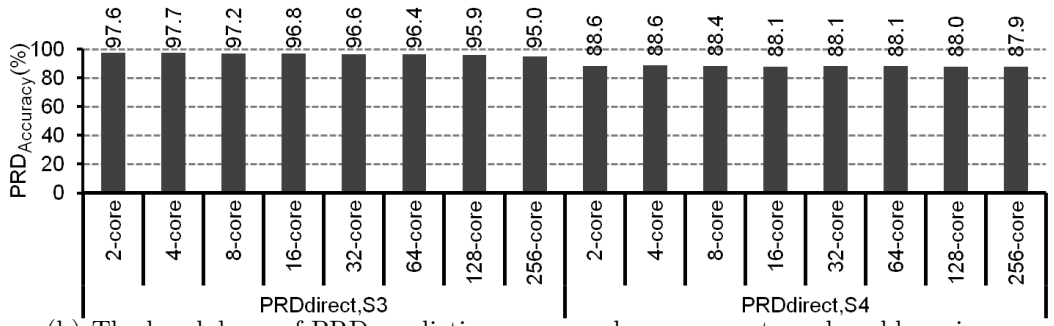
Figure 4.16 presents our PRD profile prediction results under core-problem scaling. As Figure 4.16(a) shows, PRD_P profiles are predicted with high accuracy, between 86.9% and 99.9%. Across all benchmarks, the average PRD_P accuracy is 95.1%. Compared to PRD_P profiles, PRD_S profiles are predicted with lower accuracy. PRD_S accuracy is between 65.9% and 99.8%. Across all benchmarks, the average PRD_S accuracy is only 84.8%. PRD_S profiles have lower prediction accuracy due to core count scaling. PRD_{P+S} accuracy is between 88.2% and 99.9%. The average PRD_{P+S} accuracy for all benchmarks is 95.0%. The last set of bars, PRD_{direct}, shows that PRD_{direct} is slightly worse than PRD_{P+S}. On average, PRD_{direct} accuracy is 94.2%.

Finally, Figure 4.16(c) illustrates the whole-program prediction results using the $\text{PRD\_CMC}_{Accuracy}$ metric. PRD_{P+S} and PRD_{direct} have similar CMC accuracy, between 65.7%–97.0%. On average, PRD_{P+S} and PRD_{direct} achieve 80.9% and 80.8% accuracy, respectively. Figure 4.16(b) and Figure 4.16(d) break down the $\text{PRD}_{Accuracy}$ and $\text{PRD\_CMC}_{Accuracy}$ by different core counts and problem sizes. The prediction accuracy degrades as the prediction horizon increases.

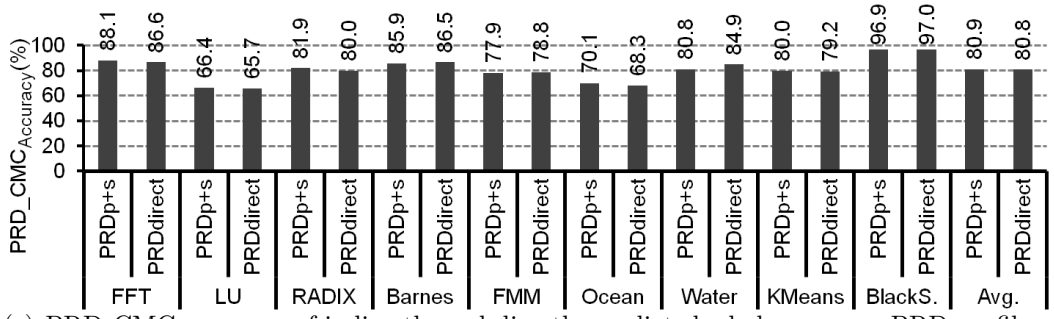
In general, the prediction accuracy results of core count scaling, problem size scaling, and core-problem scaling are qualitatively similar. However, the CMC prediction accuracy of PRD profiles at large core counts and problem sizes is lower than the prediction accuracy of CRD profiles. This is because we need to predict compulsory misses and coherence misses for PRD profiles. The growing rate varies at



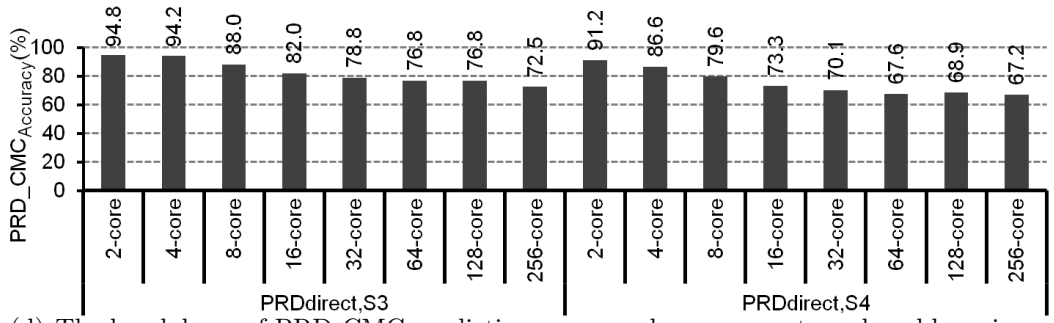
(a) PRD accuracy of predicted PRD_P and PRD_S profiles, and indirectly and directly predicted whole-program PRD profiles.



(b) The breakdown of PRD prediction accuracy by core counts and problem sizes.



(c) PRD_CMC accuracy of indirectly and directly predicted whole-program PRD profiles.



(d) The breakdown of PRD_CMC prediction accuracy by core counts and problem sizes.

Figure 4.16: PRD profile prediction accuracy for scaling core count and problem size together.

different configurations, and it is harder to predict accurately. The reference groups also under-predict the shift of PRD profiles at the S4 problem size. In contrast, the compulsory misses in CRD profiles can be predicted with higher accuracy. As a result, CRD profiles have higher CMC prediction accuracy.

Chapter 5

Multicore Cache Performance Prediction

In Chapter 2, we assume that CRD and PRD profiles are minimally cache capacity dependent for loop-based parallel programs, so our Pin tool interleaves inter-thread memory references uniformly. To prove our assumption is valid, we use our M5 simulator to investigate the profile stability across cache capacity scaling. Then we evaluate the accuracy of using CRD and PRD profile predictions to estimate the multicore cache performance, in particular MPKI (misses per kilo-instructions).

5.1 Architecture Assumptions

In Intel's tera-scale project[27], Mani et al. point out that one of the basic requirements for the on-chip interconnection is scalability. The average communication distance should be sub-linear with respect to the number of cores. One possible solution is to distribute the communication across the chip on different paths. The tiled CMP with a 2D-mesh network provides this capability. Recently, Tilera Corporation has shipped tiled CMPs[28] with 16 to 100 cores.

A tiled CMP, illustrated in Figure 5.1, consists of several identical replicated tiles. Each tile contains a processor core, a multi-level cache hierarchy, and a switch for a 2D mesh network. The switch connects four directions with its neighbors. When the requesting node is not the neighbor of the requested node, multiple hops

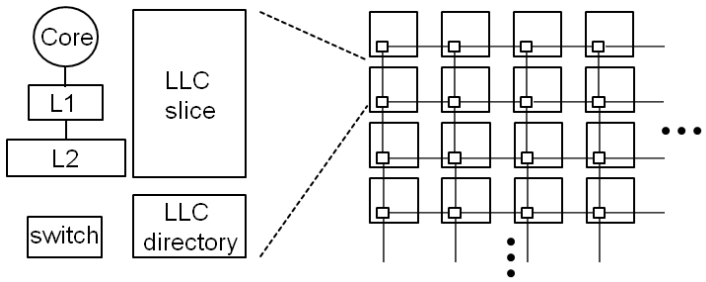


Figure 5.1: Tiled CMP.

are required. Different configurations can be achieved easily by replicating tiles. As a result, tiled CMPs are regarded as a scalable CMP organization [29, 30], and we can use tiled CMPs to study a large design space.

We use the M5 simulator[31] to model tiled CMPs. Our simulator's core is in-order, with each core executing one instruction per cycle in the absence of memory stalls. Each core has its own dedicated private caches, and the last level cache (LLC) is shared. We permit replication of data in each tile's private caches, and cache coherence is maintained by the MESI directory-based cache coherence protocol. For the shared LLC, LLC slices are managed as a simple shared cache, with no migration or replication across LLC slices. Each cache block resides in a fixed LLC slice which is known as the cache block's home tile. Each cache block's directory entry is also co-located with its associated data. We assume cache block homes are page-interleaved (with 8KB page size) across LLC slices according to their physical address.

We assume full-map directories, though this approach can lead to large directories. Several scalable directory schemes can be applied to reduce the directory size, for example, limited directories or sparse directories [32, 33, 34, 35]. Although the

directory size plays an important role in architecture designs, this issue is beyond the scope of our research.

The M5 simulator is also modified to support 4 DRAM channels, each connected to a memory controller on a special memory tile. Four memory tiles are evenly spaced on the north and south edges of the chip. We accurately model cache access, hops through the network, and DRAM access. We also model queuing at the on-chip network and memory controllers.

All of our experiments simulate application code only without any operating system code (*i.e.*, the OS is emulated) due to the difficulty of performing full-system simulations at 256 cores. During simulation, our simulator records the direct-measured whole-program CRD and PRD profiles (CRD_{direct} and PRD_{direct}) for the pre-L1 memory reference stream across all cores. CRD and PRD are computed at the granularity of 64 bytes, the block size for the caches.

To drive our simulations, we use the same benchmarks and problem sizes from Table 2.1. For benchmarks that run one time-step, we warm-up caches in the first time-step, then we begin recording performance and profiles at the second time-step. For benchmarks running the entire parallel phase, we do not perform any explicit cache warm-up.

Table 5.1 lists the parameters used in our simulations for the shared cache performance study. As Table 5.1 shows, we use a two-level cache hierarchy. Each core has its own private 32KB instruction cache and 32KB data cache. The L2 slices form a logically distributed shared last level cache. We simulate processors with 2–256 cores and 4–128MB of total L2 cache (LLC). There are 192 configurations per

Table 5.1: Simulator parameters used in the shared cache performance experiments.

Number of Tiles	2, 4, 8, 16, 32, 64, 128, 256
Core Type	Alpha ISA, Single issue, In-order, CPI = 1, clock speed = 2GHz
IL1/DL1	32KB/32KB, 64B block, 8-way, 1 CPU cycle
Total L2 Cache Size	4MB, 8MB, 16MB, 32MB, 64MB, 128MB
L2 Slice	64B blocks, 32-way, 10 CPU cycles
2-D Mesh	3 CPU cycles per-hop, bi-directional channels, 256-bit wide links
Memory channels	latency: 200 CPU cycles, bandwidth: 32GB(1-16cores) and 64GB(32-256cores)

Table 5.2: Simulator parameters used in the private cache performance experiments.

Number of Tiles	2, 4, 8, 16, 32, 64, 128, 256
Core Type	Alpha ISA, Single issue, In-order, CPI = 1, clock speed = 2GHz
IL1/DL1	8KB/8KB, 64B block, 4-way, 1 CPU cycle
Per-core L2 Cache Size	16KB, 32KB, 64KB, 128KB, 256KB
Per-core L2	64B blocks, 8-way, Latency = 4 CPU cycles
Total L3 Cache Size	32MB(1-16cores) and 128MB(32-256cores)
L3 Slice	64B blocks, 32-way, 10 CPU cycles
2-D Mesh	3 CPU cycles per-hop, bi-directional channels, 256-bit wide links
Memory channels	latency: 200 CPU cycles, bandwidth: 32GB(1-16cores) and 64GB(32-256cores)

benchmark, and 1,728 configurations across our 9 benchmarks.

Table 5.2 lists the parameters used in our simulations for the private cache performance study. To study the private cache performance, we use a three-level cache hierarchy. Each core has its own private 8KB instruction cache, 8KB data cache, and a unified L2 cache. The L2 cache capacity varies from 16KB to 256KB. The L3 slices form a logically distributed shared last level cache. There are 160 configurations per benchmark, and 1,440 configurations across our 9 benchmarks.

5.2 Profile Stability

Before presenting our prediction results, we first revisit the issue of architecture dependence. The multicore cache performance not only depends on intra-thread data locality, but also depends on inter-thread interactions. At different cache capacities, the relative execution speed between threads may change, and this may change the memory reference interleaving. As a result, the CRD and PRD profiles

measured on one cache size may not be valid for other cache sizes. So, strictly speaking, CRD and PRD profiles are not even valid across different cache sizes at the same core count. This instability defeats the benefits of multicore RD analysis. However, when threads exhibit similar locality behavior, this instability becomes minimal. For example, loop-based parallel programs often employ *symmetric threads*. When the cache size changes, these threads tend to either speed up or slow down, but by the same amount . For such programs, CRD and PRD profiles are practically stable and can provide accurate analysis for different cache capacities. In this section, we study the stability of CRD and PRD profiles across cache capacity scaling.

5.2.1 CRD Profiles

To study the stability of CRD profiles across LLC capacity scaling, we use a two-level cache hierarchy (Table 5.1). Figure 5.2 plots CRD profiles from the FFT benchmark, all running on 64 cores with the S2 problem size, but varies the LLC size at 8MB, 32MB, and 128MB. As Figure 5.2 shows, CRD profiles change with LLC capacity, so they are indeed architecture dependent. However, these profiles are almost identical, and they exhibit low sensitivity to LLC scaling. This is because LLC scaling speeds up or slows down symmetric threads by similar amounts. So, profiles tend to remain the same.

To quantify this stability, we compare CRD profiles measured at different LLC capacities. For each benchmark, core count, and problem size, we compare the CRD profiles at capacities C = 4, 8, 16, 64, and 128MB against the baseline CRD profile

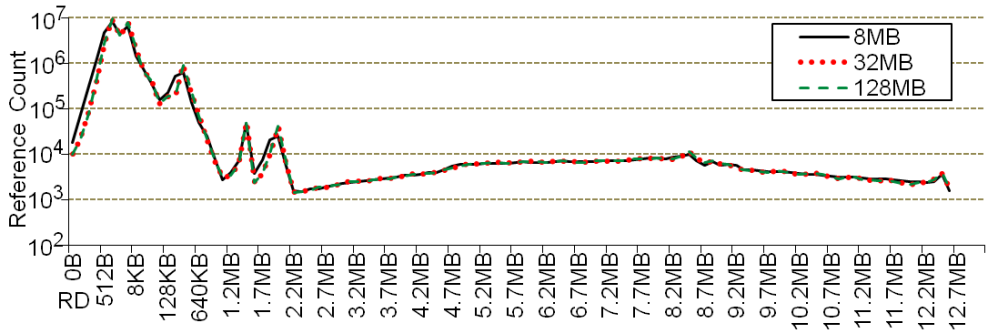


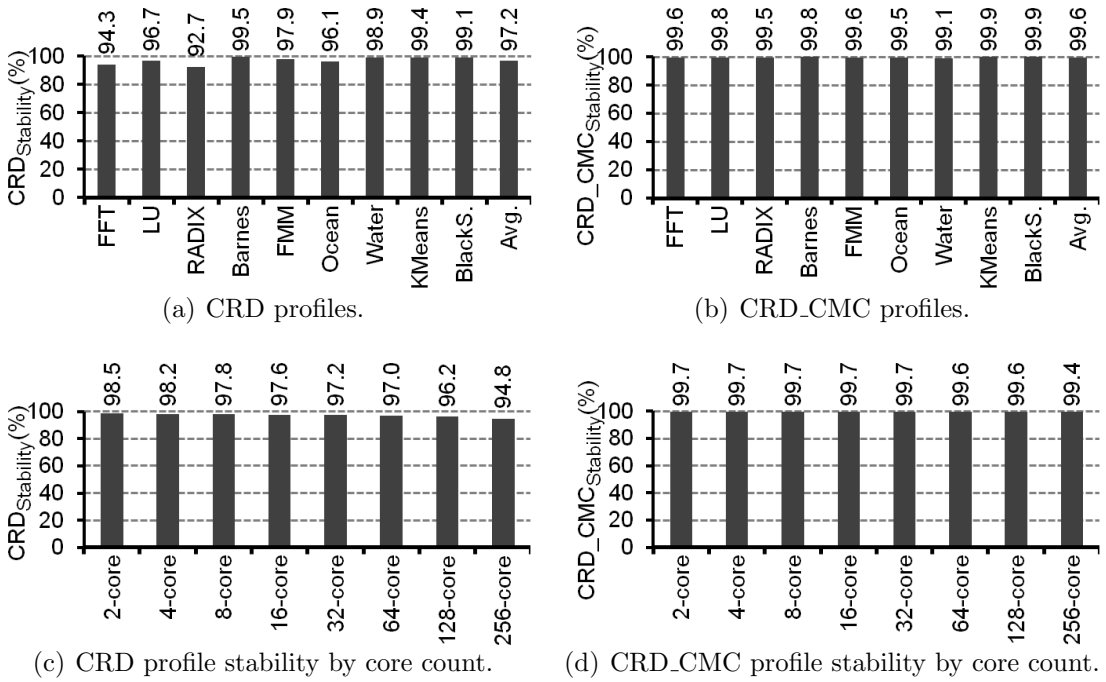
Figure 5.2: CRD profiles from the FFT benchmark running on 64 cores at the S2 problem size across 8M, 32M, and 128M LLC capacity.

at capacity C = 32MB. For each pairwise profile comparison, we use two metrics. $\text{RD}_{Stability}$ is defined in Equation 5.1, and N is the number of bins. $\text{RD}_{Stability}$ is $1 - \frac{E}{2}$, where E is the sum of the normalized absolute differences between every pair of reference counts from a measured and baseline RD profile. The second metric is $\text{RD\_CMC}_{Stability}$. We compute $\text{RD\_CMC}_{Stability}$ by averaging the error between pairs of RD values from the entire measured and baseline CMC profiles, as illustrated in Equation 5.2. The $\text{RD}_{Stability}$ and $\text{RD\_CMC}_{Stability}$ for CRD profiles are $\text{CRD}_{Stability}$ and $\text{CRD\_CMC}_{Stability}$.

$$RD_{Stability} = 1 - \frac{1}{2}\sum_{i=0}^{N-1}\frac{|RD_C[i] - RD_{baseline}[i]|}{total\ references} \tag{5.1}$$

$$RD\_CMC_{Stability} = 1 - \frac{1}{N}\sum_{i=0}^{N-1}\frac{|RD\_CMC_C[i] - RD\_CMC_{baseline}[i]|}{RD\_CMC_{baseline}[i]} \tag{5.2}$$

Figure 5.3 reports the stability measurement across our 9 benchmarks. In Figure 5.3(a), the $\text{CRD}_{Stability}$ is between 92.7% and 99.5%. Across all benchmarks, the average $\text{CRD}_{Stability}$ is 97.2%. Figure 5.3(c) shows the breakdown of $\text{CRD}_{Stability}$ by



(a) CRD profiles. (b) CRD_CMC profiles.

(c) CRD profile stability by core count. (d) CRD_CMC profile stability by core count.

Figure 5.3: Stability measurement of CRD profiles and CRD_CMC profiles.

core count. The stability decreases as core count increases, from 98.5% to 94.8%. This is because larger core counts have a higher probability to have idle cores due to the timing effect, and this might cause more irregular memory reference interleaving. In Figure 5.3(b), the CRD_CMC$_{Stability}$ is between 99.1% and 99.9%. Across all benchmarks, the average CRD_CMC$_{Stability}$ is 99.6%. The results suggest that CRD_CMC profiles are more stable than CRD profiles. The reason is that the variation is minimized when integrating cache-miss counts. Figure 5.3(d) shows the breakdown of CRD_CMC$_{Stability}$ by core count. The stability decreases as core count increases, from 99.7% to 99.4%. These results demonstrate that the vast majority of CRD profiles exhibit low sensitivity to LLC capacity scaling.

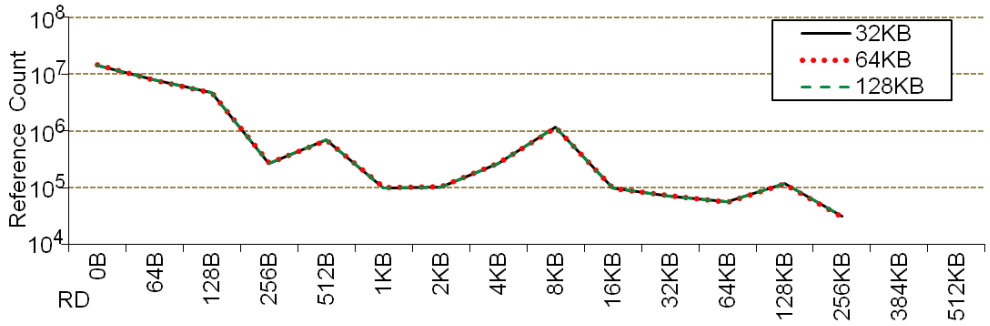
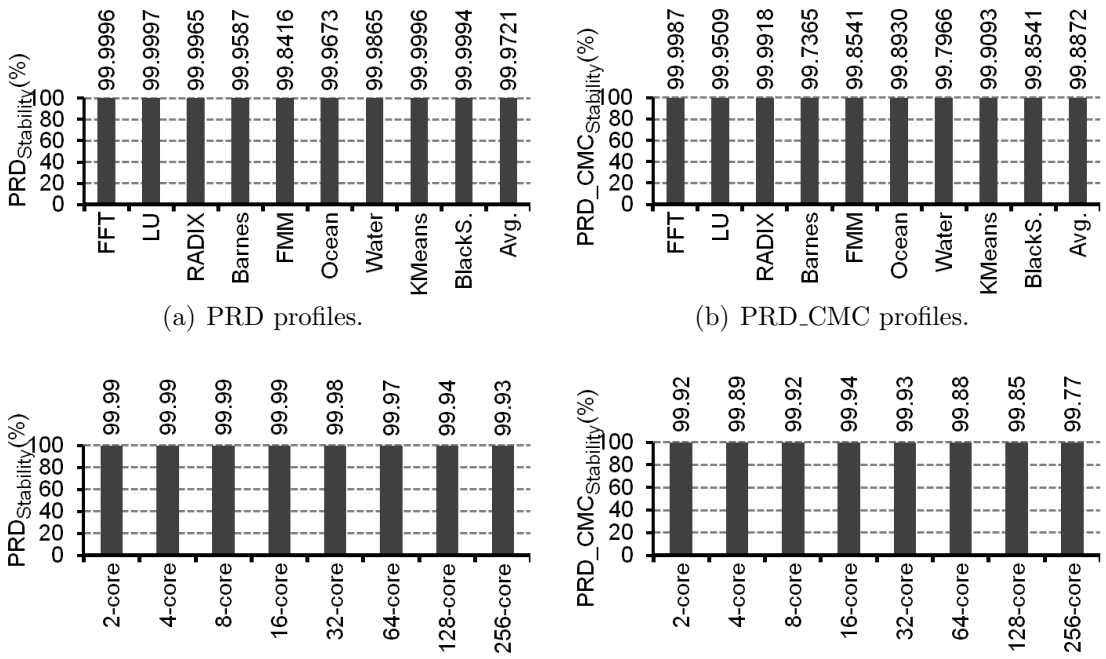


Figure 5.4: PRD profiles from the FFT benchmark running on 64 cores at the S2 problem size across 32K, 64K, and 128K per-core L2 capacity.

5.2.2 PRD Profiles

To study the stability of PRD profiles across L2 capacity scaling, we use a three-level cache hierarchy (Table 5.2). Figure 5.4 plots PRD profiles from the FFT benchmark, all running on 64 cores with the S2 problem size, but varies the L2 size at 32KB, 64KB, and 128KB. As Figure 5.4 shows, these PRD profiles are almost identical, and they exhibit very low sensitivity to L2 size scaling. This is because PRD profiles are not sensitive to inter-thread memory reference interleaving. Although inter-thread interactions cause invalidations at large RD values in PRD profiles, the number of invalidations is usually small compared to the total number of memory references. As a result, PRD profiles are more stable than CRD profiles.

To quantify this stability, we compare PRD profiles measured at different L2 capacities. For each benchmark, core count, and problem size, we compare the PRD profiles at capacities C = 16, 32, 128, and 256KB against the baseline PRD profile at capacity C = 64KB. Figure 5.5 illustrates the stability measurement across our 9 benchmarks. In Figure 5.5(a), the $\text{PRD}_{Stability}$ is between 99.8416% and 99.9997%. Across all benchmarks, the average $\text{PRD}_{Stability}$ is 99.97%. Figure 5.5(c) shows the



(a) PRD profiles.

(b) PRD_CMC profiles.

(c) PRD profile stability by core count.

(d) PRD_CMC profile stability by core count.

Figure 5.5: Stability measurement of PRD profiles and PRD_CMC profiles.

breakdown of $\text{PRD}_{Stability}$ by core counts. The stability is very high across core count, from 99.93% to 99.99%. This is because the number of invalidations is relatively small compared to total reference counts, and the timing effect is insignificant. In Figure 5.5(b), the $\text{PRD\_CMC}_{Stability}$ is between 99.7365% and 99.9987%. Across all benchmarks, the average $\text{PRD\_CMC}_{Stability}$ is 99.89%. The results suggest PRD and PRD_CMC profiles are very stable. Figure 5.5(d) shows the breakdown of $\text{PRD\_CMC}_{Stability}$ by core count. The stability decreases as core count increases, from 99.92% to 99.77%, but still higher than 99%. These results demonstrate that PRD profiles are indeed more stable than CRD profiles.

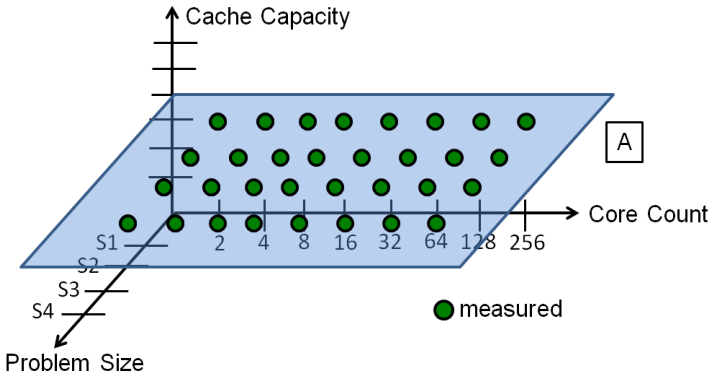


Figure 5.6: Architecture-application design space.

5.3 MPKI Prediction Accuracy

Chapter 4 evaluates the profile prediction accuracy for different scaling schemes. In this section, we evaluate the accuracy of using CRD and PRD profile predictions to estimate the cache performance, in particular MPKI.

5.3.1 Prediction Approach

Performance prediction is a three-step process. First, we acquire the CRD and PRD profiles at some configurations. Second, we use the prediction techniques described in Section 4.1 to predict the CRD and PRD profiles at different configurations. Finally, we use CRD profiles to predict shared cache performance, and use PRD profiles to predict private cache performance. In this study, we consider three prediction strategies: "No-Pred," "C-Pred," and "CP-Pred."

No-Pred does not perform any profile prediction. For each benchmark, it acquires the CRD or PRD profiles on the 32 configurations across core count and problem size in the "A" plane of Figure 5.6. The "A" plane is at 32MB LLC size

when acquiring the CRD profiles, and it is at 64KB L2 size when acquiring the PRD profiles. At each measured point, No-Pred uses the profile to predict the MPKIs at different cache sizes.

C-Pred extends No-Pred with core count prediction to reduce the profiles needed along the X-axis in Figure 5.6. At each problem size within the "A" plane, C-Pred predicts the 8- to 256-core profiles using the 2- and 4-core profiles, as illustrated in Figure 4.4. So C-Pred uses 8 measured profiles to predict the other 24 profiles for each benchmark. Then just like No-Pred, C-Pred uses the profiles in the "A" plane to predict the MPKIs at different cache sizes.

Lastly, CP-Pred extends C-Pred with problem scaling prediction to reduce the profiles needed along the Y-axis in Figure 5.6. CP-Pred acquires 2- and 4-core profiles at the S1 and S2 problem sizes, and it predicts the S3 and S4 profiles from the S1 and S2 profiles. Then just like C-Pred, CP-Pred predicts across core count to acquire all profiles in the "A" plane. Then, CP-Pred uses the profiles in the "A" plane to predict the MPKIs at different cache sizes. For each benchmark, CP-Pred only needs 4 measured points.

Once all 36 profiles within the "A" plane have been acquired (under No-Pred, C-Pred, or CP-Pred), we use Qasem and Kennedy's model [36] to predict capacity and conflict misses together at the desired capacity, C. This model takes the RD profile as input, and uses a binomial distribution to predict the number of capacity and conflict misses for a given capacity and associativity. Finally, we divide the predicted cache misses by instruction count to derive MPKI. For No-Pred, we use the measured instruction count at the same configuration that contributed the RD

profile for MPKI prediction. For C-Pred and CP-Pred, we make the assumption that instruction count grows proportionally with core count and problem size.

There are 1,728 configurations in the shared LLC design space, and there are 1,440 configurations in the private L2 design space. We simulate all of them using our M5 simulator and obtain their MPKIs and profiles. When computing MPKIs, we exclude compulsory misses, since there is no cache warm-up in reuse distance profiles. Then, we use Equation 5.3 to compute the MPKI prediction accuracy. When the measured MPKI is small, the prediction error often blows up. However, the small MPKI doest not really affect CPU performance. To address this, we add a small offset to the predicted and measured values. This offset is selected to be the MPKI value which can cause 1% CPI difference by assuming the memory latencies from Table 5.2.

$$Error = \frac{|(MPKI_{measured} + offset) - (MPKI_{predicted} + offset)|}{(MPKI_{measured} + offset)} \tag{5.3}$$

5.3.2 Shared LLC MPKI Prediction Accuracy

Figure 5.7 shows MPKI prediction error with a small offset, 0.05. Each bar in Figure 5.7 reports the average percent error across all predictions for a particular prediction strategy and benchmark. The rightmost group of bars reports averages across all 9 benchmarks.

As Figure 5.7 shows, No-Pred is able to predict shared LLC MPKI within 10.4% of simulation for 8 out of 9 benchmarks, and within 26.7% for RADIX. Across all benchmarks, prediction error is 9.4%. These results reflect baseline prediction

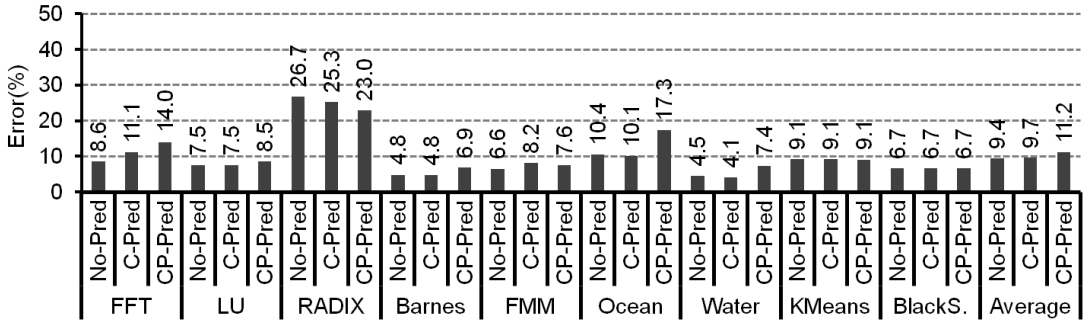


Figure 5.7: Percent shared LLC MPKI prediction error with 0.05 offset.

errors (*i.e.*, without profile prediction), and include three error sources. First, we find one of the main sources of error is the cache conflict model, especially for machines with large core count and small LLC. These machines incur pathologic conflicts that the conflict model cannot predict. Second, our error metric does not always address numeric instability. In some cases, LLC MPKI is near 0.05. These are not eliminated by our 0.05 offset, but are small enough to make percent error very sensitive to minute prediction errors. This is responsible for the high errors in RADIX. If we change the offset to be 0.5, No-Pred achieves 9.3% error for RADIX. And third, M5 profiles include timing effects. However, this error is very small. Section 5.2.1 demonstrates that CRD profiles are vary stable across cache capacity scaling. Overall, Figures 5.7 shows that No-Pred error is very low, so we can use CRD profiles to predict MPKI for loop-based parallel programs accurately.

Figure 5.7 also shows that C-Pred and CP-Pred are both less accurate than No-Pred. They are able to predict MPKI within 17.3% of simulation for 8 out of 9 benchmarks, and within 25.3% for RADIX. On average, prediction error is within 11.2%. Like No-Pred, C-Pred and CP-Pred incur cache conflict model errors. But they also incur errors due to CRD profile prediction. Sometimes the profile pre-

diction errors are additive, so total error increases. However, in some cases, errors cancel each other out. This is because the cache conflict model usually under-predicts cache misses, whereas CRD profile prediction sometimes over-predicts capacity misses. This explains why C-Pred and CP-Pred have errors similar to those of C-Pred (and in some cases even lower).

Figure 5.8 illustrates MPKI prediction error, just like Figure 5.7, but only for the S4 problem and caches with 4–16MB capacity. The results are still using 2–256 cores. Most of these configurations have cache size $< C_{core}$. Hence, Figure 5.8 examines prediction accuracy in the region of CRD profile shift. However, because LU, KMeans, and BlackScholes have small C_{core}, which are always smaller than 4MB, we omit these three benchmarks. As Figure 5.8 shows, prediction error in the shifting region is comparable to prediction error in the entire design space.

Figure 5.8 does not contain RADIX's poorly predicted cases. As a result, No-Pred is able to predict shared LLC MPKI within 11.7% of simulation for 6 benchmarks, and prediction error is 5.5% on average. C-Pred is able to predict shared LLC MPKI within 8.4% of simulation for 6 benchmarks, and prediction error is 6.4% on average. The results show that our core count prediction techniques are effective in the shifting region. However, CP-Pred is less effective, with 12.5% error due to more significant miss-prediction.

Figure 5.9 reports the prediction error for the same problem and cache sizes in Figure 5.8 broken down by prediction strategy and core count. Like Figure 5.8, Figure 5.9 shows that C-Pred is very similar to No-Pred, while CP-Pred is worse. More importantly, Figure 5.9 also shows that prediction error increases with core

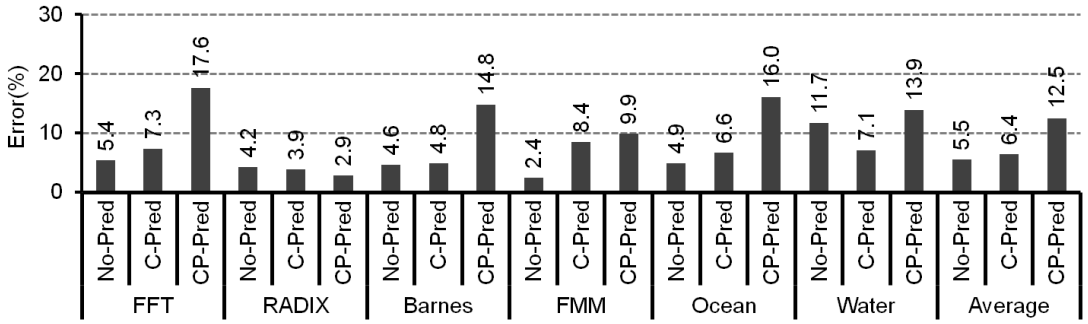


Figure 5.8: MPKI prediction error for S4 and 4–16MB shared LLCs.

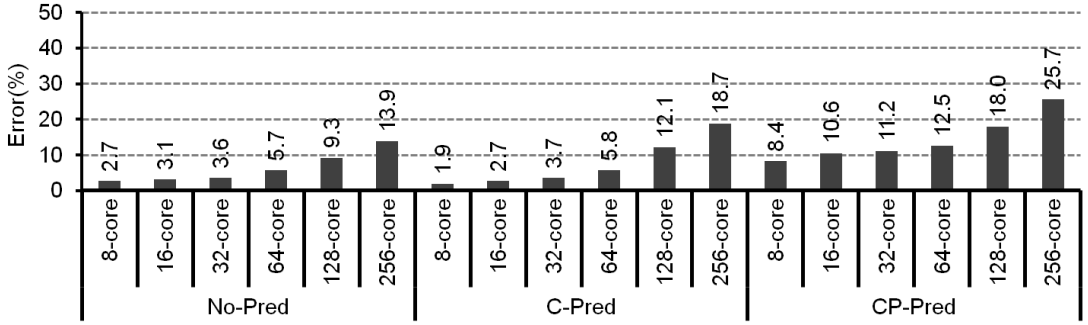
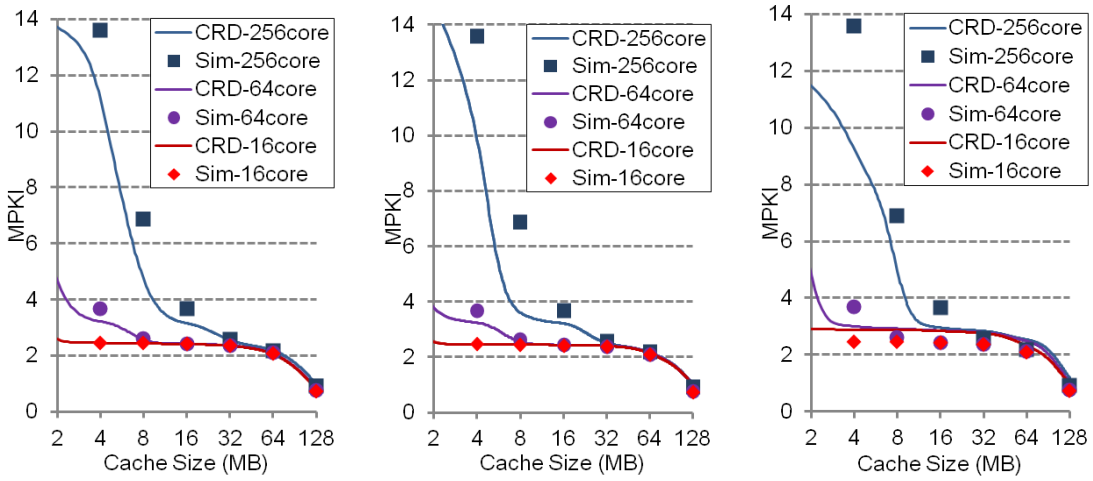


Figure 5.9: Prediction error for S4 and 4–16MB shared LLCs by core count.

count, reaching 13.9% for No-Pred, 18.7% for C-Pred, and 25.7% for CP-Pred at 256 cores. This illustrates that the cache conflict model errors mentioned earlier tend to increase with core count. Nevertheless, Figure 5.9 shows that prediction error at large core counts is still reasonable.

Figure 5.10 uses the FFT benchmark running at the S4 problem size as an example to present the predicted MPKI curves by using No-Pred, C-Pred, and CP-Pred. In Figure 5.10(a), the predicted MPKIs and simulated MPKIs are almost identical at 16 cores. When core count increases, the cache conflict model cannot predict cache misses accurately. Hence, the predicted MPKIs at 256 cores have higher errors. However, the predicted MPKI curves still capture the cache performance trends for core count scaling. C-Pred has the similar results, as illustrated in Figure 5.10(b). CP-Pred is the least accurate. In Figure 5.10(c), errors also happen



(a) No-Pred (b) C-Pred (c) CP-Pred

Figure 5.10: FFT's predicted LLC MPKI curves for No-Pred, C-Pred, and CP-Pred at the S4 problem size.

at 16 cores. However, the relative cache performance is correct, and the prediction results are still useful to study the impact of core count and problem size scaling.

Finally, we report the average MPKI difference (MPKI_{diff}) across all predictions for a particular prediction strategy and benchmark in Figure 5.11. MPKI_{diff} is calculated as $\frac{1}{N}\Sigma_{i=1}^{N}|MPKI_{measured} - MPKI_{predicted}|$, where N is the number of configurations. The rightmost group of bars shows averages across all 9 benchmarks. The MPKI_{diff} shows how close the predicted MPKI is to the simulated MPKI. No-Pred is able to predict within 0.1 MPKI for 8 out of 9 benchmarks, and within 0.28 MPKI for RADIX. On average, the MPKI_{diff} is 0.06 MPKI. Figure 5.11 also shows that C-Pred and CP-Pred are both less accurate than No-Pred. C-Pred can predict within 0.14 MPKI for 8 out of 9 benchmarks, and within 0.28 MPKI for RADIX. CP-Pred can predict within 0.19 MPKI for 7 out of 9 benchmarks, and within 0.26 MPKI for RADIX and 0.37 MPKI for Ocean. On average, prediction is within 0.10 MPKI. These results show that most of the predicted MPKIs via CRD profiles are

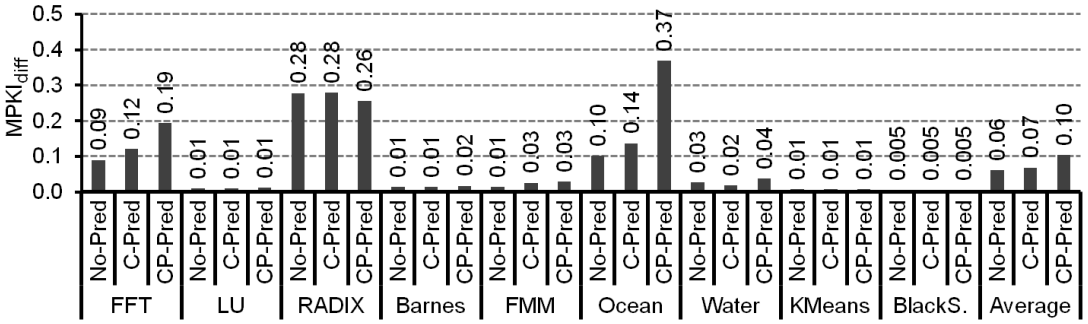


Figure 5.11: MPKI difference for shared LLC MPKI prediction.

very close to simulated MPKIs.

5.3.3 Private L2 Cache MPKI Prediction Accuracy

Figure 5.12 reports percent error with 1.0 offset for private L2 MPKI prediction. Each bar in Figure 5.12 reports the average percent error across all predictions for a particular prediction strategy and benchmark. The rightmost group of bars reports averages across all 9 benchmarks. We also predict coherence misses, which have infinite reuse distance. In our experiments, we assume that coherence misses increase proportionally with respect to core count and problem size.

As Figure 5.12 shows, No-Pred predicts private L2's MPKI within 15.4% of simulation for 9 benchmarks. Across all benchmarks, prediction error is 8.5%. These results reflect baseline prediction errors (*i.e.*, without profile prediction). Except the 3 error sources which are mentioned in Section 5.3.2, we also find that the instruction working set is replicated in private L2 caches, reducing the effective cache capacity for data working set. Hence, the error is usually large at small L2 capacities.

Figure 5.12 also shows that C-Pred and CP-Pred are less accurate than No-Pred. C-Pred and CP-Pred predict MPKI within 24.4% and 25.5% of simulation,

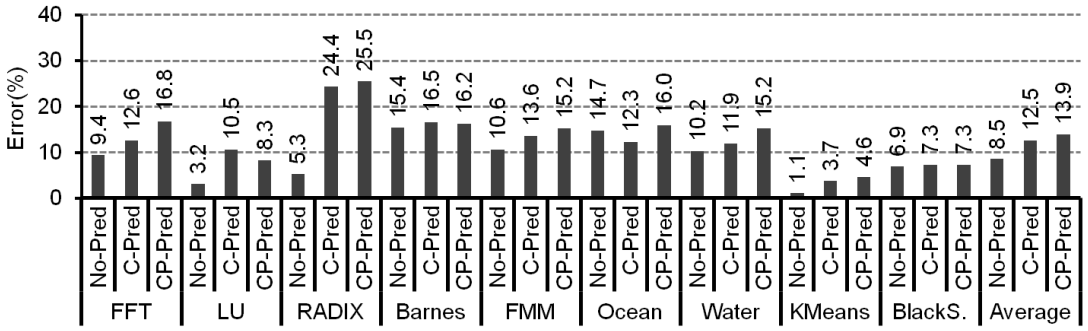


Figure 5.12: Percent private L2 MPKI prediction error with 1.0 offset.

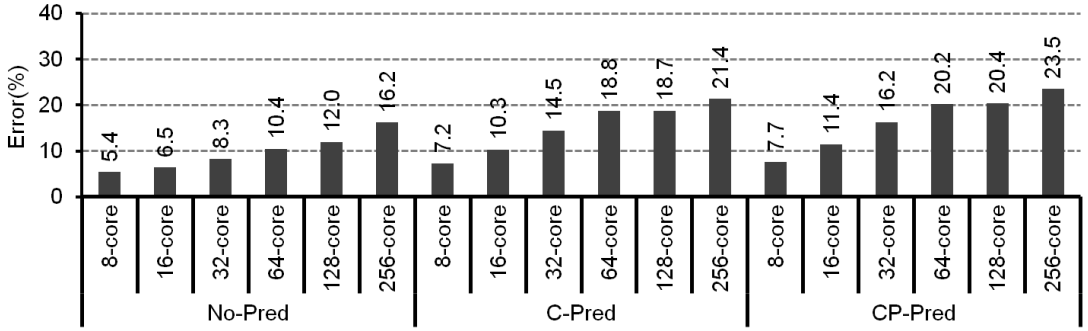
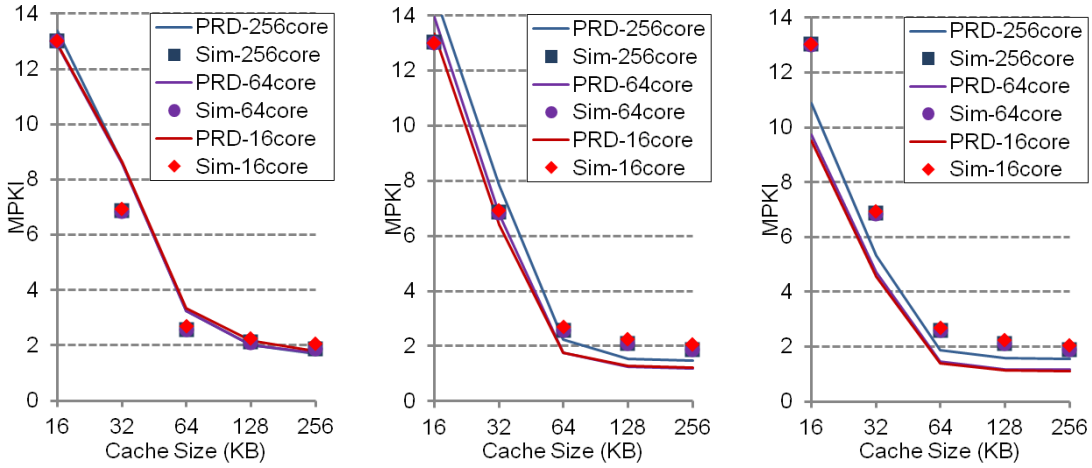


Figure 5.13: Prediction error by core count.

respectively. On average, C-Pred has 12.5% error and CP-Pred has 13.9% error. Like No-Pred, C-Pred and CP-Pred incur cache conflict model errors. But they also incur errors due to PRD profile prediction and coherence-miss prediction. In general, C-Pred and CP-Pred usually under-predict the amount of cache misses.

Figure 5.13 reports the prediction error broken down by prediction strategy and core count. For No-Pred, prediction error increases as core count scales, reaching 16.2% at 256 cores. C-Pred and CP-Pred have higher prediction error than No-Pred. Prediction error increases with core count, reaching 21.4% for C-Pred and 23.5% for CP-Pred at 256 cores. Nevertheless, Figure 5.13 shows that prediction error at large core counts is still reasonable.

Figure 5.14 uses the FFT benchmark running at the S4 problem size as an



(a) No-Pred (b) C-Pred (c) CP-Pred

Figure 5.14: FFT's predicted L2 MPKI curves for No-Pred, C-Pred, and CP-Pred at the S4 problem size.

example to present the predicted MPKI curves by using No-Pred, C-Pred, and CP-Pred. In Figure 5.14(a), the predicted MPKIs and simulated MPKIs are very similar, and No-Pred's predicted MPKI curves can capture the cache performance trend. C-Pred is less accurate, as illustrated in Figure 5.14(b). Prediction error increases with core count, but the cache performance trend is still valid. CP-Pred is the least accurate. However, the relative cache performance is correct, and the prediction is still useful to study the impact of core count and problem size scaling.

Figure 5.15 reports the MPKI_{diff} across all predictions for a particular prediction strategy and benchmark. No-Pred is able to predict within 0.54 MPKI difference for 8 out of 9 benchmarks, and within 2.01 MPKI for Ocean. Ocean has a large degree of error at 256 cores with 16KB L2 size. On average, the MPKI_{diff} is 0.49 MPKI. Figure 5.15 shows that C-Pred and CP-Pred are both less accurate than No-Pred. C-Pred can predict within 0.62 MPKI for 7 out of 9 benchmarks, and within 2.03 MPKI for RADIX and 1.77 MPKI for Ocean. CP-Pred can predict

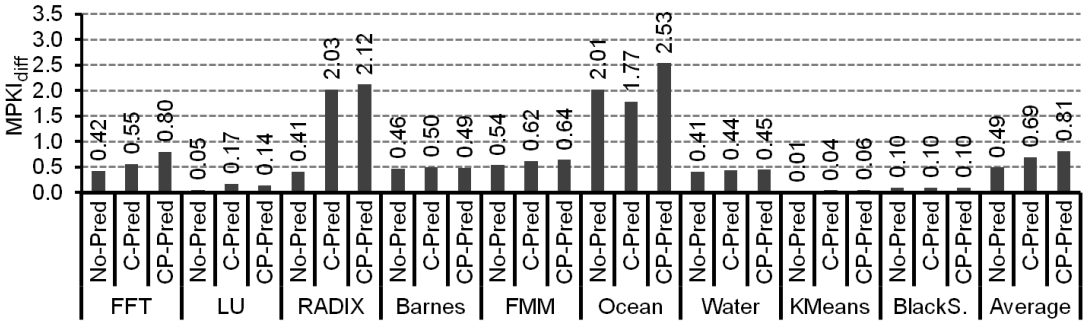


Figure 5.15: MPKI difference for private L2 MPKI prediction.

within 0.80 MPKI for 7 out of 9 benchmarks, and within 2.12 MPKI for RADIX and 2.53 MPKI for Ocean. On average, prediction is within 0.81 MPKI. These results show that private L2 MPKI prediction is less accurate then shared LLC MPKI prediction. However, the trend can still provide useful information to help us understand scaling impacts.

Overall, we find our prediction techniques for core count scaling can accelerate cache analysis without sacrificing accuracy. When combined with problem scaling prediction, analysis effort is further reduced, though error increases when predicting large core counts.

5.3.4 Sensitivity to Cache Associativity

In Section 5.3.2, the shared LLC is a 32-way set associative cache. In this section, we use the S2 problem size to study the impact of different cache associativities. Figure 5.16 reports percent shared LLC MPKI difference ($\frac{|MPKI_{32-way} - MPKI_{16-way}|}{MPKI_{32-way}}$) between 32-way and 16-way set associative caches. We add a small offset, 0.05, to the measured values, so the metric does not blow up. In Figure 5.16, the X-axis is broken down by core count and shared LLC size. The value of each bar is the

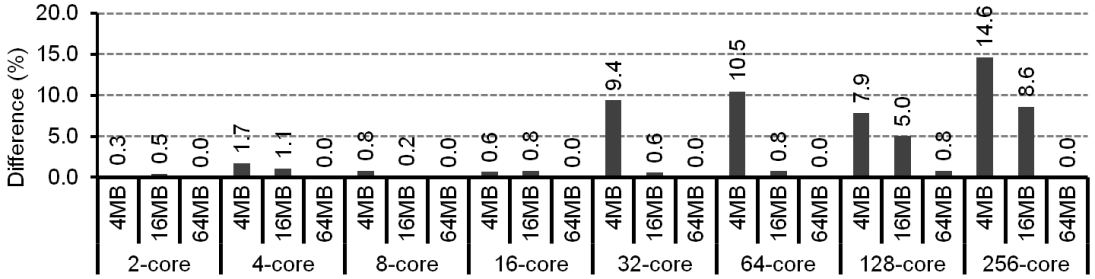


Figure 5.16: Shared LLC MPKI difference between 32-way and 16-way set associative LLCs by core count and cache capacity.

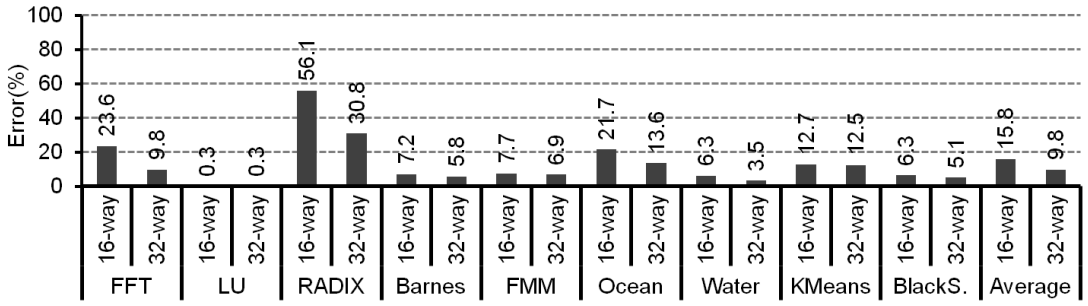
average percent difference across our 9 benchmarks.

Figure 5.16 shows that the MPKI difference between 32-way and 16-way set associative caches is large at large core counts and small cache sizes. For example, the MPKI difference for 256 cores and 4MB cache achieves 14.6%. As a result, when the cache capacity is small or the number of cores is large, lower associativity usually has a significant impact on cache performance.

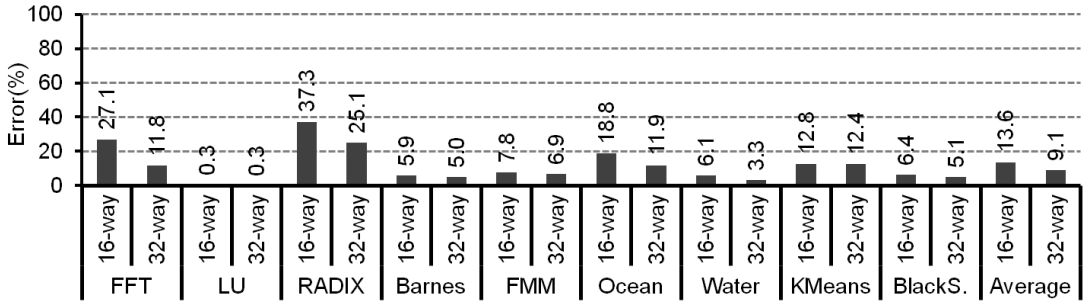
Figure 5.17 illustrates MPKI prediction error for the S2 problem size with a small offset, 0.05. Each bar in Figure 5.17 reports the average percent error across all predictions for a particular prediction strategy, benchmark, and associativity. The rightmost group of bars reports averages across all 9 benchmarks.

As Figure 5.17(a) shows, for 16-way set associative LLCs, No-Pred is able to predict shared LLC's MPKI within 23.6% of simulation for 8 out of 9 benchmarks, and within 56.1% for RADIX. On average, prediction error is 15.8%. On the other hand, for 32-way set associative LLCs, No-Pred is able to predict shared LLC's MPKI within 13.6% of simulation for 8 out of 9 benchmarks, and within 30.8% for RADIX. On average, prediction error is 9.8%.

No-Pred has lower prediction accuracy for 16-way set associative LLCs. Upon



(a) No-Pred



(b) C-Pred

Figure 5.17: MPKI prediction error for S2 and 4–128MB 16-way/32-way set associative LLCs.

closer examination, there are two main sources of error. First, we find one of the main sources of error is the cache conflict model. In our results, 16-way set associative LLCs have more cache misses than those of 32-way set associative LLCs. Although the cache conflict model also predicts more cache misses for 16-way set associative LLCs, the cache conflict model has higher prediction error for 16-way set associative LLCs at small cache sizes and large core counts. And Second, our error metric does not always address numeric instability. If we change the offset to be 0.5, No-Pred achieves 16.9% and 11.3% error for RADIX with 16-way and 32-way set associative LLCs, respectively.

In Figure 5.17(b), for 16-way set associative LLCs, C-Pred is able to predict shared LLC's MPKI within 27.1% of simulation for 8 out of 9 benchmarks, and

within 37.3% for RADIX. On average, prediction error is 13.6%. On the other hand, for 32-way set associative LLCs, C-Pred is able to predict MPKI within 12.4% of simulation for 8 out of 9 benchmarks, and within 25.1% for RADIX. On average, prediction error is 9.1%. These results show that C-Pred has errors similar to those of No-Pred. The reasons are discussed in Section 5.3.2.

As Figure 5.17 shows, the prediction accuracy of 16-way set associative LLCs is about 5% worse than the prediction accuracy of 32-way set associative LLCs. The major reason is that the cache conflict model has higher prediction error for 16-way set associative LLCs at small cache sizes and large core counts, especially at 4MB and 256 cores. However, these results show that CRD profiles can also be used to study the cache performance of set associative caches without sacrificing prediction accuracy too much. As a result, our prediction techniques can be used to study many different architecture designs quickly.

Chapter 6

Optimizing Multicore Cache Hierarchies Using Reuse Distance Analysis

CRD profiles and PRD profiles present an application's memory behavior for shared caches and private caches. This suggests we can use these profiles to study and identify the optimal cache hierarchy. In this chapter, we develop a novel framework that employs whole-program CRD and PRD profiles to study the trade-offs of multicore cache system design. We also study how core count scaling and problem size scaling impact the optimal cache hierarchy.

6.1 Performance Models

To study different cache hierarchies, we select the tiled architecture due to its scalability. Figure 5.1 depicts an example tiled CMP. Each tile contains a core, a private L1 cache, a private L2 cache, and an LLC module. The LLC module can either be a private cache, or a slice of a shared cache. Tiles are connected by a 2D mesh network.

In our study, we assume caches are inclusive and allow data blocks to be replicated in private caches. Hence, when a cache miss happens in the L1 and L2 caches, the cache sends a request to the next-level cache directly. However, when a cache miss happens in the private LLC, the coherence protocol first checks the

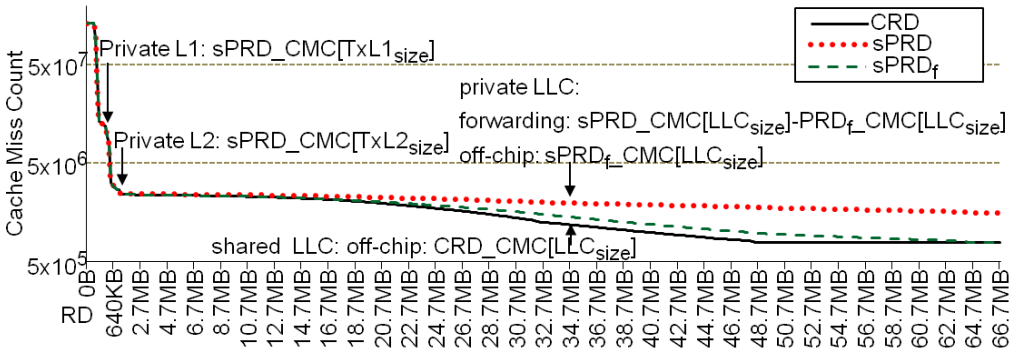


Figure 6.1: CRD_CMC, sPRD_CMC, and sPRD$_f$_CMC profiles for FFT running on 16 cores at the S3 problem size.

on-chip directory. If the cache block resides in the private LLC of the other tiles, the cache block will be forwarded to the requesting tile. Otherwise, the request will be sent to the off-chip DRAM. To model the data-forwarding in the private LLC, we need another profile (PRD$_f$) to indicate whether the cache block resides in the other tiles or not. To compute PRD$_f$ profiles, we find the minimum reuse distance among per-thread stacks for a given memory access. The intuition is that this minimum RD is the smallest cache size that contains the replication of a cache block. The collected RD values form the PRD data-forwarding profile, PRD$_f$.

Figure 6.1 uses FFT running on 16 cores at the S3 problem size to summarize how we compute the cache misses at each cache level. For private L1 and L2 caches, the total cache misses are $sPRD\_CMC[T \times L1_{size}]$ and $sPRD\_CMC[T \times L2_{size}]$. $L1_{size}$ and $L2_{size}$ are the private L1 and L2 cache sizes per core, and T is the number of tiles. For the private LLC with total LLC_{size} capacity, the directory-access traffic is $sPRD\_CMC[LLC_{size}]$, and the access latency is $(DIR_{lat} + HOP_{lat})$. We assume data blocks are distributed uniformly on the LLC slices, so network messages incur $\sqrt{T} + 1$ hops on average [37]. Hence, HOP_{lat} is per-hop latency $\times$ average hops.

After checking the directory, the data-forwarding traffic is ($sPRD\_CMC\ [LLC_{size}] - sPRD_f\_CMC[LLC_{size}]$), and the access latency is ($LLC_{lat} + 2 \times HOP_{lat}$) which contains two-way data forwarding communication and one cache access to acquire the data. The off-chip traffic is $sPRD_f\_CMC[LLC_{size}]$. We also model the two-way communication when accessing the memory controller. Hence, the average memory access time (AMAT) for the tiled processors with private LLCs can be modeled using Equation 6.1, where $sPRD\_CMC[0]$ is the number of total memory references.

$$
\begin{aligned}
AMAT_p = {} & L1_{lat} + L2_{lat} \times \frac{sPRD\_CMC[T \times L1_{size}]}{sPRD\_CMC[0]} \\
& + LLC_{lat} \times \frac{sPRD\_CMC[T \times L2_{size}]}{sPRD\_CMC[0]} \\
& + (DIR_{lat} + HOP_{lat}) \times \frac{sPRD\_CMC[LLC_{size}]}{sPRD\_CMC[0]} \\
& + (LLC_{lat} + 2 \times HOP_{lat}) \times \frac{sPRD\_CMC[LLC_{size}] - sPRD_f\_CMC[LLC_{size}]}{sPRD\_CMC[0]} \\
& + (DRAM_{lat} + 2 \times HOP_{lat}) \times \frac{sPRD_f\_CMC[LLC_{size}]}{sPRD\_CMC[0]}
\end{aligned}
\tag{6.1}
$$

For shared LLCs, the LLC accesses are $sPRD\_CMC[T \times L2_{size}]$ with access latency ($LLC_{lat} + 2 \times HOP_{lat}$). In shared LLCs, the data always resides on the home tile, so there is two-way communication. The off-chip cache misses are $CRD\_CMC[LLC_{size}]$. The average memory access time (AMAT) for the tiled processors with shared LLCs can be modeled using Equation 6.2.

$$
\begin{aligned}
AMAT_s = & L1_{lat} + L2_{lat} \times \frac{sPRD\_CMC[T \times L1_{size}]}{sPRD\_CMC[0]} \\
& + (LLC_{lat} + 2 \times HOP_{lat}) \times \frac{sPRD\_CMC[T \times L2_{size}]}{sPRD\_CMC[0]} \\
& + (DRAM_{lat} + 2 \times HOP_{lat}) \times \frac{CRD\_CMC[LLC_{size}]}{sPRD\_CMC[0]}
\end{aligned} \tag{6.2}
$$

These two simple performance models do not consider queuing in the on-chip and off-chip networks, but they provide insights about an application's cache performance on different cache hierarchies.

6.2 Performance Analysis

At the same cache capacity, the shared cache has the best on-chip miss-rate, but it has longer access latency. In contrast, the private cache has the worst miss-rate, but it keeps data locally. In this section, we first study when shared LLCs perform better than private LLCs. Then we study the trade-off between L2 and LLC capacities. We also study how core count scaling and problem size scaling impact the cache system design.

6.2.1 Private vs. Shared LLCs

Shared LLCs are better than private LLCs when $AMAT_p > AMAT_s$. Given Equation 6.1 and Equation 6.2, this occurs when:

$$(DRAM_{lat} + 2 \times HOP_{lat}) \times (sPRD_f\_CMC[LLC_{size}] - CRD\_CMC[LLC_{size}])$$
$$> (2 \times HOP_{lat}) \times sPRD\_CMC[T \times L2_{size}] -$$
$$((DIR_{lat} + HOP_{lat}) \times sPRD\_CMC[LLC_{size}] +$$
$$(LLC_{lat} + 2 \times HOP_{lat}) \times (sPRD\_CMC[LLC_{size}] - sPRD_f\_CMC[LLC_{size}])) \quad (6.3)$$

Equation 6.3 shows that shared LLCs are better when the total off-chip memory stall saved via $sPRD_f\_CMC/CRD\_CMC$ gap in shared LLCs (the LHS of Equation 6.3) exceeds the total on-chip memory stall saved in private LLCs (the RHS of Equation 6.3). The shared LLC's on-chip access latency is weighted by the LLC access frequency–*i.e.*, the L2's misses. Hence, the choice between private and shared LLCs not only depends on the program behavior, but it also depends on L2 and LLC capacities.

To illustrate, Figure 6.2 plots AMAT_p and AMAT_s as a function of total LLC size for the FFT benchmark running on 16 cores at the S3 problem size. Different pairs of curves show results for different L2 sizes. When computing AMAT, we assume an 8KB L1 cache with 1-cycle latency, 4-cycle L2 latency, 10-cycle LLC latency, 10-cycle directory latency, 200-cycle DRAM latency, and 3-cycle per-hop network latency. Figure 6.2 shows that the choice between private and shared LLCs depends on two major effects. The first effect is L2 cache capacity. At small L2 capacities, the first term in the RHS of Equation 6.3 always dominates due to the

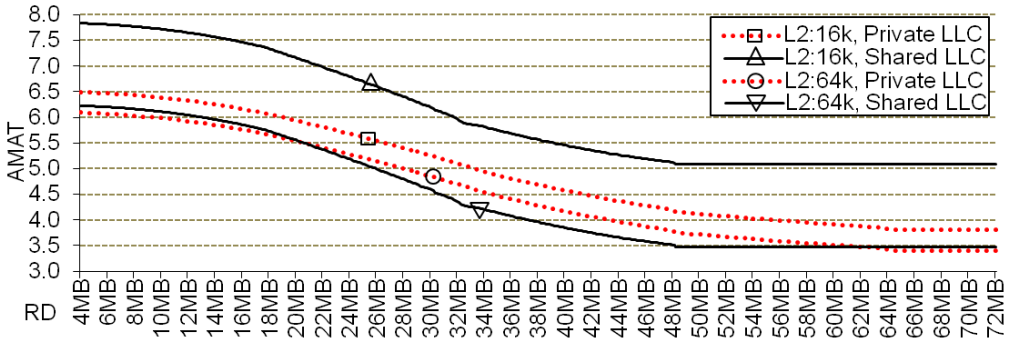


Figure 6.2: FFT's $AMAT_p$ and $AMAT_s$ for different L2 and LLC capacities.

high shared LLC accesses. So private LLCs are always better. This occurs in Figure 6.2 when the L2 is 16KB. For larger L2 caches, the first term in the RHS of Equation 6.3 reduces, and it allows the LHS of Equation 6.3 to dominate when the $sPRD_f$_CMC/CRD_CMC gap is sufficiently large. This occurs in Figure 6.2 for the 64KB L2 with a private-to-shared LLC cross-over at 20.4MB. Finally, at large private LLC capacities that can contain replications, $CRD\_CMC[LLC_{size}]$, and $sPRD_f\_CMC[LLC_{size}]$ are almost identical. As a result, the LHS of Equation 6.3 is close to 0 again. This diminishes the advantage of shared LLCs. In fact, private LLCs may regain a performance advantage when the total on-chip memory stall in shared LLCs is higher than the total on-chip memory stall in private LLCs. This occurs in Figure 6.2 for the 64KB L2 with a shared-to-private LLC cross-over at 62.0MB.

6.2.2 Scaling Private-vs-Shared LLCs

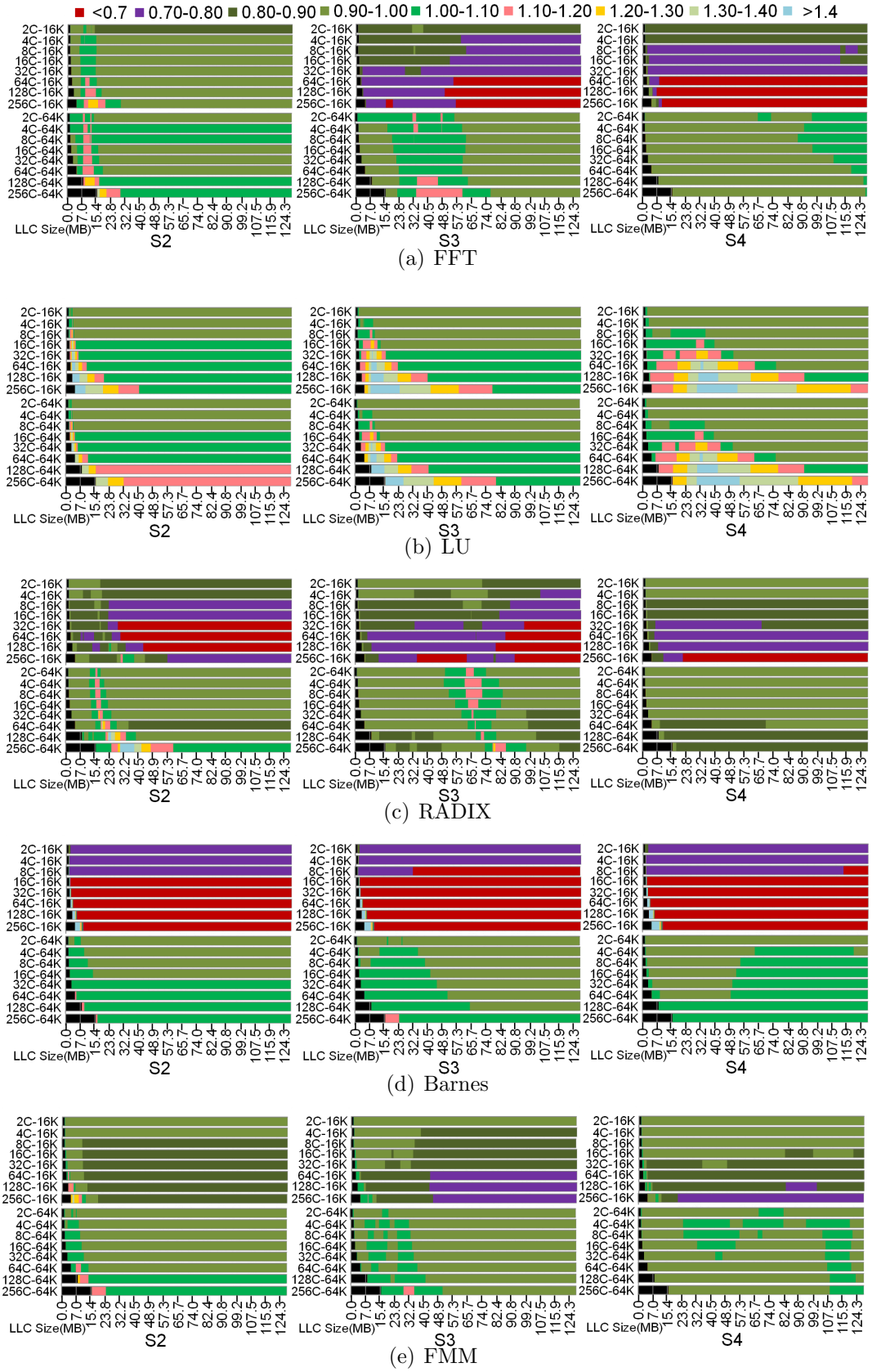
In this section, we extend the architecture insights from Section 6.2.1 by incorporating the core count scaling and problem size scaling effects discussed in Sec-

tion 3.5. Figure 6.3 and Figure 6.4 present the results for our 9 benchmarks. In Figure 6.3 and Figure 6.4, we plot three problem sizes, S2–S4, per benchmark. Each problem size has two graphs: the top graph shows tiled CMPs with 8KB L1 caches and 16KB L2 caches, while the bottom graph shows tiled CMPs with 8KB L1 caches and 64KB L2 caches. Within each graph, the LLC capacity is varied from 0–128MB along the X-axis. The core count scaling is studied along the Y-axis for 2–256 cores. For each CMP configuration, the ratio $\frac{AMAT_p}{AMAT_s}$ is plotted at different colors. We do not consider CMPs with less total LLC capacity than total L2 capacity. These cases are shaded black in Figure 6.3 and Figure 6.4.

All basic insights from Figure 6.2 are also visible in Figure 6.3 and Figure 6.4. Shared LLCs are best only when conditions make the LHS of Equation 6.3 dominate. Hence, the L2 capacity must be sufficiently large to reduce LLC access frequency. The total off-chip memory stall saved in shared LLCs must also be greater than the total on-chip memory stall saved in private LLCs.

For our benchmarks and the tiled CMP configurations, 16KB L2 is usually not large enough for FFT, RADIX, Barnes, FMM, Ocean, and BlackScholes to reduce LLC access frequency; so private LLCs are usually best in these cases. Increasing the L2 cache size can benefit shared LLCs. We also see that most configurations for which shared LLCs are best occur around or beyond their corresponding C_{share} value in Figure 3.15(b).

Because the CRD_CMC/PRD_f_CMC gap varies across the LLC capacity, preference may change from private LLC to shared LLC and back to private LLC again. FFT, LU, RADIX, Barnes, FMM, and Water show this behavior. Kmeans



(a) FFT

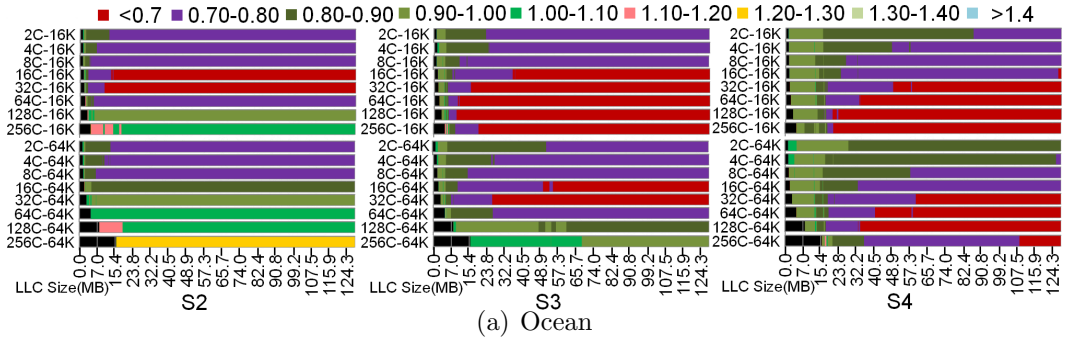
(b) LU

(c) RADIX

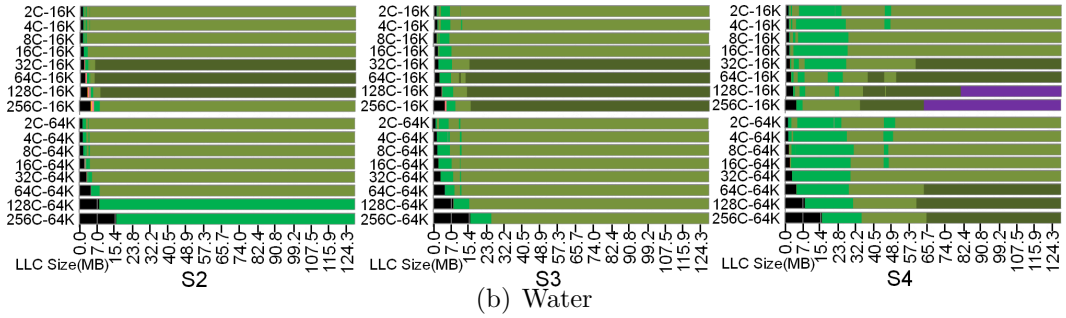
(d) Barnes

(e) FMM

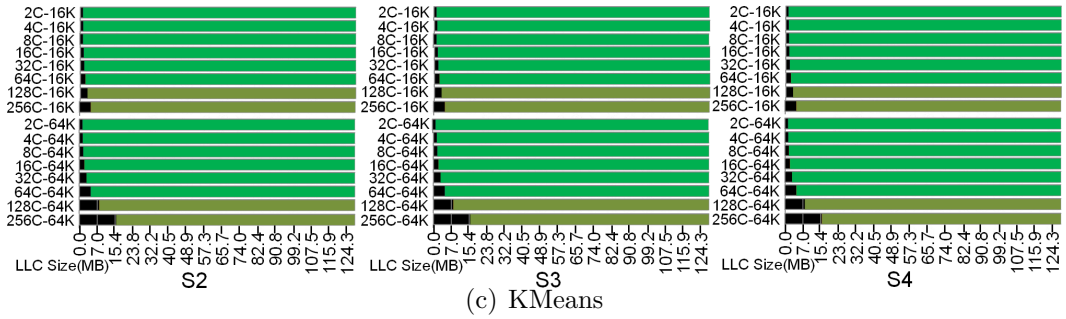
Figure 6.3: Private vs. shared LLC performance across L2 capacity, LLC capacity, core count, and problem size for FFT, LU, RADIX, Barnes, and FMM.



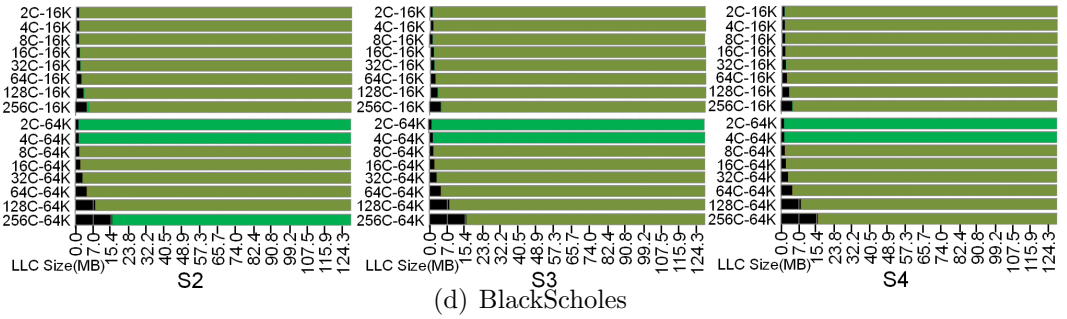
(a) Ocean



(b) Water



(c) KMeans



(d) BlackScholes

Figure 6.4: Private vs. shared LLC performance across L2 capacity, LLC capacity, core count, and problem size for Ocean, Water, KMeans, and BlackScholes.

and BlackScholes have very good data locality. Hence, LLC capacity doesn't affect the preference of private LLCs or shared LLCs.

Figure 6.3 and Figure 6.4 also show the impact of core count scaling on the private vs. shared LLCs selection. As described in Section 3.3, both CRD_CMC and sPRD_CMC profiles shift coherently with core count scaling at small RD, which can increase cache capacity pressure on L1 and L2 caches. As we add cores to our tiled CMP, we also increase total L1 and L2 cache capacity linearly, and this cancels the effect of the shift. However, core count scaling also increases the average communication hops, making shared LLC accesses more costly than private LLC accesses. These scaling trends tend to make private caches more desirable for larger core counts. For example, FFT, RADIX, FMM, Ocean, KMeans, and BlackScholes show this effect at the S4 problem size. On the other hand, core count scaling also shifts C_{share} to smaller RD and increases the gap between sPRD_CMC and CRD_CMC profiles, as illustrated in Figure 3.14. These scaling trends tend to make shared caches more desirable for larger core counts when the gap is large. For example, LU and Barnes show this effect at the S2-S4 problem sizes. Hence, the choice between private LLCs and shared LLCs is highly application- and architecture-dependent.

Lastly, Figure 6.3 and Figure 6.4 show the impact of problem size scaling on the private vs. shared LLCs selection. As described in Section 3.5, both C_{core} and C_{share} move to larger RD values as problem size scales. So the region where shared caches are best tends to move to larger cache capacity. For example, FFT, LU, Radix, Barnes, FMM, and Water show this effect. In addition, large problem size also causes higher pressure on the cache, and 64KB L2 might not be large enough

to reduce the cost of shared LLC access. As the problem size increases, private LLC usually prevails. We see this effect in FFT, Radix, FMM, and Ocean.

6.2.3 Trade-off Between L2 and LLC Capacities

In the previous section, we find that the L2 capacity has significant impact on cache performance. In this section, we study the trade-off between L2 and LLC capacities when the on-chip cache capacity (C) is fixed (*i.e.*, $T \times L2_{size} + LLC_{size} = C$) and $T \times L2_{size} \leq LLC_{size}$. We assume L1 is closely coupled with the core, so its size is fixed. The trade-off between L2 and LLC capacities impacts the balance between on-chip and off-chip traffic. For a fixed capacity, there exists an optimal [$L2_{size}$,LLC_{size}] point. In this section, we examine how different scaling schemes impact this optimal point.

For a tiled CMP with the constraints, $T \times L2_{size} + LLC_{size} = C$ and $LLC_{size} \geq T \times L2_{size}$, the $LLC_{size,opt}$ exists when the change in the LLC size causes $\triangle\ AMAT_p \geq 0$ and $\triangle\ AMAT_s \geq 0$. After replacing $T \times L2_{size}$ by $C - LLC_{size}$ in Equation 6.1 and Equation 6.2, Equation 6.4 and Equation 6.5 show the inequalities for $\triangle\ AMAT_p \geq 0$ and $\triangle\ AMAT_s \geq 0$.

$$
\begin{aligned}
&\triangle AMAT_p \geq 0 \\
&\Rightarrow (LLC_{lat}) \times (sPRD\_CMC[C - LLC_{size}] - sPRD\_CMC[C - LLC_{size,opt}]) \\
&+ (DIR_{lat} + HOP_{lat}) \times (sPRD\_CMC[LLC_{size}] - sPRD\_CMC[LLC_{size,opt}]) \\
&+ (LLC_{lat} + 2 \times HOP_{lat}) \times \\
&\quad ((sPRD\_CMC[LLC_{size}] - sPRD_f\_CMC[LLC_{size}]) \\
&\quad - (sPRD\_CMC[LLC_{size,opt}] - sPRD_f\_CMC[LLC_{size,opt}])) \\
&\geq (DRAM_{lat} + 2 \times HOP_{lat}) \times \\
&\quad (sPRD_f\_CMC[LLC_{size,opt}] - sPRD_f\_CMC[LLC_{size}])
\end{aligned}
\tag{6.4}
$$

$$
\begin{aligned}
&\triangle AMAT_s \geq 0 \\
&\Rightarrow (LLC_{lat} + 2 \times HOP_{lat}) \times \\
&\quad (sPRD\_CMC[C - LLC_{size}] - sPRD\_CMC[C - LLC_{size,opt}]) \\
&\geq (DRAM_{lat} + 2 \times HOP_{lat}) \times (CRD\_CMC[LLC_{size,opt}] - CRD\_CMC[LLC_{size}])
\end{aligned}
\tag{6.5}
$$

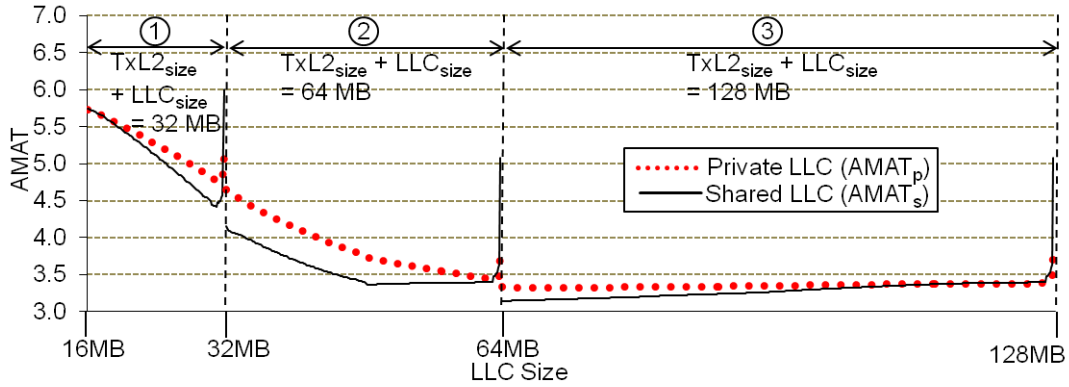
Equation 6.4 and Equation 6.5 show that there exists an optimal LLC cache capacity–*i.e.*, $LLC_{size,opt}$. For all $LLC_{size} > LLC_{size,opt}$, the increase of on-chip memory stall is greater than the decrease of off-chip memory stall. For all $LLC_{size} < LLC_{size,opt}$, the increase of off-chip memory stall is greater than the decrease of on-chip memory stall. As a result, $LLC_{size,opt}$ exists when *the change of the total on-chip*

memory stall is equal to the change of the total off-chip memory stall.

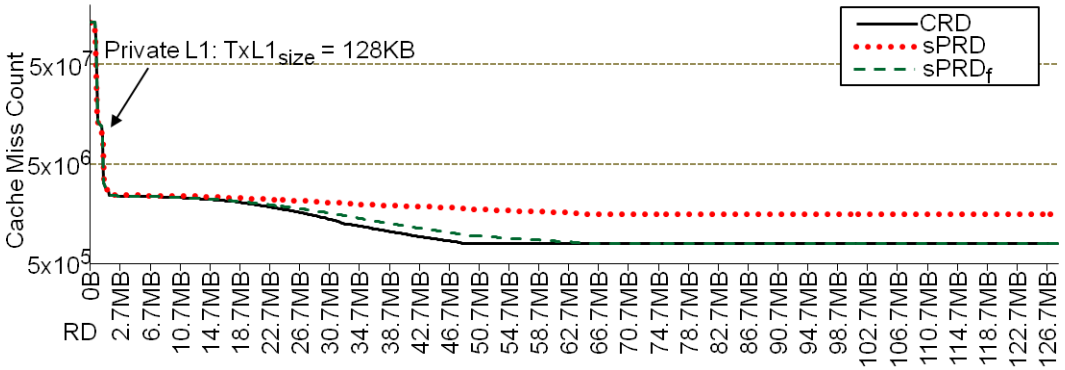
Figure 6.5(a) plots AMAT_p and AMAT_s as a function of LLC_{size} for the FFT benchmark running on 16 cores at the S3 problem size. Figure 6.5(b) plots the corresponding CRD_CMC, sPRD_CMC, and sPRD_f_CMC profiles to explain the insights. Figure 6.5(a) is divided into three regions to represent three total cache capacities, marked as ①$T \times L2_{size} + LLC_{size} = 32MB$, ②$T \times L2_{size} + LLC_{size} = 64MB$, and ③$T \times L2_{size} + LLC_{size} = 128MB$. For each region, the leftmost point represents $T \times L2_{size} = LLC_{size} = \frac{1}{2}C$. As LLC_{size} increases along the X-axis, $L2_{size}$ decreases as $T \times L2_{size} = C - LLC_{size}$. The rightmost point represents $L2_{size} = 2 \times L1_{size}$ and $LLC_{size} = C - T \times L2_{size}$. In the graph, the dotted lines represent AMAT_p and the solid lines represent AMAT_s.

Figure 6.5(a) provides three major insights that are valid for all of our benchmarks. First, when L2_{size} is close to L1_{size} (8KB in our study and marked in Figure 6.5(b)), high LLC accesses cause high AMAT. AMAT_s is higher than AMAT_p at small L2_{size}. This is because shared LLCs have a higher on-chip communication cost than private LLCs (the first term in the RHS of Equation 6.3 always dominates).

Second, as L2_{size} increases, the AMAT drops rapidly. When L2_{size} is large enough to capture the major working set, further increasing L2_{size} doesn't reduce LLC accesses significantly, as illustrated in Figure 6.5(b). Hence, off-chip traffic grows as L2_{size} keeps increasing, and the AMAT goes up again. There exists an optimal LLC_{size} which has the lowest AMAT. In Figure 6.5(a), region ① and ② show this behavior. For AMAT_s (AMAT_p), the optimal LLC_{size} are 30.9MB (30.9MB) and 48.4MB (62.9MB) in region ① and ②, respectively.



(a) The trade-off between L2 and L3 cache capacities.



(b) CRD_CMC, sPRD_CMC, and sPRD$_f$_CMC profiles.

Figure 6.5: The trade-off between L2$_{size}$ and LLC$_{size}$ for FFT running on 16 cores at the S3 problem size.

In region ①, when LLC$_{size}$ is between 17.7MB and 31.7MB, shared LLCs always outperform private LLCs. This is because the gap between CRD_CMC and sPRD$_f$_CMC profiles causes the total off-chip memory stall saved in shared LLCs to be higher than the total on-chip memory stall saved in private LLCs. Decreasing the LLC$_{size}$ reduces the gap, and AMAT$_s$ approaches AMAT$_p$. Finally, private LLCs outperform shared LLCs at 17.5MB. In region ②, shared LLCs always outperform private LLCs between 32MB and 62.9MB LLC due to the gap between CRD_CMC and sPRD$_f$_CMC profiles. Because CRD_CMC also decreases faster than sPRD_CMC inside this range, increasing L2$_{size}$ has more benefit for shared

LLCs. Hence, the shared $\text{LLC}_{size,opt}$ is smaller than the private $\text{LLC}_{size,opt}$.

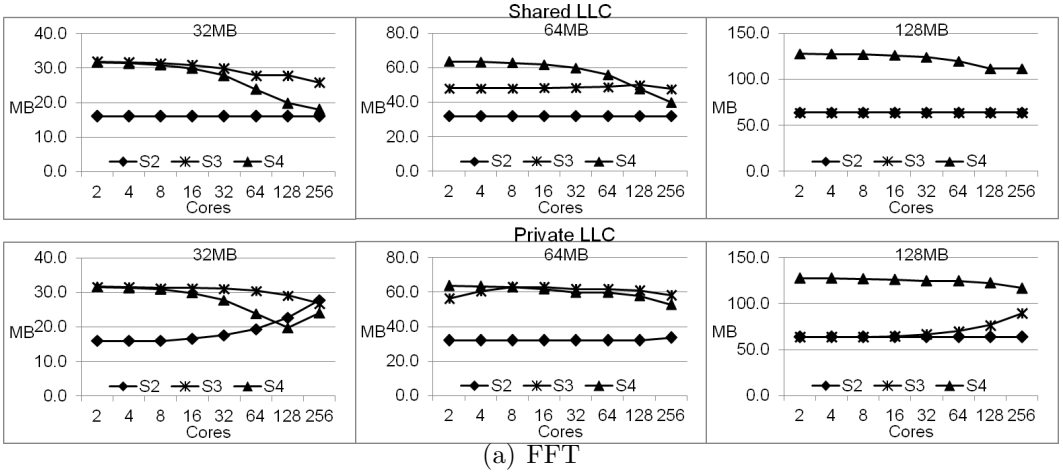
Lastly, when the LLC capacity is large enough (region ③ in Figure 6.5(a)), the LLC misses only change slightly with respect to LLC_{size}, as illustrated in Figure 6.5(b). The decrease of LLC_{size} doesn't impact the off-chip traffic significantly, but the increase of L2_{size} reduces the number of LLC cache accesses. Hence, the $\text{LLC}_{size,opt}$ is close to $LLC_{size} = T \times L2_{size} = \frac{1}{2}C$. For AMAT_s (AMAT_p), the $\text{LLC}_{size,opt}$ is 64.0MB (64.7MB) in Figure 6.5(a). Shared LLCs also outperform private LLCs between 64MB and 115.5MB, because the total on-chip memory stall in private LLCs is high.

Figure 6.6, Figure 6.7, and Figure 6.8 show the $\text{LLC}_{size,opt}$ for our 9 benchmarks. In Figure 6.6, Figure 6.7, and Figure 6.8, we plot three $T \times L2_{size} + LLC_{size}$, 32MB, 64MB, and 128MB per benchmark. Each cache capacity has two graphs: the top graph shows the shared $\text{LLC}_{size,opt}$, while the bottom graph shows the private $\text{LLC}_{size,opt}$. For each graph, the X-axis is the number of cores, and the Y-axis is the corresponding $\text{LLC}_{size,opt}$. We report the $\text{LLC}_{size,opt}$ for three problem sizes, S2–S4.

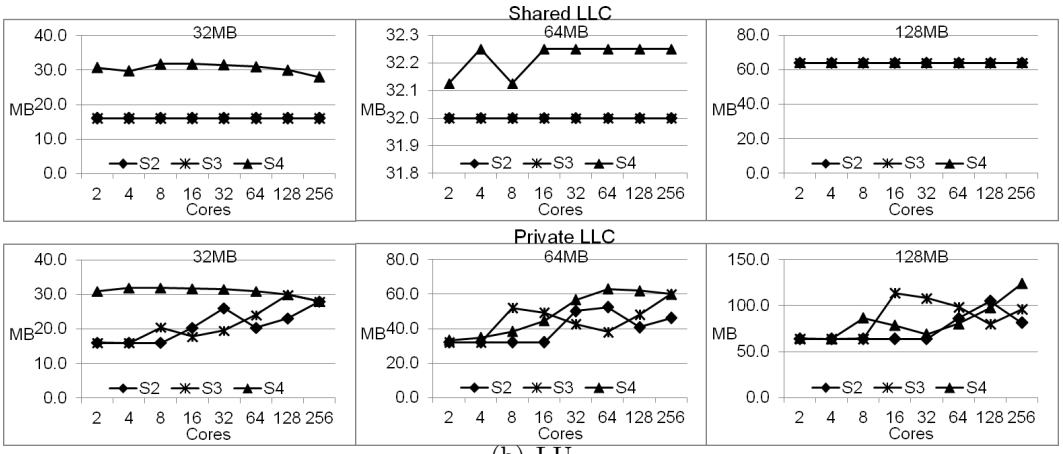
All basic insights from Figure 6.5 can be applied to Figure 6.6–Figure 6.8. First, for both shared and private $\text{LLC}_{size,opt}$, the L2_{size} must be sufficiently large to reduce LLC access frequency. Second, the optimal LLC_{size} depends on the balance between on-chip memory stall and off-chip memory stall. Because the $\text{LLC}_{size,opt}$ shows different behaviors for shared and private LLCs, we discuss these two cases separately.

Shared LLCs

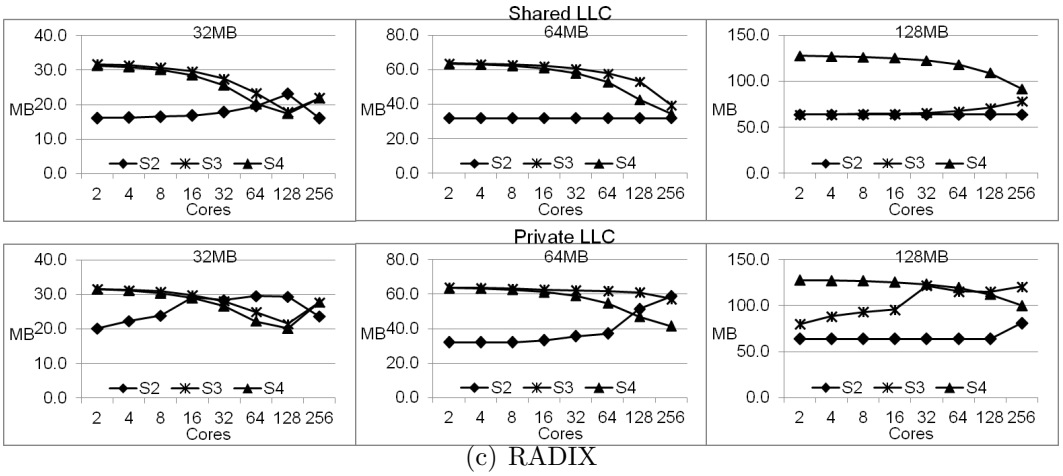
As described in Section 3.3, core count scaling only impacts cache performance



(a) FFT

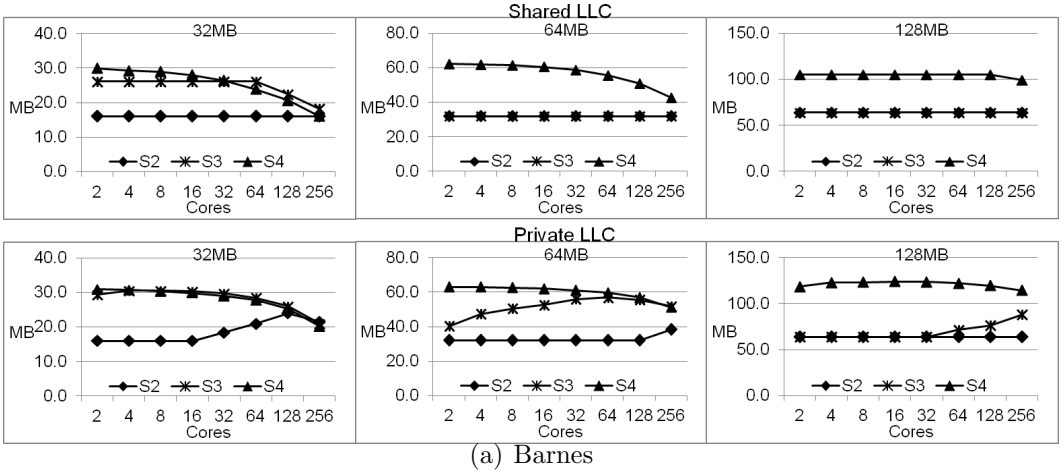


(b) LU

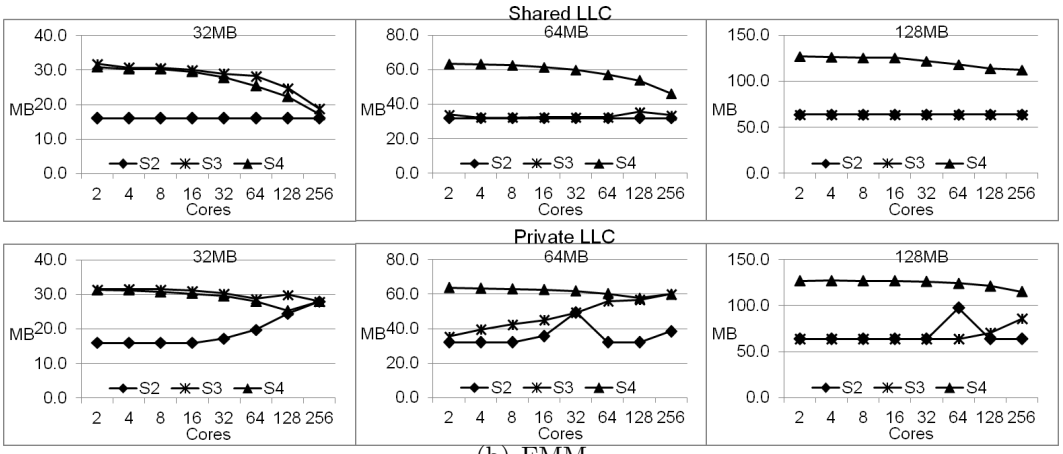


(c) RADIX

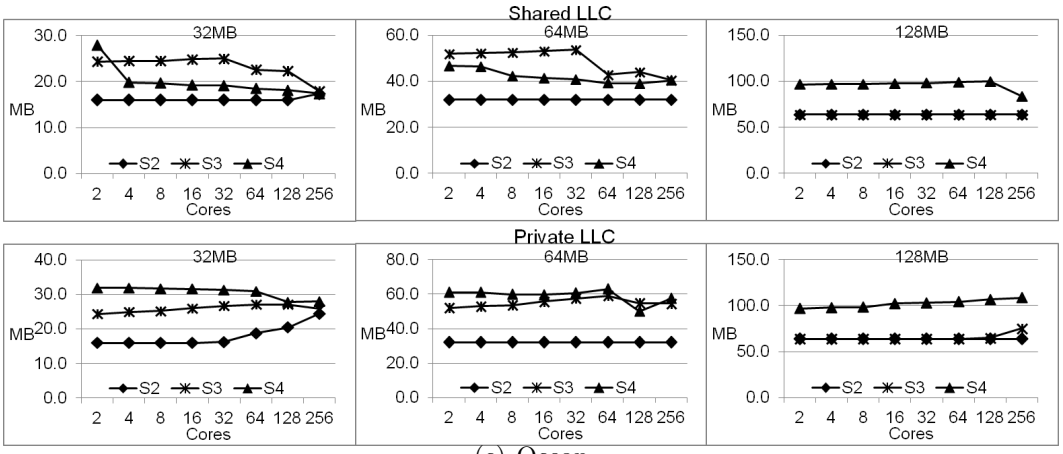
Figure 6.6: Optimal LLC_{size} at different problem sizes (S2-S4), total cache sizes (32M-128M), and the number of cores for FFT, LU, and RADIX.



(a) Barnes

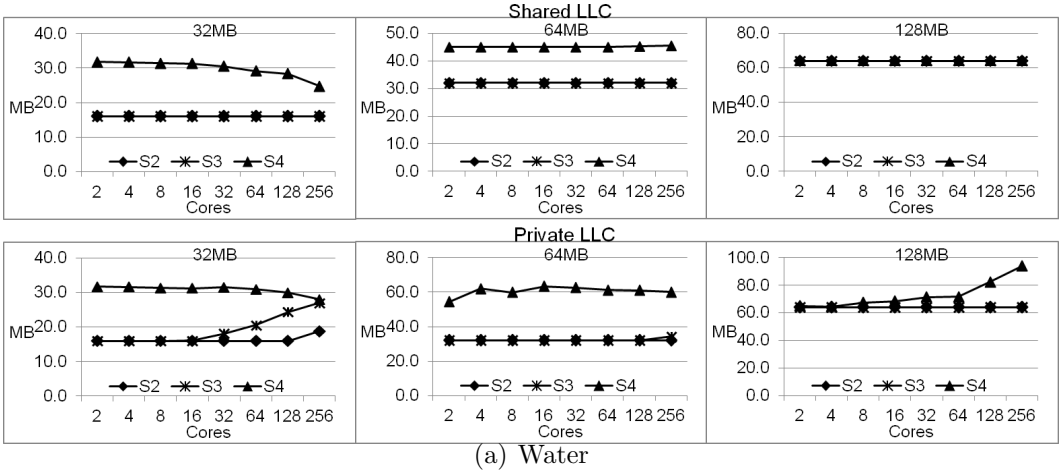


(b) FMM

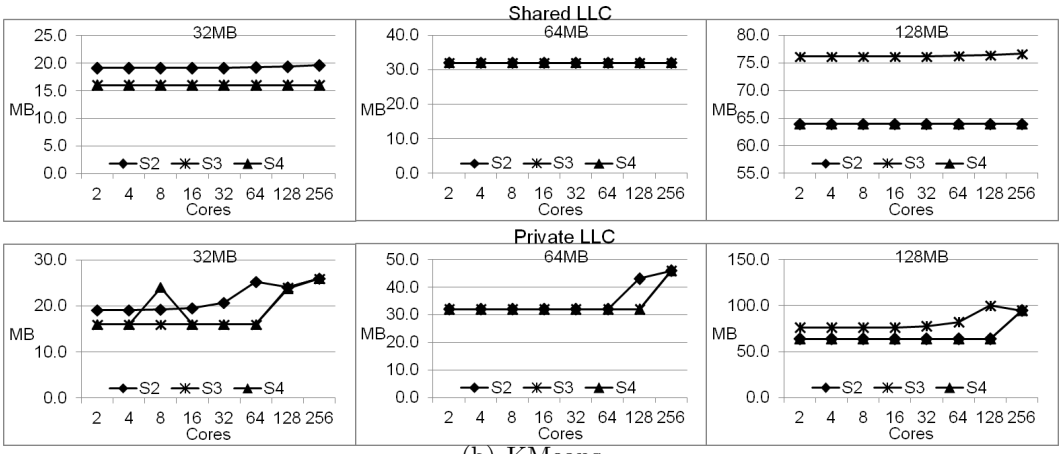


(c) Ocean

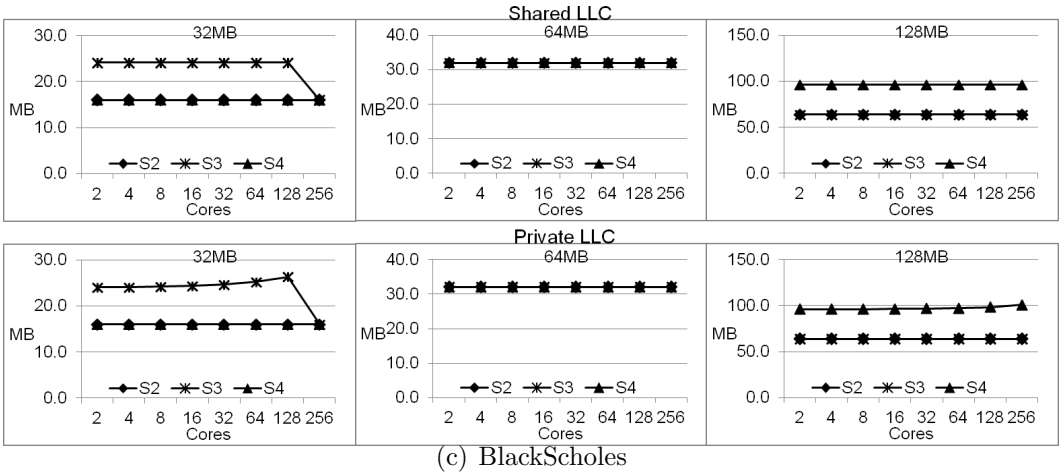
Figure 6.7: Optimal LLC_{size} at different problem sizes (S2-S4), total cache sizes (32M-128M), and the number of cores for Barnes, FMM, and Ocean.



(a) Water



(b) KMeans



(c) BlackScholes

Figure 6.8: Optimal LLC_{size} at different problem sizes (S2-S4), total cache sizes (32M-128M), and the number of cores for Water, KMeans, and BlackScholes.

significantly below a particular cache capacity in shared caches, and this cache capacity grows with core count. When core count scales, the routing distance also increases in shared LLCs. To offset these effects, the optimal $T \times L2_{size}$ (LLC_{size}) must grow (decrease) with core count scaling. Figure 6.6–Figure 6.8 confirm this effect across our 6 benchmarks, FFT, RADIX, Barnes, FMM, Ocean, and Water at the S4 problem size with 32MB. LU, KMeans, and BlackScholes have good data locality, so the optimal LLC_{size} is almost constant across core counts.

Problem size scaling affects the $\text{LLC}_{size,opt}$ selection in two ways. First, when the problem size is small compared to the total cache capacity, the optimal cache configuration usually happens at $T \times L2_{size} = LLC_{size} = \frac{1}{2}C$ across different core counts. For example, the optimal LLC_{size} for the S2 problem size is around half of the total cache size–*i.e.*, 16MB, 32MB, and 64MB.

Second, because problem size scaling increases the memory footprint, the $\text{LLC}_{size,opt}$ grows as problem scales to reduce expensive off-chip accesses. For the S3 and S4 problem sizes, the 32MB total cache capacity is usually too small, and the $\text{LLC}_{size,opt}$ varies with core count. The $\text{LLC}_{size,opt}$ of S3 and S4 are almost identical at small core counts, but the $\text{LLC}_{size,opt}$ of S3 is larger than that of S4 at larger core counts. For example, FFT shows this behavior at 32MB. This is because the high reference counts region is large enough to affect the LLC access frequency. So, we need large L2_{size} to keep memory access locally at the S4 problem size. In contrast, for a large cache capacity that can contain the major working set of the S3 problem size, the $\text{LLC}_{size,opt}$ of S4 is larger than the $\text{LLC}_{size,opt}$ of S3 across all core counts. For example, we see this behavior in FFT, RADIX, Barnes, FMM, and Ocean when

the total cache capacity is 128MB.

Private LLCs

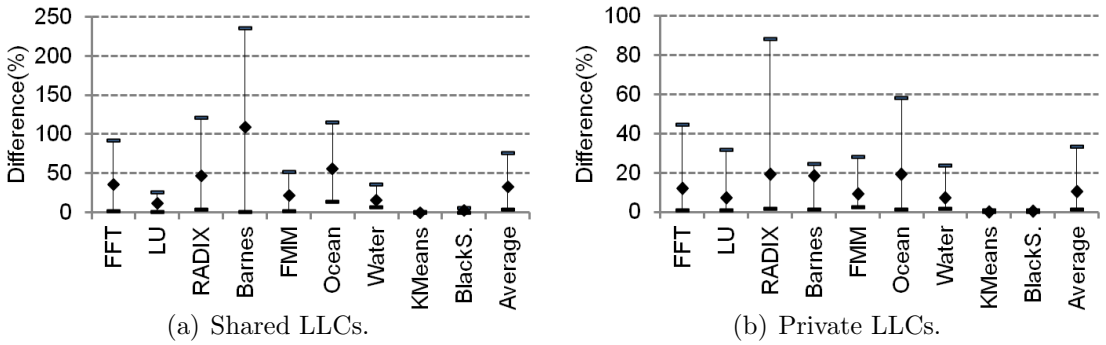
For private LLCs, the total L2 size should also contain the major working set which grows as core count scales. However, core count scaling also degrades data locality at large RD values due to increased replications and invalidations, which prefer a larger LLC_{size}. These combined effects complicate the $\text{LLC}_{size,opt}$ behavior.

We use the FFT's graph at 32MB as an example to explain this complicated behavior. For the S2 problem size, the major working set size is small and can be contained within a small L2_{size}. So the $\text{LLC}_{size,opt}$ usually increases as core count scales to reduce the directory-access and off-chip traffic. However, at large problem sizes (*i.e.*, S4), the major working set shifts to larger RD values. Reducing private LLC accesses has more benefit–*i.e.*, the decreasing rate of the total on-chip memory stall is faster than the increasing rate of the total off-chip memory stall. So the $\text{LLC}_{size,opt}$ usually decreases as core count scales. We see this in FFT, LU, RADIX, Barnes, FMM, Ocean, and Water.

For benchmarks with good data locality in private caches, KMeans and BlackScholes, the optimal LLC_{size} is almost constant across core counts, and the values are vary similar to the shared $\text{LLC}_{size,opt}$. LU is an interesting benchmark. It has good locality in shared caches, but it has very bad data locality in private caches due to massive replications. Hence, the optimal LLC_{size} usually increases as core count scales.

AMAT Variation for L2/LLC Partition

For a given core count, problem size, total cache size, and benchmark, we



(a) Shared LLCs. (b) Private LLCs.

Figure 6.9: AMAT difference between the highest and the lowest AMAT.

use our models to compute the highest AMAT and the lowest AMAT. We report the largest performance variation due to L2/LLC capacity partition, so we can understand the impact of capacity partition.

Figure 6.9 shows the percentage difference between the highest and the lowest AMAT_s (AMAT_p) across our 9 benchmarks. The percentage difference is defined as $\frac{highestAMAT - lowestAMAT}{lowestAMAT} \times 100\%$. Each benchmark reports the average difference across 2–256 cores, S2–S4 problem sizes, and 32MB–128MB total cache sizes. The last bar is the average across all benchmarks.

When applications have very good locality (*i.e.*, KMeans and BlackScholes), the percentage difference is close to 0. This is because 8KB L1_{size} can capture the main working set. For shared LLCs, the variation can reach 236% in Barnes, as illustrated in Figure 6.9(a). On average, the difference is between 3.4% and 76.0%, and the overall average difference across benchmarks is 33.4%.

Figure 6.9(b) illustrates the percentage difference for private LLCs. The largest difference is 85.6% in RADIX. On average, the difference is between 1.0% and 33.3%, and the overall average difference across benchmarks is 10.3%. The percentage

difference of private LLCs is smaller than the difference of shared LLCs. This is because the higher access latency in shared LLCs causes higher AMAT variations. So the $L2_{size}$ has greater impact for shared LLCs than for private LLCs.

Private vs. Shared LLC at the Optimal LLC Capacity

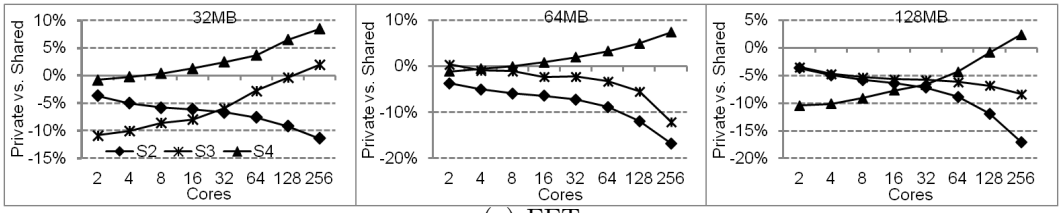
Lastly, Figure 6.10 and Figure 6.11 illustrates the percentage difference between $AMAT_s$ and $AMAT_p$ ($\frac{AMAT_s - AMAT_p}{AMAT_s} \times 100\%$) at $LLC_{size,opt}$. We plot three $T \times L2_{size} + LLC_{size}$, 32MB, 64MB, and 128MB, per benchmark. For each graph, the X-axis is the core count, and the Y-axis is the corresponding percentage difference. We also report the difference for three problem sizes, S2–S4. Private LLCs outperform shared LLCs when the Y-axis value > 0.

Although the preference of shared or private LLCs depends on core count and problem size, there are two important trends. First, because problem size scaling increases both C_{core} and C_{share}, continued problem size scaling usually prefers private LLCs (*i.e.*, curves move toward > 0) at the same on-chip cache capacity. Second, when the problem size is small compared to the on-chip cache capacity, the total on-chip memory stall (directory access + data forwarding) in private LLCs grows as core count scales and can be worse than the total on-chip memory stall in shared LLCs. Hence, core count scaling prefers shared LLCs in this case. For example, we see this in FFT, LU, RADIX, Barnes, FMM, Ocean, and Water at the S2 problem size with 64MB capacity. When problem size is large compared to cache capacity, core count scaling prefers private LLCs due to a high access penalty in shared LLCs. For example, we see this in FFT, RADIX, FMM, Ocean, Water, KMeans, and BlackScholes at 32MB and 64MB capacity for the S4 problem size. LU always

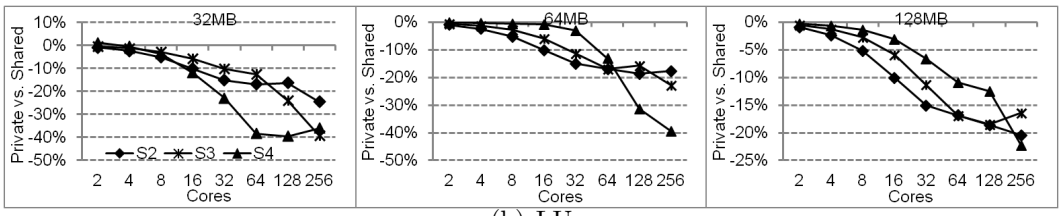
prefers shared LLCs due to the large number of replications in private LLCs.

Figure 6.12(a) summarizes the highest, the lowest, and the average percentage difference between private and shared LLCs in Figure 6.10 and Figure 6.11. Shared LLCs can outperform private LLCs by 39.5% in LU, and private LLCs can outperform shared LLCs by 29.0% in Ocean. On average, the difference is between -14.9% and 9.1%.

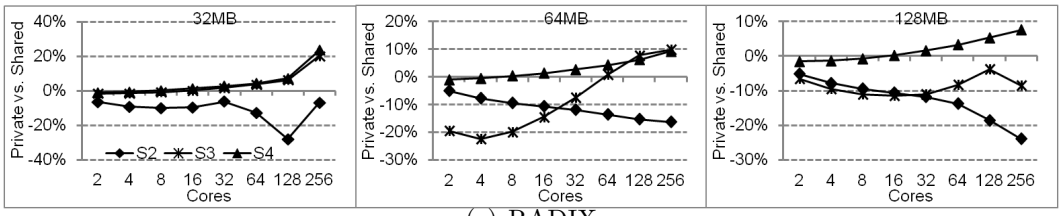
We can also model the IPC as $1/(1 + \frac{memory\ accesses}{instructions} \times AMAT)$ by assuming $CPI = 1$ in the absence of memory stall. Figure 6.12(b) reports the percentage difference ($\frac{IPC_s - IPC_p}{IPC_s} \times 100\%$). Private LLCs outperform shared LLCs when the Y-axis value < 0. The shared LLC's IPC can outperform the private LLC's IPC by 25.7% in LU, and the private LLC's IPC can outperform the shared LLC's IPC by 28.2% in Ocean. The average IPC difference is between -8.0% and 9.2%. From Figure 6.9 and Figure 6.12, we find that the capacity-partition has a larger impact than private-vs-shared-LLC selection on the cache performance. This suggests that physical data locality is vary important for future CMPs.



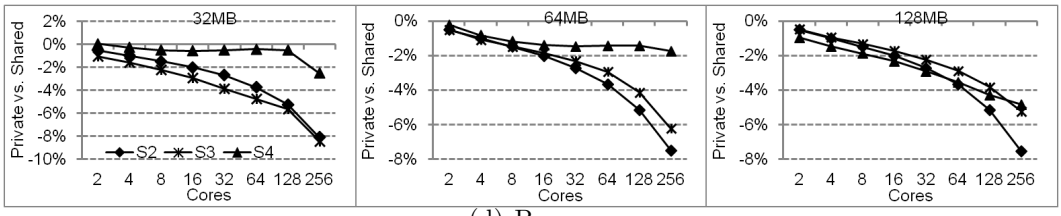
(a) FFT



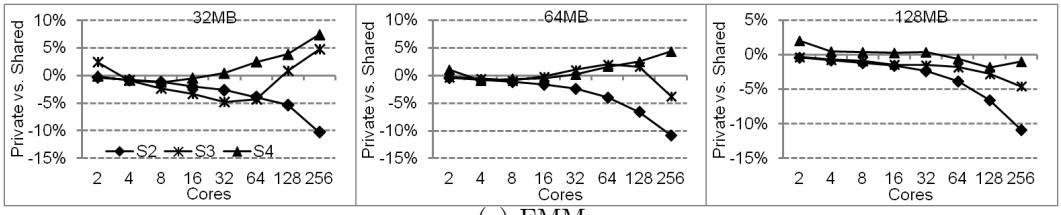
(b) LU



(c) RADIX



(d) Barnes



(e) FMM

Figure 6.10: AMAT difference between private and shared LLCs at $LLC_{size,opt}$ for FFT, LU, RADIX, Barnes, and FMM.

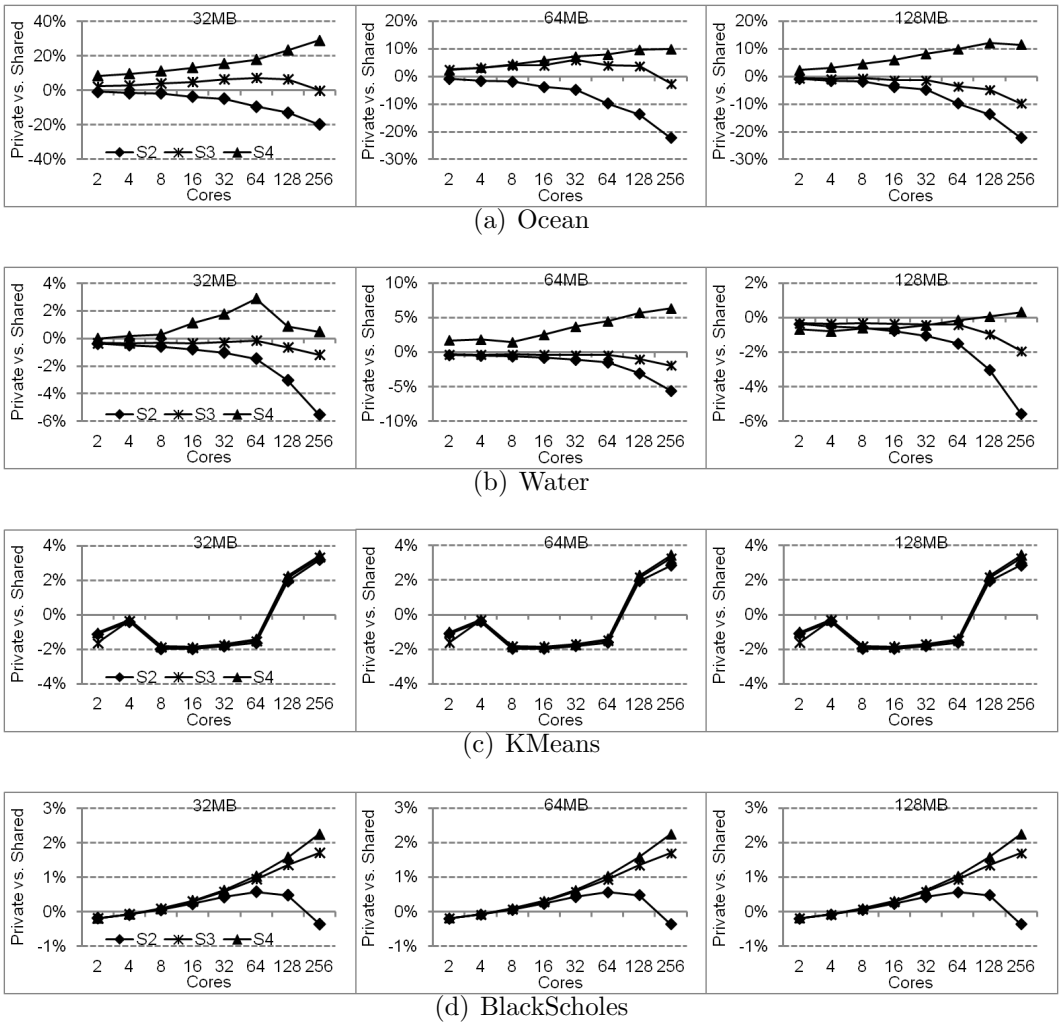


Figure 6.11: AMAT difference between private and shared LLCs at $LLC_{size,opt}$ for Ocean, Water, KMeans, and BlackScholes.

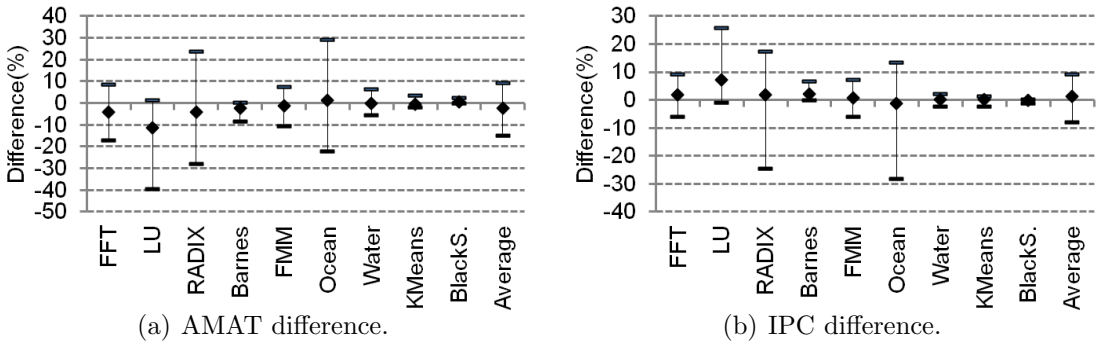


Figure 6.12: AMAT and IPC difference between private and shared LLCs at the optimal LLC_{size}.

Chapter 7

Prediction versus Sampling

In Section 2.2.1, we introduce our in-house built Pin tool. There are two major effects that slow down the reuse distance profiling. First, we use the fine-grain context switch at every memory reference, and the context switch is slow. Second, RD analysis maintains the depth information of every memory reference in LRU stacks. The splay tree[38] is a common algorithm to implement the LRU stack. However, the amortized complexity of the splay tree is $O(log(n))$. Reducing the number of references to track in LRU stacks can reduce the profiling time. Acquiring RD profiles with sampling is a technique for reducing the number of tracked memory references. In this chapter, we first introduce the RD sampling technique, then we compare the sampling technique with our prediction technique.

7.1 Multicore Reuse Distance Sampling

Instead of continuously tracking every memory reference, the sampling technique divides the entire profiling period into a sequence of interleaved fast-forward periods and profiling periods [39, 17]. In the fast-forward period, the profiler only does minimal maintenance to gather necessary information. In the profiling period, the profiler selects memory references from the dynamic memory reference stream and collects the reuse distances based on the selected references. The switching be-

tween the fast-forward period and the profiling period is controlled by the sampling rate. The sampling rate $R_{Sampling}$ is defined as $R_{Sampling} = \frac{T_p}{T_p+T_f}$, where T_p and T_f are the number of memory references in the profiling period and in the fast-forward period, respectively. A high sampling rate tracks more memory references and often gives higher accuracy. However, a high sampling rate also causes a longer profiling time. The sampling rate is adjustable to balance accuracy and performance.

Completely turning off the profiling in fast-forward periods can affect the accuracy for the references which have reuse windows $> T_p$. Because memory references that have long reuse distances can exist across a large number of fast-forward periods, one way to improve accuracy is to continue the profiling period until all of the references that have appeared in a profiling period have been reused. However, if a reference has been touched in the first profiling period and can only be reused at the end of the program, this outstanding reference will force the profiler to stay in the profiling period for the entire execution. Schuff *et al*[17] proposed a pruning technique to prevent this problem. The pruning procedure checks the oldest reference. If this reference's current reuse distance is sufficiently large, it is pruned and recorded as if it were a cold miss. We adopt their technique and make some modifications. We define the pruning rate, $R_{Pruning}$, as $R_{Pruning} = \frac{reused\ unique\ references}{total\ unique\ references}$ in the profiling period.

In our Pin tool, the profiler begins in a profiling period. In the profiling period, the memory references from P threads are interleaved uniformly, and each thread executes $\frac{T_p}{P}$ memory references. In our experiments, T_p is 100K memory references. Once T_p reaches 100K references and the desired $R_{Pruning}$, the profiler switches to

the fast-forward period. There are no fine-grain context switches and LRU stack updates in this period. After each thread executes $\frac{T_f}{P}$ memory references, the profiler switches back to the profiling period.

In our experiments, we first acquire the whole-program CRD_{direct} and PRD_{direct} profiles for 2–256 cores at the S1–S4 problem sizes directly for each benchmark. Then, we use the sampling technique to acquire the whole-program CRD_{direct} and PRD_{direct} profiles for each configuration. After we acquire the sampled profiles, we normalize the sampled profiles by the total number of references counts. Finally, to determine the accuracy of sampled profiles, we use two metrics, $\text{RD}_{Accuracy}$ and $\text{RD\_CMC}_{Accuracy}$, which are defined in Equation 4.4 and Equation 4.5.

7.2 Sampling Accuracy and Performance

In our experiments, T_p is 100K memory references, and T_f is 900K memory references ($\text{R}_{Sampling} = 0.1$). We set $\text{R}_{Pruning}$ to be 0.99.

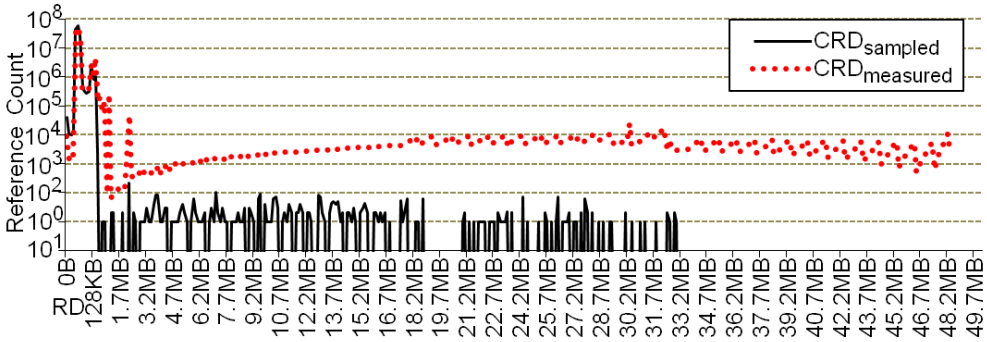
Figure 7.1(a) compares FFT's measured CRD_{direct} profile (dotted line) and sampled CRD_{direct} profile (solid line) running on 16 cores at the S3 problem size. In Figure 7.1(a), the measured CRD profile and the sampled CRD profile are very similar at small RD values. However, the sampled CRD profile has fewer reference counts at large RD values, and it ends earlier than the measured CRD profile. This is because in FFT, more than 99% of the unique memory references are reused in the same profiling period, and 0.99 $\text{R}_{Pruning}$ discards the references which have large RD values. The actual sampling rate is 9.5%. As a result, significant distortion happens

at large RD values, and the corresponding CMC profile also shows significant errors, as illustrated in Figure 7.1(b).

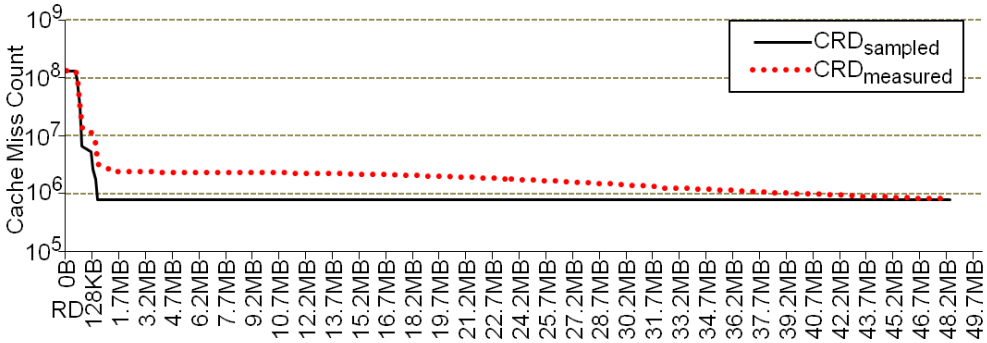
Figure 7.1(c) (Figure 7.1(d)) shows Barnes' measured and sampled CRD_{direct} (CRD_CMC_{direct}) profiles running on 16 cores at the S3 problem size. In contrast to FFT, Barnes' measured and sampled CRD_{direct} profiles are almost identical across all RD values. This is because in Barnes, less than 99% of unique memory references are reused in the same profiling period, and the profiler spends most of the time in the profiling period (almost never pruning). As a result, the actual sampling rate is 98.5%. Hence, the same sampling parameters, $\text{R}_{Sampling}$ and $\text{R}_{Pruning}$, cannot sample Barnes' profile efficiently.

The sampled PRD_{direct} profiles show the same behavior, as illustrated in Figure 7.2. FFT's sampled PRD profile also ends earlier than the measured PRD profile due to pruning. On the other hand, Barnes' sampled PRD profile is almost identical to the measured PRD profile because of the low pruning trigger-rate. These examples show the major challenge of sampling techniques. Sampling efficiency and accuracy highly depend on sampling parameters, and the same parameter cannot be used on different benchmarks.

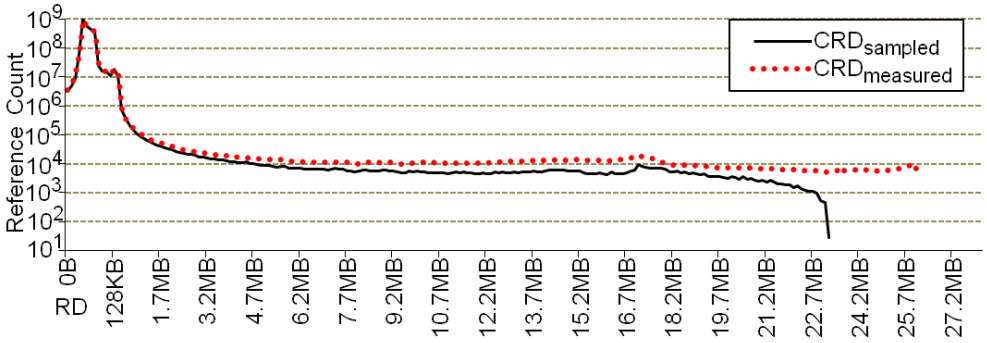
Figure 7.3(a) presents the accuracy results of sampled CRD profiles by using the $\text{CRD}_{Accuracy}$ metric. Each bar in Figure 7.3(a) reports the average CRD accuracy achieved over the 32 sampled CRD profiles per benchmark. The rightmost bar reports the average accuracy across all benchmarks. As Figure 7.3(a) shows, sampled CRD profiles have high accuracy. For all benchmarks, the CRD profile accuracy is between 83.2% and 98.7%. Across all benchmarks, the average accuracy is 94.2%.



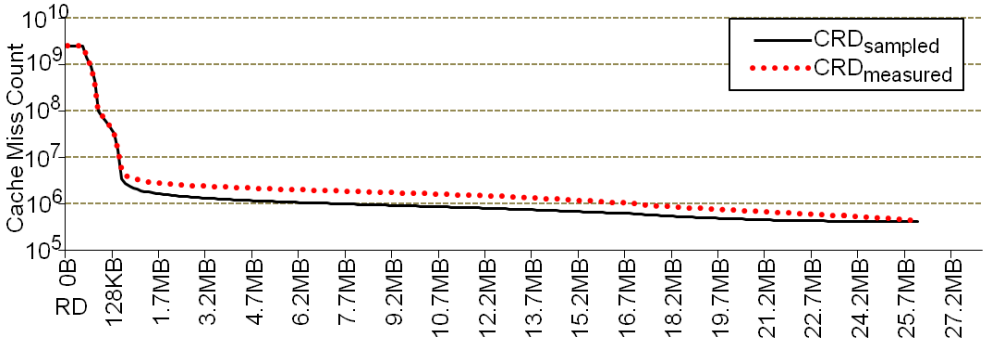
(a) FFT's measured and sampled CRD profiles running on 16 cores at S3.



(b) FFT's measured and sampled CRD_CMC profiles running on 16 cores at S3.

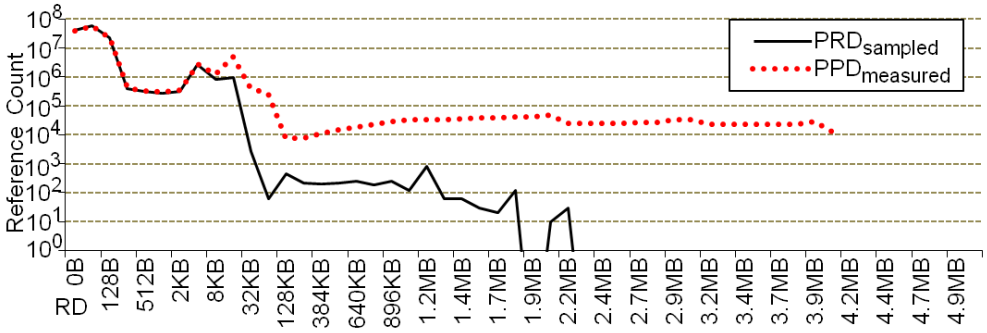


(c) Barnes' measured and sampled CRD profiles running on 16 cores at S3.

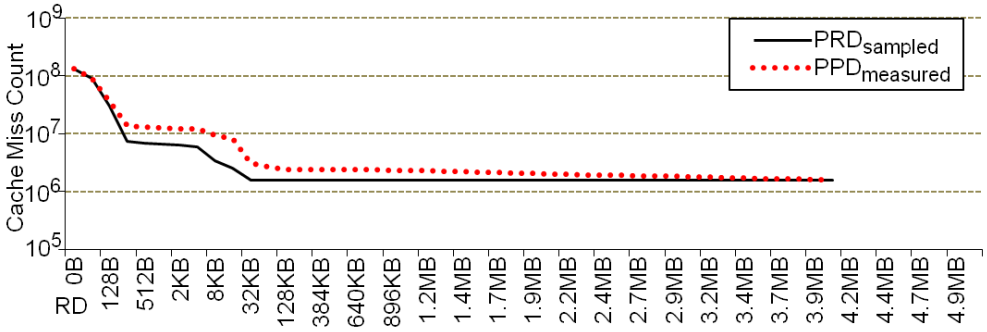


(d) Barnes' measured and sampled CRD_CMC profiles running on 16 cores at S3.

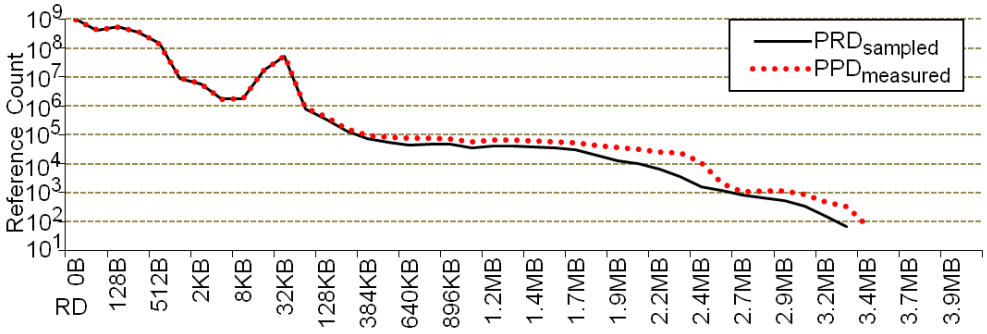
Figure 7.1: Measured and sampled CRD_{direct} profiles with $R_{sampling} = 0.1$ and $R_{pruning} = 0.99$.



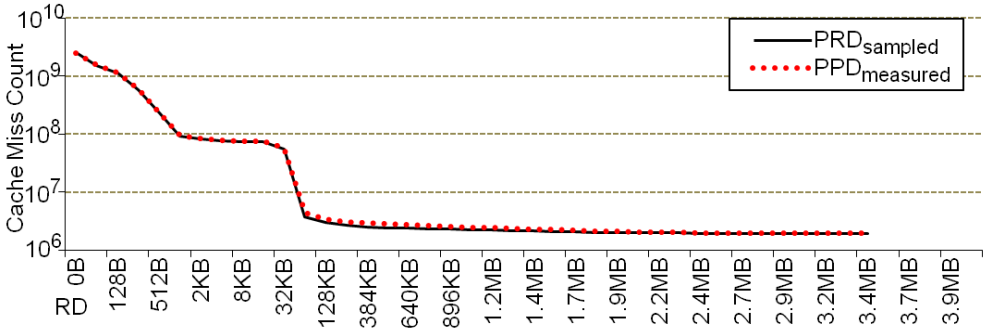
(a) FFT's measured and sampled PRD profiles on 16 cores at S3.



(b) FFT's measured and sampled PRD_CMC profiles on 16 cores at S3.

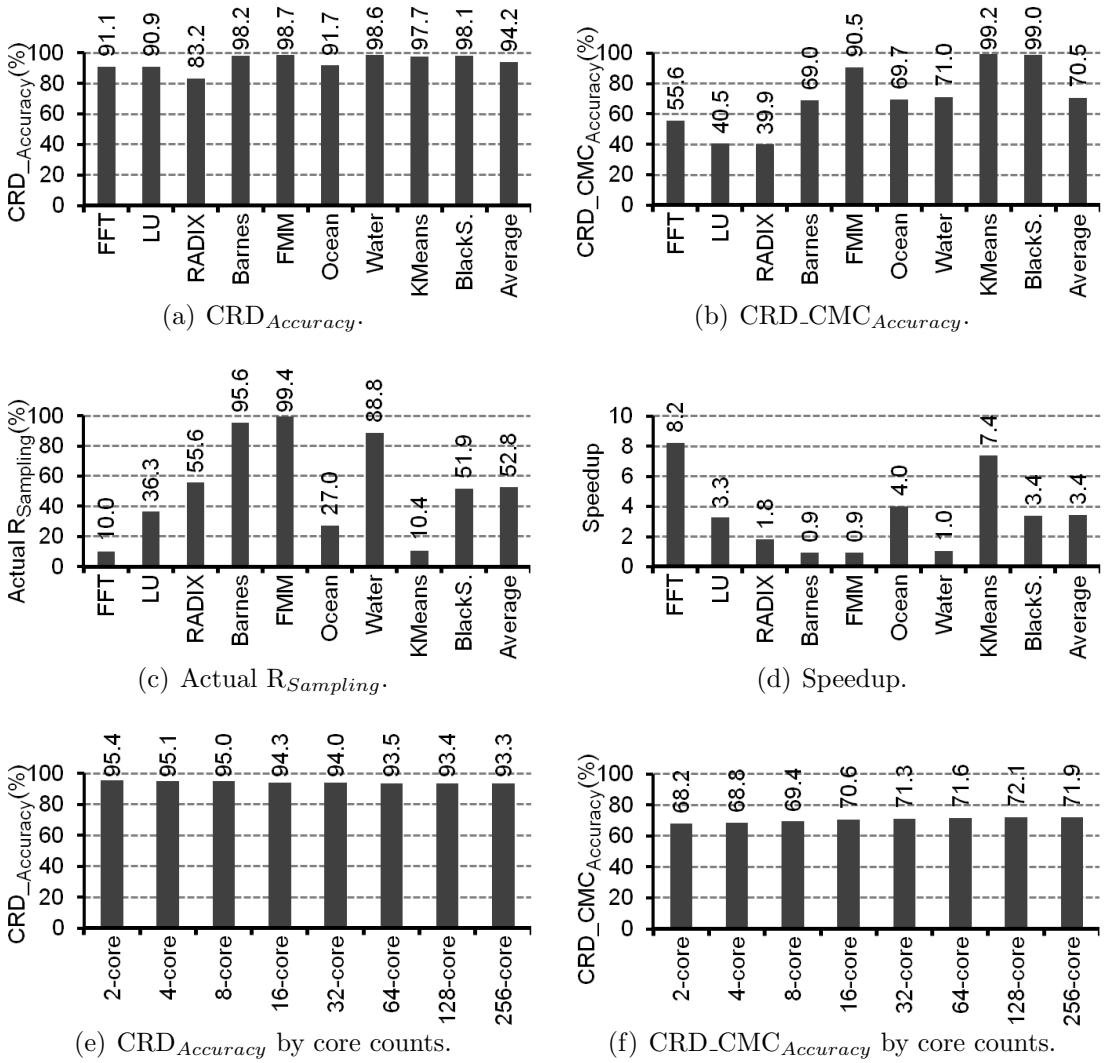


(c) Barnes' measured and sampled PRD profiles on 16 cores at S3.



(d) Barnes' measured and sampled PRD_CMC profiles on 16 cores at S3.

Figure 7.2: Measured and sampled PRD_{direct} profiles with $R_{sampling} = 0.1$ and $R_{pruning} = 0.99$.



(a) $\text{CRD}_{Accuracy}$. (b) $\text{CRD\_CMC}_{Accuracy}$.

(c) Actual $\text{R}_{Sampling}$. (d) Speedup.

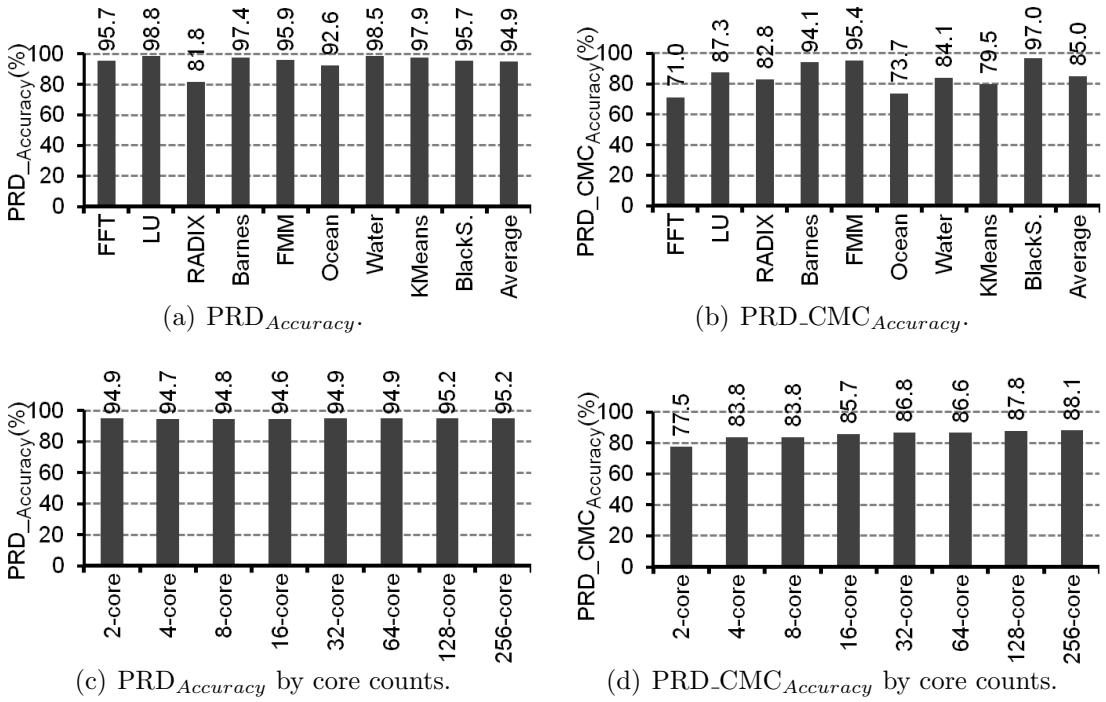
(e) $\text{CRD}_{Accuracy}$ by core counts. (f) $\text{CRD\_CMC}_{Accuracy}$ by core counts.

Figure 7.3: Accuracy and performance of the sampling technique with $\text{R}_{Sampling} = 0.1$ and $\text{R}_{Pruning} = 0.99$ for CRD profiles.

The high $\text{CRD}_{Accuracy}$ is because the sampling technique has better accuracy at small RD values and the metric, $\text{CRD}_{Accuracy}$, weights the small RD values that have high reference counts more heavily. At large RD values where the pruning happens, sampled profiles have larger errors. $\text{CRD\_CMC}_{Accuracy}$ treats each RD value equally, so it can reflect the error that happens at large RD values. Figure 7.3(b) reports $\text{CRD\_CMC}_{Accuracy}$, and the accuracy is much lower, between 39.9% and 99.2% for all benchmarks. On average, the accuracy is 70.5%. This result shows that pruning can cause significant errors.

Figure 7.3(c) reports the actual sampling rate. The actual $\text{R}_{Sampling}$ varies between 10.0% to 99.4%. Higher $\text{R}_{Sampling}$ means that the profiler stays in the profiling period longer. However, comparing Figure 7.3(c) and Figure 7.3(b) shows that a high sampling rate doesn't always guarantee high accuracy. For example, Barnes' profiler spends 95.6% of its time in the profiling period, but the $\text{CRD\_CMC}_{Accuracy}$ is 69.0%. In contrast, KMeans' profiler spends 10.4% of its time in the profiling period, but the $\text{CRD\_CMC}_{Accuracy}$ is 99.2%. Hence it is difficult to use the actual sampling rate to predict the accuracy. Figure 7.3(d) reports the actual speedup, which is related to the actual sampling rate. Lower sampling rate means higher speedup. On average, the speedup is 3.4x. Figure 7.3(e) and Figure 7.3(f) shows the $\text{CRD}_{Accuracy}$ and $\text{CRD\_CMC}_{Accuracy}$ breakdown by different core counts. In general, different core counts do not affect the sampling accuracy.

Figure 7.4(a) illustrates our accuracy results for sampled PRD profiles by using the $\text{PRD}_{Accuracy}$ metric. Sampled PRD profiles also have high accuracy. For all benchmarks, the PRD profile accuracy is between 81.8% and 98.8%. Across all



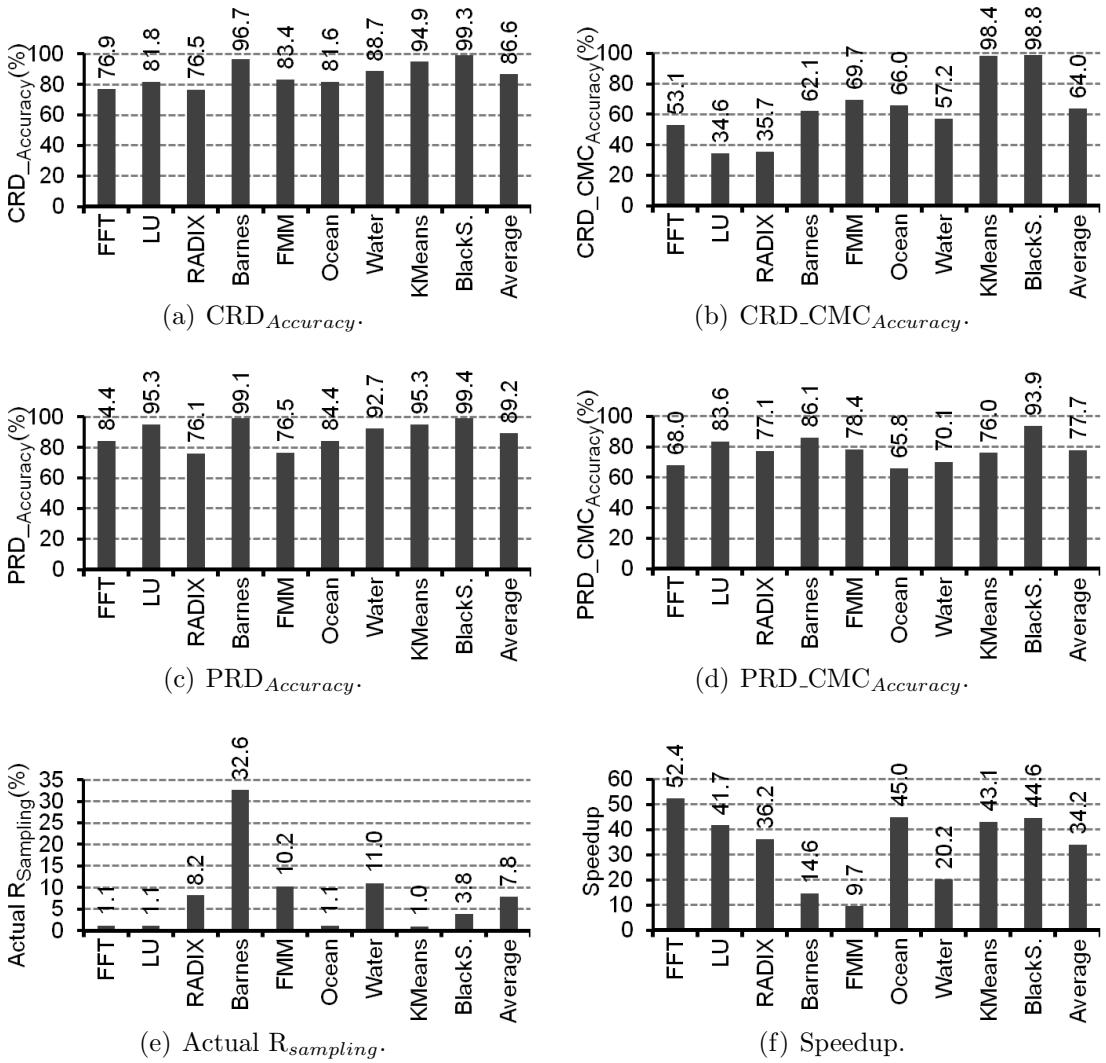
(a) $\text{PRD}_{Accuracy}$. (b) $\text{PRD\_CMC}_{Accuracy}$.

(c) $\text{PRD}_{Accuracy}$ by core counts. (d) $\text{PRD\_CMC}_{Accuracy}$ by core counts.

Figure 7.4: Accuracy of the sampling technique with $\text{R}_{Sampling} = 0.1$ and $\text{R}_{Pruning} = 0.99$ for PRD profiles.

benchmarks, the average accuracy is 94.9%. Figure 7.4(b) illustrates the CMC profiles accuracy using the $\text{PRD\_CMC}_{Accuracy}$ metric. The accuracy is lower, between 71.0% and 97.0% for all benchmarks. On average, the accuracy is 85.0%. The results show that pruning can cause significant error. Figure 7.4(c) and Figure 7.4(d) report the $\text{PRD}_{Accuracy}$ and $\text{PRD\_CMC}_{Accuracy}$ breakdown by different core counts. In general, core counts do not affect accuracy.

We can increase the sampling speed by relaxing $\text{R}_{Sampling}$ and $\text{R}_{Pruning}$. Figure 7.5 illustrates the sampling accuracy and speedup with $\text{R}_{Sampling} = 0.01$ and $\text{R}_{Pruning} = 0.90$. The average $\text{CRD}_{Accuracy}$, $\text{CRD\_CMC}_{Accuracy}$, $\text{PRD}_{Accuracy}$, and $\text{PRD\_CMC}_{Accuracy}$ drops to 86.6%, 64.0%, 89.2%, and 77.7%, respectively, compared to 94.2%, 70.5%, 94.9%, and 85.0% when using $\text{R}_{Sampling} = 0.1$ and $\text{R}_{Pruning} = 0.99$.



(a) $\text{CRD}_{Accuracy}$. (b) $\text{CRD\_CMC}_{Accuracy}$.

(c) $\text{PRD}_{Accuracy}$. (d) $\text{PRD\_CMC}_{Accuracy}$.

(e) Actual $\text{R}_{sampling}$. (f) Speedup.

Figure 7.5: Accuracy and performance of the sampling technique with $\text{R}_{Sampling} = 0.01$ and $\text{R}_{Pruning} = 0.90$ for CRD and PRD profiles.

However, the average speedup increases about 10x, achieving 34.2x. The results show that the sampling technique usually need to sacrifice accuracy for better speedup.

7.3 Compare with Prediction

Table 7.1 compares the accuracy and speedup of the sampling technique and the prediction technique. The three columns represent three different predictions, core count scaling, problem size scaling, and core-problem scaling as illustrated in Chapter 4. For each design space, we report $\text{CRD}_{Accuracy}$, $\text{CRD\_CMC}_{Accuracy}$, $\text{PRD}_{Accuracy}$, $\text{PRD\_CMC}_{Accuracy}$, and the speedup of the sampling technique and the prediction technique.

First, we compare the sampling technique ($\text{R}_{Sampling}$=0.1 and $\text{R}_{Pruning}$=0.99) with the prediction technique. For each design space, the sampling technique and the prediction technique have very similar $\text{CRD}_{Accuracy}$ and $\text{PRD}_{Accuracy}$. This is because both techniques can provide very high accuracy at small RD values. However, the sampling technique has lower $\text{CRD\_CMC}_{Accuracy}$ than the prediction technique. This is because the CRD profile usually has a long tail, and pruning causes higher error at large RD values. Hence, CMC accuracy is lower across a wide range of cache capacities. In contrast, the sampling technique has better $\text{PRD\_CMC}_{Accuracy}$ than the prediction technique. There are two reasons for this. First, the prediction technique has to predict the compulsory misses and coherence misses, which have infinite reuse distance. But the sampling technique can measure these reference counts

Table 7.1: Accuracy and speedup comparison between the sampling technique and the prediction technique.

Scaling method	Core count scaling	Problem size scaling	Core-Problem scaling
$CRD_{Accuracy}$ / $CRD\_CMC_{Accuracy}$			
Sampling($R_{Sampling}$=0.1,$R_{Rruning}$=0.99)	93.9%/71.1%	94.6%/65.4%	94.1%/70.1%
Sampling($R_{Sampling}$=0.01,$R_{Pruning}$=0.9)	85.9%/64.3%	90.8%/61.6%	86.9%/63.9%
Prediction	89.4%/83.6%	89.3%/86.1%	89.1%/83.5%
$PRD_{Accuracy}$ / $PRD\_CMC_{Accuracy}$			
Sampling($R_{Sampling}$=0.1,$R_{Pruning}$=0.99)	94.9%/86.5%	95.4%/84.0%	95.0%/85.4%
Sampling($R_{Sampling}$=0.01,$R_{Pruning}$=0.9)	88.9%/78.6%	92.4%/80.5%	89.6%/78.4%
Prediction	96.0%/83.6%	91.8%/80.7%	94.2%/80.8%
Speedup compared to full measurement			
Sampling($R_{Sampling}$=0.1,$R_{Pruning}$=0.99)	3.3	3.2	3.4
Sampling($R_{Sampling}$=0.01,$R_{Pruning}$=0.9)	34.4	33.2	34.7
Prediction	4.6	25.2	140.0

more precisely. Second, PRD profiles are shorter than CRD profiles. Although pruning also causes distortion at large RD values in RD profiles, the affected region is shorter. As a result, the sampling technique has higher $PRD\_CMC_{Accuracy}$.

For core count scaling, the prediction technique has 4.6x speedup, and the sampling technique only has 3.3x speedup. This is because the prediction technique uses 8 profiles to predict the other 24 profiles for each benchmark. However, the sampling technique has to measure all 24 profiles, and pruning doesn't always provide a stable speedup.

For problem size scaling, the prediction technique uses profiles at the S1 and S2 problem sizes to predict the profiles at the S3 and S4 problem sizes. The prediction technique avoids the long profiling time at the S3 and S4 problem sizes. However, the sampling technique still needs to profile every configuration. So the prediction technique shows higher speedup (25.2x) than the sampling technique (3.2x). Lastly,

for core-problem scaling, the prediction technique achieves 140.0x speedup, because it only requires 4 profiles for each benchmark to predict the other 28 profiles. The sampling technique has to profile these 28 profiles and only provides 3.4x speedup.

We can relax $R_{Sampling}$ and $R_{Pruning}$ ($R_{Sampling}$=0.01 and $R_{Pruning}$=0.90) to increase the sampling performance. As Table 7.1 shows, accuracy decreases, but performance increases about 10x. This shows that the sampling technique is 7.5x and 1.3x faster than the prediction technique for core count scaling and problem size scaling, but it is still 4x slower for core-problem scaling.

Although the sampling technique can improve the profiling performance, one drawback is that the same parameters cannot be applied to all benchmarks. Furthermore, we don't know how good the parameters will be when we profile a new application. Usually, we need to try several different parameters to achieve good accuracy and performance at the same time. These drawbacks reduce the benefit of using the sampling technique. Some adaptive sampling techniques may be developed to help solve these issues. However, this topic is beyond the scope of this work. In contrast, the prediction technique is more stable than the sampling technique. The main drawback of the prediction technique is that accuracy decreases as the prediction horizon increases. We may need to measure more profiles to increase the prediction accuracy. However, the strength of the prediction technique is that it can predict any configuration without measuring them. As a result, we believe our prediction technique is more efficient than the sampling technique.

Chapter 8

Related Work

This chapter surveys background material and related work. First, it introduces recent developments in multicore reuse distance analysis. It then focuses on the design space exploration.

8.1 Reuse Distance Analysis

Multicore RD analysis is relatively new, but it is becoming a viable tool as the result of recent research.

Multicore Reuse Distance

Jiang *et al* [14] propose a probabilistic model for deriving CRD profiles from per-thread traces. They find that for the special case of parallel programs, the relative execution speed of threads does not change across different multicore architectures. Hence, CRD profiles remain the same. We find that the CRD and PRD profiles of loop-based parallel programs exhibit low sensitivity to cache capacity scaling is a very similar observation. However, Jiang's model requires knowing all per-thread traces, and it cannot explore multicore configurations that have not yet been profiled.

Suh *et al* [40] have developed the locality model to capture the effect that the reuse distance of a memory reference is inflated by memory accesses from another

program. Chandra *et al* [41] have also developed statistical models to predict the impact of cache sharing on co-scheduled threads. They focus on a two-core system, and don't consider scaling up to more cores. They calculate the related execution speed between threads and compute how many distinctive memory blocks from the second thread should be inserted into the reuse distance profile of the co-running thread. They focus on multi-programmed workloads, whereas our work focuses on parallel programs.

Ding and Chilimbi [13] extend Chandra's techniques and present techniques to construct CRD profiles for multi-threaded programs. They analyze memory traces to extract statistics on per-thread locality and data sharing to reconstruct CRD profiles. Their approach is general, because it can handle non-symmetric threads. However, it requires at-scale profiling to obtain the memory traces for analysis. Furthermore, their algorithm needs to consider the possible ways that memory references can be interleaved, incurring exponential time complexity. Ding and Chilimbi can also predict CRD profiles for machine scaling, but the additional threads must be identical to the already profiled threads. As a result, for parallel programs where per-thread computation changes with core count scaling and problem size scaling, their model does not work.

Ding and Chilimbi [42] also develop a method for measuring the footprint of concurrent execution applications. Xiang *et al* [43, 18] follow Ding and Chilimbi's work [42], and they propose a more efficiently composable model that uses the all-window footprint of each program to predict its cache interference with other programs. However, their model only considers multi-program workloads, and they

have to gather all the traces. We do not require traces, and our analysis is simple, allowing our approach to handle loop-based parallel programs running on LCMP-sized machines.

Shi *et al* [15] propose a stack simulation method to study the performance of multiple cache organizations in a single-pass. Their method uses a shared stack and per-core private stacks to collect the reuse distances. Instead of acquiring full reuse distance profiles, they organize shared and private stacks as groups [44], and these groups can represent various cache sizes. Another work by Schuff [16] investigates the accuracy of RD analysis for multicore processors, and they predict the miss-rate of shared and private caches. They also propose using multicore RD analysis and communication analysis to model memory systems [45]. However, they don't provide detailed methods and analyses.

Both Shi and Schuff predict cache performance at different cache sizes, but they cannot predict configurations for varied core counts and problem sizes. In contrast, we focus on studying and predicting the cache performance impact under different scaling schemes–*i.e.*, core count scaling, problem size scaling, and core-problem scaling.

Time Distance

Reuse distance analysis tracks every memory reference and maintains the depth information of every memory reference in LRU stacks. The splay tree[38] is a common algorithm for implementing the LRU stack, but the amortized complexity of the splay tree is $O(log(n))$. Another approach is to use time distance to estimate reuse distance. Time distance is defined as the number of intermediate

memory references between data reuse, and the cost of counting is constant. Berg and Hagersten [46, 47] propose a probabilistic model, StateCache, for computing cache miss rate from time distance. Based on their work, Eklov and Hagersten [48] propose a new model, StatStack, to estimate an application's reuse distance and cache miss rate. Shen *et al* [49, 50] also propose a statistical model to approximate a reuse distance histogram from time distance. However, these works focus on the sequential programs.

Sampling

Recently, researchers also propose using sampling techniques to improve the performance of reuse distance profiling. Berg and Hagersten [46, 51, 52] propose using sampling to gather the time distance information. Their method uses SPARC hardware performance counters and watch-points to track the selected addresses. Time distance is easier to collect than reuse distance, and can use the performance counter to count the number of memory references passed.

Zhong and Chang [39] use sampling to reduce the time overhand in the reuse distance collection. Their system utilizes the structure of bursty tracing [53], which divides the memory reference stream into a sequence of interleaved sampling intervals and hibernating intervals. The reuse distance is collected in the sampling interval, and the hibernating interval only has minimum maintenance. They also apply Ding and Zhong's tree-based approximate reuse distance analysis to save profiling time and space[26]. The average speedup is 7.5x compared with non-sampled profiling.

Beyls and D'Hollander [54] apply the sampling technique on their measurement

and visualization tool to identify the causes of cache misses. The reuse distances are measured for 20 million references, then the next 180 million accesses are skipped. The slowdown is between 15x and 25x. Their later work uses reservoir sampling to reduce the execution time and memory requirements of the reuse path analysis [55]. Their main goal is to find the most appropriate refactorings of single-threaded programs and to optimize the data locality [54, 56, 55, 57, 58].

Schuff *et al* [17] use the Intel Pin tool along with sampling and parallelization to accelerate CRD and PRD profiles acquisition for a 4-threads application running on a 4-cores machine. They use hash tables to track the unique memory references between the reuse of selected samples. However, their parallelization cannot exceed the machine's core counts. To profile the application that has more threads, the memory streams from threads must be serialized, hence reducing the benefit of parallel profiling.

Our work is orthogonal to sampling techniques. We reduce the number of needed profiles, whereas the sampling technique reduces the profiling time per profile. Our experiments show that the sampling time indeed increases as core counts and problem sizes increase. Hence, even with profiling acceleration, it is still very difficult to exhaustively explore LCMP design spaces that can easily reach more than 1000 configurations.

Reuse Distance Prediction

RD analysis has also been used to analyze uniprocessor caches across all data input sets [26, 20, 59, 60]. As discussed earlier, our work uses reference groups from Zhong *et al* [20] to predict the profile shift under core count scaling.

8.2 Design Space Exploration

Several researchers have conducted CMP design space explorations to understand the performance, energy and temperature of different CMP architectures.

Detailed Simulation

Hu *et al* [1] consider the area and performance trade-offs for LCMPs in order to determine the number of cores and core type for future server CMPs. They conclude that out-of-order cores will maximize jobs throughput on future CMPs, because out-of-order cores are more area efficient than in-order cores. Ekman and Stenstrom [2] study the trade-off between the issue-width of the cores and the number of cores on a chip. They find four-issue cores achieve a good balance between ILP and TLP. Hsu *et al* [3] explore the cache hierarchy requirements of LCMP platforms. They find on-chip and off-chip bandwidth demands play a significant role in optimizing LCMP cache hierarchy.

Li and Martinez [4] show that parallel computation on a CMP can improve energy efficiency, compared to the same performance achieved by a uniprocessor setup. They also find that a limited power budget can cause significant performance degradation beyond a certain core count. Li *et al* [5] explore the multi-dimensional design space across a range of possible chip sizes and thermal constraints. They conclude that thermal constraints dominate other physical constraints such as off-chip bandwidth and power. It is important to consider thermal constraints while optimizing other parameters. Monchiero *et al* [7] explore the design space related to core count, cache size, and core complexity up to 8 cores, and they show how

different configurations impact performance, energy, and thermal distribution. They conclude that LLC is an important factor in determining performance and thermal behavior. To achieve the best energy-delay, LCMPs should consist of a large number of fairly narrow cores.

Zhao *et al* [8] study cache design space for 32 cores LCMP by considering area constraints and on-chip/off-chip bandwidth limitations. They introduce a constraints-aware analysis methodology to narrow down the design space and explore LCMP cache design options. They also recommend an LCMP architecture which has a three-level cache hierarchy with 512KB to 1MB of L2 cache per node, and each node has 4 cores. The LLC size should be a minimum of 16MB. The platform should also support at least 64GB/s of memory bandwidth and 512GB/s of interconnect bandwidth. Davis *et al* [6] consider CMTs with up to 34 cores and 8MB LLCs, but studied large-scale parallelism–up to 240 threads–when factoring in per-core multithreading. They explore the design space for core type, core count, cache size, and the degree of multithreading. They find that the best CMT performance happens when using simple cores with 4-8 threads per core.

Wu and Yeung [61] study tiled CMPs scaling from 1-256 cores and 4-128MB of total L2 cache. They evaluate the impact of scaling on off-chip bandwidth and on-chip communication. Their results show that there should be ample on-chip bandwidth. However, for memory intensive programs, off-chip memory overheads dominate.

Analytic Model

Rogers *et al* [9] develop an analytical model in order to study the bandwidth

wall problem for CMP systems. They find that the bandwidth wall problem can severely limit core scaling. Esmaeilzadeh *et al* [10] model multicore scaling and show that regardless of chip organization and topology, multicore scaling is power limited. 21% of a fixed-size chip must be powered off at 22nm, and this number grows to more than 50% at 8nm.

Hill and Marty [62] apply Amdahl's law to build a cost model for a CMP. They assume that a CMP can support at most n base core equivalents (BCEs) for a given size and technology generation. They use their cost model to study the speedups for symmetric, asymmetric, and dynamic CMPs. However, they do not consider the impact of cache organizations. Sun *et al* [11] develop a model to optimize a cache hierarchy under a power constraint. They apply the so-called-2-to-$\sqrt{2}$ rule [63]–*i.e.*, if the cache size is doubled, the miss rate drops by a factor of $\sqrt{2}$. Ho *et al* [64] study the trade-offs between core counts and cache capacities. They also employ the square-root-rule cache-miss model to estimate the trade-offs between shared, private, and hybrid cache organizations. They find that different cache organizations have different optimal core counts and cache capacities. At its peak performance, shared caches can contain more cores than private caches.

Wentzlaff *et al* [65] develop a system-level IPC model to evaluate a large range of cache and core count configurations. To model cloud computing applications, they assume a workload of running independent SPEC Int 2000 programs on each core. Hence, they don't consider about data sharing. They find the increased area provided by technology advances is better used for cache due to the off-chip bandwidth constraints. They also suggest using embedded DRAM as L2 caches, because

the area density of embedded DRAM can outcome the latency overhead. Krishna *et al* [66] extend Wentzlaff's model [65] to take multi-threaded data sharing in to account. They find that data sharing can significantly improve system throughput when compared to a parallel application which has no data sharing. However, the benefit from data sharing is diminished in an off-chip bandwidth constrained system.

These analytic models usually simplify an application's memory behavior when driving the models because it is difficult to gather the detailed cache performance at many cache capacities. However, multicore RD profiles can provide detailed memory behavior (locality and data sharing) at any cache capacity, and allow to build more realistic models.

Machine Learning

These previous studies show that multicore design spaces are very large and complex. Using detailed simulation to study many combinations of different architecture parameters is very time consuming. Some researchers propose using machine learning techniques to speed up design space exploration [67, 68, 69]. Ipek *et al* [67] use artificial neural networks to train the approximation models. Lee and Brooks [68] develop regression models to build approximation functions. Cook and Skadron [69] propose using genetically programmed response surfaces (GPRS) to address this challenge.

Our work is closely related to these previous studies. However, we apply multicore RD analysis to study how different scaling schemes impact multicore cache performance. Our approach learns more per sample (CRD and PRD profiles), reducing the number of needed simulations. Our prediction technique can also explore

the complete cross product of design space (core count, cache size, and problem size) very efficiently. Finally, our work investigates performance scaling only; we do not consider how power scales, and other physical constraints. Understanding the limitations on scaling LCMPs due to physical constraints is critical, and an important direction for future work.

Chapter 9

Conclusion and Future Work

In this chapter, we first summarize our work, then we propose possible directions for future research.

9.1 Summary

Recently, researchers have extended reuse distance analysis to parallel programs running on multicore processors. A major problem is memory reference interleaving. Hence, CRD and PRD profiles are *architecture dependent*. However, such architecture dependency is minimal when threads have similar access patterns. For example, loop-based parallel programs contain symmetric threads. For these programs, CRD and PRD profiles are *minimally architecture dependent* and can provide accurate analysis. Our study confirms this characteristic, and it enables us to develop an efficient multicore RD analysis.

In this work, we investigate different inter-thread interactions in CRD and PRD profiles. We find that dilation and overlap are the major effects in CRD profiles. Scaling and demotion absorption are the major effects in sPRD profiles. In addition to the insights of inter-thread interactions, we notice that the gap between CRD_CMC and sPRD_CMC profiles represents the cache performance difference between shared and private caches. This dissertation defines an important split

point between CRD_CMC and sPRD_CMC profiles, C_{share}. Beyond this point, shared caches show the locality advantage over private caches. Most importantly, the degree of data sharing is not a fixed characteristic of a given application; it is a function of RD value. So the choice between private and shared caches also depends on the cache capacity.

Because machines and problem sizes will continue to scale in the future, it is important to understand the scaling characteristics of parallel programs. When core count increases, CRD profiles shift *coherently* to larger RD values in a shape-preserving way. Shifting slows down and eventually stops at a certain RD value. We call this point C_{core}. Core count scaling only impacts cache performance significantly below this stopping point in shared caches. When core count increases, sPRD profiles also shift to larger RD values in a shape-preserving way. However, core count scaling increases the amount of replications and coherence misses in private caches. In contrast to CRD profiles, there is no C_{core} in sPRD profiles, and data locality degradation happens across all RD values. In this work, we find that C_{core} shifts to larger RD values and C_{share} shifts to smaller RD values with core count scaling. When considering problem size scaling, both C_{core} and C_{share} shift to larger RD values. Hence, problem size scaling may reduce the benefit of using shared caches at a fixed cache capacity.

Because the CRD and PRD profiles of loop-based parallel programs show the coherent shifting with core count scaling and problem size scaling, we develop techniques to predict the coherent movement of CRD and PRD profiles under different scaling schemes. The average profile prediction accuracy is between 80.7% and

96.3%. Then, we use M5 simulator to model tiled CMPs, and we simulate a total of 3,168 configurations to validate profile stability and MPKI prediction. We confirm that CRD and PRD profiles are minimally architecture dependent for cache capacity scaling. Hence, it is valid to assume uniform memory interleaving for loop-based parallel programs. Our core count prediction techniques can predict shared LLC (private L2 cache) MPKI to within 10% (13%) of simulation across 1,728 (1,440) configurations using 72 measured CRD (PRD) profiles. When combined with the existing prediction technique for problem size scaling, we can predict shared LLC (private L2 cache) MPKI to within 12% (14%) of simulation using 36 measured CRD (PRD) profiles. The results show that our prediction technique can help explore a large design space efficiently. Overall, we find our prediction techniques for core count scaling can accelerate cache analysis without sacrificing accuracy. When combined with problem scaling prediction, analysis effort is further reduced, though error increases when predicting large core counts.

We also develop a novel framework to identify optimal multicore cache hierarchies for loop-based parallel programs by using multicore reuse distance analysis. Although CRD profiles show better data locality over sPRD profiles beyond C_{share}, this benefit must be weighted against the higher access latency of shared caches. The optimal cache hierarchy exists when *the total on-chip and off-chip memory stalls are balanced.* We find that the capacity of the last private cache above the last level cache (LLC) must exceed the region in the PRD profile where significant data locality degradation happens. Shared LLCs can outperform private LLCs when *the total off-chip memory stall saved in shared LLCs is larger than the total on-chip mem-*

ory stall saved in private LLCs. At the optimal LLC size, the average performance (AMAT) difference between private LLCs and shared LLCs can reach as high as 15%, but it is smaller than the performance difference caused by L2/LLC partition (76% in shared LLC, and 33% in private LLC). This suggests that physical data locality is very important for multicore cache designs.

Lastly, we compare our prediction technique against the RD sampling technique, which can also accelerate the acquisition of profiles. The prediction technique and the sampling technique have similar average accuracy. However, the sampling technique needs to collect profiles at every configuration in the design space. In contrast, the prediction technique can predict any configuration from a small number of measurements. The benefit of prediction becomes more significant for core-problem scaling. As a result, our prediction technique can outperform the RD sampling technique.

9.2 Future Directions

In this research, we show that multicore RD analysis can provide extremely rich information about the memory behavior of parallel programs. We also implement prediction techniques to predict profiles under different scaling schemes. We believe this work builds a solid foundation for several possible future avenues.

Dynamic Multicore Resource Management

Time distance can also be used to estimate an application's memory behavior [46, 47, 48, 49, 50]. The nature of time distance means that it can be measured

by using hardware performance counters easily. Fedorova [70] uses Berg and Hagersten's model [47] to estimate the cache-miss ratios of co-running programs and drive the OS scheduler to improve processor throughput. Jiang *et al* [71] develop a locality model based on concurrent reuse distance for shared-cache contention prediction. Their contention-aware scheduling system for co-running programs can achieve performance and fairness.

For parallel programs running on LCMPs, we can use performance counters and Berg's technique to estimate reuse distance profiles. After profiling a few samples, the OS scheduler can use our prediction technique to predict memory performance across different machine configurations. Depending on the predicted memory stress, the OS scheduler can decide how many cores and cache slices should be active, and it can then shut down the other cores or cache slices to improve power efficiency.

Multicore Performance Analysis and Optimization Tool

High performance processors have performance monitoring counters (PMCs) to gather the runtime information of applications–*e.g.*, instruction counts and cache miss counts. Several performance tools have been developed to help programmers profile their applications on the real hardware. For example, Intel's VTune [72] is a widely used tool. However, the profiling information only reflects the machine performance where the tool is running on. There is no information as to how the application will perform on different machines.

Multicore RD analysis provides a good visualization tool. Our multicore RD analysis framework can easily help programmers understand the application's mem-

ory behavior on different processors. The locality information can also provide in-depth information about how to optimize the data locality. Several works have been done to provide visualization tools and to optimize the data locality for sequential programs [54, 56, 55, 57]. The same techniques can be extended to multicore RD analysis.

Irregular Memory Reference Interleaving

In this work, we focus on loop-based parallel programs on homogeneous CMPs. The symmetric threads make the interleaving of memory reference streams systematically, and profiles are predictable for different scaling schemes. However, for asymmetric-thread programs or heterogeneous CMPs, it is difficult to acquire CRD and PRD profiles due to the irregular memory reference interleaving. Although there exist several composable models [13, 14] to handle this issue, the analysis usually requires exponential time and cannot scale up to large core counts. Hence, multicore RD analysis needs to be further investigated to handle these challenges.

Bibliography

[1] J. Huh, S. W. Keckler, and D. Burger, "Exploring the design space of future CMPs," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[2] M. Ekman and P. Stenstrom, "Performance and power impact of issue-width in chip-multiprocessor cores," in *Proceedings of the International Conference on Parallel Processing*, 2003.

[3] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell, "Exploring the cache design space for large scale CMPs," *ACM SIGARCH Computer Architecture News*, 2005.

[4] J. Li and J. F. Martinez, "Power-performance implications of thread-level parallelism on chip multiprocessors," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2005.

[5] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron, "CMP design space exploration subject to physical constraints," in *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 2006.

[6] J. Davis, J. Laudon, and K. Olukotun, "Maximizing CMP throughput with mediocre cores," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[7] M. Monchiero, R. Canal, and A. González, "Design space exploration for multicore architectures: a power/performance/thermal view," in *Proceedings of the 20th Annual International Conference on Supercomputing*, 2006.

[8] L. Zhao, R. Iyer, S. Makineni, J. Moses, R. Illikkal, and D. Newell, "Performance, area and bandwidth implications on large-scale CMP cache design," in *Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnect*, 2007.

[9] B. Rogers, A. Krishna, G. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," in *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[10] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th International Symposium on Computer Architecture*, 2011.

[11] G. Sun, C. J. Hughes, C. Kim, J. Zhao, C. Xu, Y. Xie, and Y.-K. Chen, "Moguls: a model to explore the memory hierarchy for bandwidth improvements," in *Proceedings of the 38th International Symposium on Computer Architecture*, 2011.

[12] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, 1970.

[13] C. Ding and T. Chilimbi, "A composable model for analyzing locality of multi-threaded programs," Technical Report MSR-TR-2009-107, Microsoft Research, 2009.

[14] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, "Is reuse distance applicable to data locality analysis on chip multiprocessors?," in *Proceeding of Compiler Construction*, 2010.

[15] X. Shi, F. Su, J.-K. Peir, Y. Xia, and Z. Yang, "Modeling and single-pass simulation of cmp cache capacity and accessibility," in *Proceedings of the International Symposium on Performance Analysis of Systems Software*, 2007.

[16] D. L. Schuff, B. S. Parsons, and J. S. Pai, "Multicore-aware reuse distance analysis," Technical Report TR-ECE-09-08, Purdue University, 2009.

[17] D. L. Schuff, M. Kulkarni, and V. S. Pai, "Accelerating multicore reuse distance analysis with sampling and parallelization," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[18] X. Xiang, B. Bao, C. Ding, and Y. Gao, "Linear-time modeling of program working set in shared cache," in *Proceedings of the 20th International Symposium on Parallel Architectures and Compilation Techniques*, 2011.

[19] M.-J. Wu and D. Yeung, "Coherent profiles: enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs," in *Proceedings of the 20th International Symposium on Parallel Architectures and Compilation Techniques*, 2011.

[20] Y. Zhong, S. G. Dropsho, and C. Ding, "Miss rate prediction across all program inputs," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[21] C. keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Programming Language Design and Implementation*, 2005.

[22] C. McCurdy and C. Fischer, "Using pin as a memory reference generator for multiprocessor simulation," *SIGARCH Computer Architecture News*, 2005.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[24] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: a benchmark suite for data mining workloads," in *Proceedings of the International Symposium on Workload Characterization*, 2006.

[25] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[26] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

[27] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. S. Vaidya, "Integration challenges and tradeoffs for tera-scale architectures," *Intel Technology Journal*, 2007.

[28] http://tilera.com/products/processors, "Processors from Tilera Corporation."

[29] 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 International Symposium on Computer Architecture*, 2009.

[30] M. Zhang and K. Asanovic, "Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.

[31] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 simulator: modeling networked systems," *IEEE Micro*, 2006.

[32] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Proceedings of the 15th International Symposium on Computer Architecture*, 1988.

[33] A. Gupta, W. dietrich Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Proceedings of the International Conference on Parallel Processing*, 1990.

[34] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS directories: a scalable cache coherence scheme," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[35] S. S. Mukherjee and M. D. Hill, "An evaluation of directory protocols for medium-scale shared-memory multiprocessors," in *Proceedings of the International Conference on Supercomputing*, 1994.

[36] A. Qasem and K. Kennedy, "Evaluating a model for cache conflict miss prediction," Technical Report CS-TR05-457, Rice University, 2005.

[37] D. Sanchez, G. Michelogiannakis, and C. Kozyrakis, "An analysis of on-chip interconnection networks for large-scale chip multiprocessors," *ACM Transactions on Architecture and Code Optimization*, 2010.

[38] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, 1985.

[39] Y. Zhong and W. Chang, "Sampling-based program locality approximation," in *Proceedings of the 7th International Symposium on Memory Management*, 2008.

[40] G. E. Suh, S. Devadas, and L. Rudolph, "Analytical cache models with applications to cache partitioning," in *Proceedings of International Conference on Supercomputing*, 2001.

[41] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2005.

[42] C. Ding and T. Chilimbi, "All-window profiling of concurrent executions," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[43] X. Xiang, B. Bao, T. Bai, C. Ding, and T. Chilimbi, "All-window profiling and composable models of cache sharing," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, 2011.

[44] Y. H. Kim, M. D. Hill, and D. A. Wood, "Implementing stack simulation for highly-associative memories," in *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991.

[45] M. Kulkarni, V. Pai, and D. Schuff, "Towards architecture independent metrics for multicore performance analysis," *ACM SIGMETRICS Performance Evaluation Review*, 2010.

[46] E. Berg and E. Hagersten, "Statcache: a probabilistic approach to efficient and accurate data locality analysis," in *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.

[47] E. Berg and E. Hagersten, "Efficient data-locality analysis of long-running applications," Technical Report TR 2004-021, University of Uppsala, 2004.

[48] D. Eklov and E. Hagersten, "Statstack: efficient modeling of lru caches," in *Proceedings of the International Symposium on Performance Analysis of Systems Software*, 2010.

[49] X. Shen, J. Shaw, B. Meeker, and C. Ding, "Locality approximation using time," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[50] X. Shen and J. Shaw, "Scalable implementation of efficient locality approximation," in *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 2008.

[51] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," in *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2005.

[52] E. Berg, H. Zeffer, and E. Hagersten, "A statistical multiprocessor cache model," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2006.

[53] M. Hirzel and T. Chilimbi, "Bursty tracing: a framework for low-overhead temporal profiling," in *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[54] K. Beyls and E. H. D'Hollander, "Platform-independent cache optimization by pinpointing low-locality reuse," in *Proceedings of the International Conference on Computational Science*, 2004.

[55] K. Beyls and E. H. D'Hollander, "Discovery of locality-improving refactoring by reuse path analysis," in *Proceedings of the International Conference on High Performance Computing and Communication*, 2006.

[56] K. Beyls, E. H. D'Hollander, and F. Vandeputte, "Rdvis: a tool that visualizes the causes of low locality and hints program optimizations," in *Proceedings of the International Conference on Computational Science*, 2005.

[57] K. Beyls and E. H. D'Hollander, "Intermediately executed code is the key to find refactorings that improve temporal data locality," in *Proceedings of the 3rd Conference on Computing Frontiers*, 2006.

[58] K. Beyls and E. H. D'Hollander, "Refactoring for data locality," *IEEE Computer*, 2009.

[59] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding, "Miss rate prediction across program inputs and cache configurations," *IEEE Transactions on Computers*, 2007.

[60] Y. Zhong, X. Shen, and C. Ding, "Program locality analysis using reuse distance," *ACM Transactions on Programming Languages and Systems*, 2009.

[61] M.-J. Wu and D. Yeung, "Scaling single-program performance on large-scale chip multiprocessors," Technical Report UMIACS-TR-2009-16, University of Maryland, 2009.

[62] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, 2008.

[63] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma, "Cache miss behavior: is it sqrt(2)?," in *Proceedings of the Conference Computing Frontiers*, 2006.

[64] T. Oh, H. Lee, K. Lee, and S. Cho, "An analytical model to study optimal area breakdown between cores and caches in a chip multiprocessor," in *Proceedings of the IEEE Computer Society Symposium on VLSI*, 2009.

[65] D. Wentzlaff, N. Beckmann, J. Miller, , and A. Agarwal, "Core count vs cache size for manycore architectures in the cloud," Technical Report MIT-CSAIL-TR-2010-008, Massachusetts Institute of Technology, 2010.

[66] A. Krishna, A. Samih, and Y. Solihin, "Impact of data sharing on cmp design: a study based on analytical modeling," in *Proceedings of the Second International Workshop on New Frontiers in High-performance and Hardware-aware Computing*, 2011.

[67] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *Proceedings of Architectural Support for Programming Languages and Operating Systems*, 2006.

[68] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of Architectural Support for Programming Languages and Operating Systems*, 2006.

[69] H. Cook and K. Skadron, "Predictive design space exploration using genetically programmed response surfaces," in *45th ACM/IEEE Conference on Design Automation (DAC)*, 2008.

[70] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, "Performance of multithreaded chip multiprocessors and implications for operating system design," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.

[71] Y. Jiang, K. Tian, and X. Shen, "Combining locality analysis with online proactive job co-scheduling in chip multiprocessors," in *Proceedings of the International Conference on High-Performance Embedded Architectures and Compilers*, 2010.

[72] www.intel.com/software/products/vtune/, "Intel VTune performance profiler and analyze."