To appear in *ACM Transactions on Computer System*.

Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs

MENG-JU WU, University of Maryland at College Park
DONALD YEUNG, University of Maryland at College Park

Reuse distance (RD) analysis is a powerful memory analysis tool that can potentially help architects study multicore processor scaling. One key obstacle, however, is that multicore RD analysis requires measuring *concurrent reuse distance (CRD)* and *private-LRU-stack reuse distance (PRD)* profiles across thread-interleaved memory reference streams. Sensitivity to memory interleaving makes CRD and PRD profiles architecture dependent, preventing them from analyzing different processor configurations. For loop-based parallel programs, CRD and PRD profiles *shift coherently* across RD values with core count scaling because interleaving threads are symmetric. Simple techniques can predict such shifting, making the analysis of numerous multicore configurations from a small set of CRD and PRD profiles feasible. Given the ubiquity of parallel loops, such techniques will be extremely valuable for studying future large multicore designs.

This article investigates using RD analysis to efficiently analyze multicore cache performance for loop-based parallel programs, making several contributions. First, we provide an in-depth analysis on how CRD and PRD profiles change with core count scaling. Second, we develop techniques to predict CRD and PRD profile scaling, in particular employing reference groups [Zhong et al. 2003] to predict coherent shift, demonstrating 90% or greater prediction accuracy. Third, our CRD and PRD profile analyses define two application parameters with architectural implications: C_{core} is the minimum shared cache capacity that "contains" locality degradation due to core count scaling, and C_{share} is the capacity at which shared caches begin to provide a cache-miss reduction compared to private caches. And fourth, we apply CRD and PRD profiles to analyze multicore cache performance. When combined with existing problem scaling prediction, our techniques can predict shared LLC MPKI (private L2 cache MPKI) to within 10.7% (13.9%) of simulation across 1,728 (1,440) configurations using only 36 measured CRD (PRD) profiles.

1. INTRODUCTION

Multicore processor performance depends in large part on how well programs utilize the on-chip cache hierarchy. In the past, many studies have tried to characterize multicore memory behavior [Davis et al. 2005; Hsu et al. 2005; Huh et al. 2001; Li et al. 2009; Li et al. 2006; Li and Martinez 2005; Rogers et al. 2009; Zhao et al. 2007]. These studies simulate processors with varying *core count* and *cache capacity* to quantify how different designs impact memory performance. A significant problem is the large number of configurations that must be explored due to the multi-dimensional nature of the design space. Worse yet, this design space is becoming larger as processors scale.

Today, eight state-of-the-art cores or 10s of smaller cores [Agarwal et al. 2007; Hoskote et al. 2007] along with 10s of MBs of cache can fit on a single die. Since Moore's law scaling is expected to continue at historic rates for the foreseeable future, processors with 100s of cores and 100+ MB of cache–*i.e.* large-scale chip multiprocessors (LCMPs) [Hsu et al. 2005; Zhao et al. 2007]–are conceivable after only 2 or 3 generations. As processors scale to the LCMP level, evaluating memory performance via simulation alone will become extremely challenging.

A powerful tool that can help address this problem is *reuse distance (RD) analysis*. RD analysis measures a program's memory reuse distance histogram, or *RD profile*, capturing the application-level locality responsible for cache performance. For sequential programs, RD profiles are *architecture independent*. Once acquired, they can be used to predict different cache sizes without additional program runs. This saves time

This research was supported in part by NSF under grant #CCF-1117042, and in part by DARPA under grant #HR0011-10-9-0009. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

by reducing the number of cache designs that need to be run or simulated. RD analysis has also been applied to parallel programs on multicore processors [Ding and Chilimbi 2009; Jiang et al. 2010; Schuff et al. 2009; Schuff et al. 2010; Wu and Yeung 2011]. For parallel programs, not only can RD analysis address cache scaling, it can potentially predict performance across core count scaling as well. This can provide even greater leverage to save time when evaluating different cache designs.

Compared to uniprocessors, though, RD analysis for multicore processors is much more complex. This is because locality in multithreaded programs depends not only on per-thread reuse, but also on how simultaneous threads' memory references interact. So, analyzing multicore workloads requires extending RD analysis to account for thread interactions. For example, *concurrent reuse distance (CRD) profiles* [Ding and Chilimbi 2009; Jiang et al. 2010; Schuff et al. 2009; Schuff et al. 2010; Wu and Yeung 2011] quantify reuse across thread-interleaved memory reference streams, and account for interference and data sharing between threads accessing shared caches. In addition, *private-LRU-stack reuse distance (PRD) profiles* [Schuff et al. 2009; Schuff et al. 2010] quantify reuse within per-thread memory reference streams under invalidation-based coherence, and account for replication and communication between threads accessing private caches.

A major problem is thread interactions are sensitive to inter-thread memory interleaving which is *architecture dependent*. In particular, scaling core count increases the number of memory streams that interleave. So, CRD and PRD profiles are not valid for machine sizes that differ from what was profiled. Even scaling cache capacity can alter relative thread speed and memory interleaving. So, strictly speaking, CRD and PRD profiles may not even be valid across different cache sizes at the *same* core count. Such architecture dependences prevent a single locality profile from analyzing different multicore configurations, defeating the time-saving benefits of RD analysis.

Recently, researchers have tried *predicting* multicore locality profiles across different core counts by analyzing and accounting for the effects of increased memory interleaving on reuse distance [Ding and Chilimbi 2009; Jiang et al. 2010]. The predicted profiles can then be used to predict cache performance for the scaled CPUs. Unfortunately, existing techniques are extremely costly, employing trace-based analyses to address the combinatorially large number of ways that threads' memory references can interleave. Moreover, the techniques require at-scale profiling and traces. As such, they are impractical for even moderately-sized machine/problem sizes, and completely out of the question for LCMPs.

In our work, we make the key observation that the complexity of predicting locality profiles depends on how programs are parallelized. Parallel programs generally express either *task-based* or *loop-based* parallelism. In task-based parallel programs, threads execute dissimilar code, giving rise to irregular memory interleavings. This tends to make CRD and PRD profiles highly unpredictable across different core counts. In loop-based parallel programs, however, simultaneous threads execute similar code–*i.e.*, from the same parallel loop–so they exhibit almost identical locality characteristics. Such *symmetric threads* produce regular memory interleavings, and cause CRD and PRD profiles to change systematically across core count scaling. Hence, the profiles of scaled-up CPUs can be directly predicted from the profiles of smaller-scale CPUs without having to perform trace-based interleaving analysis.

While techniques borne out of this observation will necessarily be specific to loop-based parallel programs, such workloads are pervasive. For example, data parallel codes–*e.g.*, scientific, media, and bioinformatics programs–derive all of their parallelism from loops. Programs written in OpenMP, one of the most popular parallel environments, consist almost entirely of parallel loops. In addition, loop-based parallel programs are also highly scalable. Most can provide large amounts of parallelism sim-

ply by increasing problem size, so they are a good match for LCMPs. For these reasons, we believe RD analysis for loop-based parallel programs will be extremely valuable to future multicore designers.

This article investigates RD analyses for loop-based parallel programs that leverage thread symmetry to accelerate CRD and PRD profile prediction. Our goal is to provide techniques that allow architects to rapidly assess cache performance in LCMP-sized machines without having to run at-scale simulations or to perform costly trace-based analyses. In fact, our techniques are fast enough to enable exhaustive evaluation of complete LCMP design spaces, yielding insights into multicore memory behavior that are impossible to obtain via simulation alone.

To realize this goal, our work makes several contributions. First, we provide an in-depth analysis on how CRD and PRD profiles from loop-based parallel programs change across different core counts. We find that both CRD and thread-aggregated PRD profiles *shift coherently–i.e.*, in a shape-preserving fashion–to larger RD values as core count increases. In CRD profiles, shifting slows down and eventually stops due to overlapping references to shared data. In thread-aggregated PRD profiles, shifting slows down due to invalidation-induced holes, but unlike CRD profiles, the shifting never fully stops. Inter-thread shared references also cause intercepts and invalidations in CRD and PRD profiles, respectively, that can spread or otherwise distort profiles, but coherent shift is by far the dominant behavior.

Second, we develop techniques to predict the systematic CRD/PRD profile movement. We employ reference groups [Zhong et al. 2003], a technique previously used to predict RD profiles across problem scaling, to predict coherent shifting. We also propose techniques to predict spreading and invalidation increases. Our techniques require obtaining profiles on 2 and 4 cores only, and can predict the CRD and PRD profiles at any core count very quickly (in seconds). To evaluate our techniques, we use the Intel PIN tool [Luk et al. 2005] to acquire CRD and PRD profiles across 9 benchmarks running 4 different problem sizes on 2–256 cores. We find our techniques can predict the measured CRD and PRD profiles with 90% and 96% accuracy, respectively.

Third, we study two insights into cache design that follow from our analyses. As mentioned earlier, CRD profile shifting slows down and stops beyond a certain point. This implies core count scaling only impacts shared cache performance below the stopping point, which we call C_{core}. Shared caches with capacity $> C_{core}$ can "contain" the locality degradation due to core count scaling. We measure C_{core} and find 16MB shared caches contain scaling up to 256 cores across most of our benchmarks and problem sizes. Another cache design insight follows from the fact that CRD profiles' shifting eventually slows down relative to thread-aggregated PRD profiles' shifting. This implies shared caches provide a cache-miss benefit over private caches only beyond the point where the CRD-PRD shifting rates diverge, which we call C_{share}. Below C_{share}, where private and shared caches incur the same cache-miss count, private caches will always achieve higher performance due to their lower access latencies. We measure C_{share} and find it is highly core count and problem size dependent, suggesting that no fixed assignment of private or shared caches across different caching levels is optimal for all cases. Instead, architectures that can adapt cache sharing may be the best.

Fourth, we demonstrate our techniques' ability to accelerate LCMP design space analysis. Using the M5 simulator [Binkert et al. 2006], we model a tiled CMP, and simulate our benchmarks on processors with 2–256 cores, 16-256KB private caches, and 4–128MB shared last-level caches (LLCs). In total, we simulate 3,168 different configurations split across two different design spaces–one that varies shared LLC size and another that varies private L2 cache size. Our core count prediction techniques can predict the MPKI (misses per kilo-instructions) for all configurations in the shared LLC and private L2 design spaces to within 9.5% and 12.5% of simulation, re-

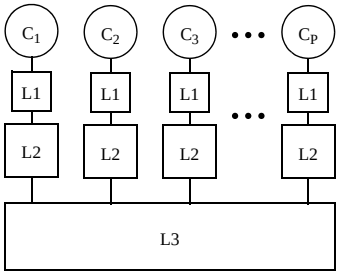

Fig. 1. Multicore cache hierarchy with two levels of private cache and a shared last-level cache.

spectively, using 72 measured profiles per design space. When combined with existing problem scaling prediction techniques, we can predict all configurations with similar accuracy using only 36 measured profiles per design space. Our M5 simulations also show symmetric threads' CRD and PRD profiles are relatively insensitive to cache capacity scaling, confirming that a single set of profiles can indeed be used to analyze different cache sizes.

Finally, we compare our profile prediction techniques against profile sampling [Schuff et al. 2010; Zhong and Chang 2008], an approach that can speedup individual profiling runs. Across our design space, we find prediction can acquire all profiles in less time than sampling while simultaneously achieving higher accuracy. However, the best choice between prediction and sampling depends on the user's needs. Prediction is preferable when evaluating large design spaces requiring numerous profiles whereas sampling is preferable for quickly acquiring a few profiles.

The rest of this article is organized as follows. Section 2 discusses CRD and PRD profiles, and Section 3 shows how they change with core count scaling. Then, Section 4 develops techniques to predict the scaling changes, and Section 5 presents our cache design insights. Next, Section 6 demonstrates our techniques' ability to accelerate cache evaluation. Lastly, Section 7 presents our profile sampling study. Sections 8 and 9 end with related work and conclusions.

2. MULTICORE RD ANALYSIS

Reuse distance (also known as LRU stack distance [Gecsei et al. 1970]) measures the number of unique memory references performed between two references to the same data block. RD profiles–*i.e.*, the histogram of RD values for all references in a sequential program–are useful for analyzing uniprocessor cache performance. Because a cache of capacity C can satisfy references with RD $< C$ (assuming a basic LRU replacement policy), the number of cache misses is the sum of all reference counts in an RD profile above the RD value for capacity C.

This paper studies RD analysis for multicore processors. Figure 1 illustrates a modern multicore cache hierarchy. Such cache hierarchies often integrate a combination of private and shared caches on chip, with private caches employed near the cores and shared caches employed near the off-chip interface. For example, Figure 1 shows a processor with two levels of private cache backed by a single last-level cache (LLC) that is shared by all the cores.

To enable RD analysis for multicore processors, researchers have developed new notions of reuse distance that account for the thread interactions occurring within multicore caches. Because inter-thread interactions differ depending on the cache type, two forms of RD–*concurrent reuse distance* and *private-LRU-stack reuse distance*–have been used to analyze shared and private caches separately. The rest of this section

describes CRD and PRD in detail, as well as the different interaction effects each notion of reuse distance captures.

2.1. Concurrent Reuse Distance

CRD is the reuse distance measured across interleaved memory reference streams from multiple cores, and represents the locality seen by a shared cache referenced by the interleaved streams. CRD profiles can be computed by assuming an interleaving between threads' memory references, and applying the interleaved stream on a single (global) LRU stack [Jiang et al. 2010; Schuff et al. 2009; Schuff et al. 2010; Wu and Yeung 2011]. To illustrate, consider the interleaved memory references from two cores, C_1 and C_2, shown in Figure 2. In Figure 2, C_1 references blocks A–E, and then re-references A, while C_2 references blocks C and F–H. Figure 3a shows the state of the global LRU stack when C_1 re-references A. C_1's reuse of A exhibits an intra-thread RD of 4, but the CRD which accounts for interleaving is 7. In this case, CRD $>$ RD because some of C_2's interleaving references (F–H) are distinct from C_1's references, causing *dilation* of intra-thread reuse distance.

In many multithreaded programs, threads share data, which can offset dilation in two ways. First, it can introduce *overlapping references*. For example, in Figure 2, while C_2's reference to C interleaves with C_1's reuse of A, this does not increase A's CRD because C_1 already references C in the reuse interval. Second, data sharing can also introduce *intercepts*. For example, if C_2 references A instead of C at time 6 in Figure 2, then C_1's reuse of A has CRD = 3, so CRD actually becomes *less* than RD.

Dilation, overlap, and intercepts are the 3 forms of inter-thread interaction that occur in shared caches. CRD profiles reflect the impact of these interactions on shared cache locality.

2.2. Private-LRU-stack Reuse Distance

PRD is the reuse distance measured within per-thread memory reference streams that maintain coherence, and represents the locality seen by coherent private caches referenced by the per-thread streams. PRD profiles can be computed by assuming an interleaving between threads' memory references, and applying the per-thread streams on coherent private LRU stacks [Schuff et al. 2009; Schuff et al. 2010]. In the absence of writes, the private stacks do not interact, so PRD is the same as per-thread RD. To illustrate, Figure 3b shows the state of the private LRU stacks when C_1 re-references A in Figure 2 assuming all references are reads. For C_1's reuse of A, PRD = RD = 4. Note, however, the multiple private stacks still contribute to increased cache capacity. In Figure 3b, the total capacity needed to keep A in cache is 10–*i.e.*, 2 caches with 5 cache blocks each. (We assume per-core private caches are always the same size). We call the aggregate cache capacity needed to capture a particular reuse the *scaled PRD*, or sPRD. sPRD $= T \times PRD$, where T is the number of threads and PRD is the private-LRU-stack reuse distance. Unlike PRD which only reflects capacity in a single private cache, sPRD can be meaningfully compared against CRD since both reflect total cache capacity. For C_1's reuse of A in Figure 3, we see sPRD $>$ CRD.

Similar to CRD, data sharing also affects PRD. In particular, read sharing causes *replication*, increasing overall capacity pressure. Figure 3b illustrates this, showing duplication of C in the private stacks. Because PRD scaling aggregates private LRU stack contents, replication effects are automatically captured in sPRD profiles. In contrast, write sharing causes inter-stack communication. For example, consider what would happen if C_2's reference to C were a write instead of a read. Because PRD stacks maintain coherence, a coherence operation is needed for C_1's copy of C. One option is to *invalidate* the copy, as shown in Figure 3c. (This is the approach taken by existing PRD techniques [Schuff et al. 2009; Schuff et al. 2010] because invalidation is the most com-

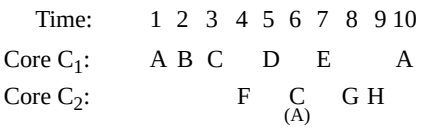

Fig. 2. Two interleaved memory reference streams, illustrating different interactions among inter-thread memory references.

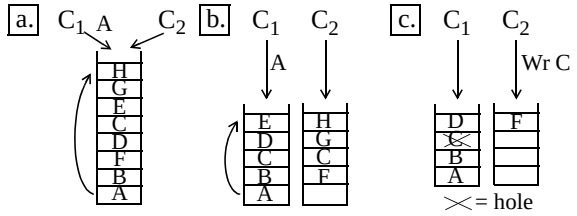

Fig. 3. LRU stacks showing (a) dilation and overlap in CRD profiles, (b) scaling, replication, and (c) holes in PRD profiles.

mon mechanism for maintaining coherence in today's CPUs). To prevent invalidations from promoting blocks further down the LRU stack, invalidated blocks become *holes* rather than being removed from the stack [Schuff et al. 2009]. Holes are unaffected by references to blocks above the hole (*e.g.*, if C_1 were to reference D in Figure 3c), but a reference to a block below the hole moves the hole to where the referenced block was found. For example, if C_1 were to reference B in Figure 3c, D would be pushed down and the hole would move to depth 3, preserving A's stack depth.

While invalidations reduce locality for victimized data blocks since reuse of these blocks will always cache miss (*i.e.*, coherence misses), they can also improve locality because the holes they leave behind eventually absorb stack demotions. For example, in Figure 3c, C_1's subsequent reference to E does not increase A's stack depth because D will move into the hole. (Since this is E's first reference, in effect, the hole moves to RD = ∞). Hence, C_1's reuse of A has PRD (sPRD) = 3 (6) instead of 4 (8). We call this effect *demotion absorption*.

PRD scaling (with replication), demotion absorption, and invalidations are the 3 forms of inter-thread interaction that occur in private caches. PRD profiles, as well as sPRD profiles, reflect the impact of these interactions on private cache locality.

3. CORE COUNT SCALING IMPACT

This section studies the impact of core count scaling on multicore locality profiles. In particular, we analyze how dilation, overlap, and intercepts vary in CRD profiles and how PRD scaling, demotion absorption, and invalidations vary in PRD/sPRD profiles across different machine sizes. Section 3.1 describes how we acquire locality profiles, including techniques for quantifying inter-thread interactions. The latter addresses the problem that CRD and PRD profiles reflect the combined effects of all sources of thread interaction; our profiling techniques isolate the individual contributions, thus explicitly accounting for each type of inter-thread interaction. Next, Section 3.2 presents our core count scaling insights for CRD profiles, and then Section 3.3 presents the same for PRD/sPRD profiles. Notice, the core count scaling analyses in these sections assume problem size is fixed. Later, we will consider problem scaling as well, but core count and problem scaling are always handled separately in our work.

3.1. Profiling

To facilitate our study, we acquire multicore RD profiles using the Intel PIN tool [Luk et al. 2005]. We modified PIN to maintain a global LRU stack to acquire CRD profiles and coherent private LRU stacks to acquire PRD profiles. The acquired PRD profiles are then scaled to derive the sPRD profiles. We also maintain a non-coherent LRU stack for each thread to acquire per-thread RD profiles. We assume 64-byte memory blocks in all LRU stacks. When a thread performs a data reference, PIN computes the block's depth in the global stack and the coherent and non-coherent private stacks.

These stack depths update CRD, PRD, and per-thread RD profiles, respectively. Then, the referenced blocks are moved to the MRU stack position. Our PIN tool follows McCurdy's method [McCurdy and Fischer 2005] and performs functional execution only, context switching between threads after every memory reference. This interleaves threads' references uniformly in time. (Later, we will consider timing effects).

Because inter-thread interactions occur within individual parallel loops, we record profiles on a per-loop basis. In our benchmarks, parallel loops usually begin and end at barriers. Our PIN tool records profiles in between every pair of barrier calls–*i.e.*, per parallel region. Multiple loops can occur within a single parallel region so this does not isolate all parallel loops, but it is sufficient for our study.

Within each parallel region, we acquire CRD and PRD profiles for references to mostly private versus shared data separately. (Note, we still keep all data in the same LRU stacks from Figure 3; we only register computed CRD/PRD values in separate profiles). We call the private profiles CRD_P and PRD_P, and the shared profiles CRD_S and PRD_S. We also scale the separated PRD profiles to obtain sPRD versions, called sPRD_P and sPRD_S. CRD_P and PRD_P do not include interactions directly associated with shared data–*i.e.*, intercepts and invalidations, respectively. So, CRD_P shows dilation and overlap effects while PRD_P–and in particular, sPRD_P–shows PRD scaling and demotion absorption effects. CRD_S and PRD_S/sPRD_S primarily show intercept and invalidation effects, respectively.

To acquire all of these profiles, we record each memory block's CRD and PRD values separately[1] as well as the number of times the block is referenced by each core. After a parallel region completes, we determine each block's sharing status: if a single core is responsible for 90% or more of a block's references, the block is private; otherwise, it is shared. We then accumulate all memory blocks' CRD (PRD) counts into either the CRD_P (PRD_P) or CRD_S (PRD_S) profiles based on their observed sharing.[2]

Next, we isolate sharing-based interactions within the private profiles. We maintain a second global LRU stack and a second set of coherent private LRU stacks in which we artificially remove sharing effects. For CRD, we prepend every memory reference's address with the ID of the core performing the reference (tracking private vs shared blocks still uses the unmodified addresses), and apply this CID-extended reference stream on the second global LRU stack. We compute CRD_P profiles from the CID-extended stack exactly as described above. We call these profiles CRD_{PC}. In CRD_{PC} profiles, inter-thread references are always unique, so there is never any overlap. Thus, comparing CRD_{PC} and CRD_P profiles shows the impact of the overlapping references.

For PRD, we remove write sharing by converting all writes into reads, and apply this read-converted reference stream on the second set of coherent private LRU stacks. Then, we compute PRD_P profiles (and sPRD_P profiles via scaling) from the read-converted stacks exactly as described above. We call these profiles PRD_{PR} and sPRD_{PR}. In these profiles, there are no invalidations, so there are never any holes. Thus, comparing PRD_{PR}/sPRD_{PR} profiles against PRD_P/sPRD_P profiles shows the impact of demotion absorption.

3.2. Scaling Impact on CRD Profiles

Figures 4 and 5 show how core count scaling affects CRD profiles using FFT from the SPLASH2 benchmark suite [Woo et al. 1995] as an example. CRD profiles are presented for the most important parallel region in FFT. Each profile plots reference count

[1]Individual memory blocks tend to exhibit a small number of distinct CRD and PRD values, so this bookkeeping does not increase storage appreciably.

[2]The 90% threshold ensures CRD_S/PRD_S profiles reflect mostly shared references. Small changes to the threshold will affect these profiles, but Section 4.3 will show that this will have no impact on our results.

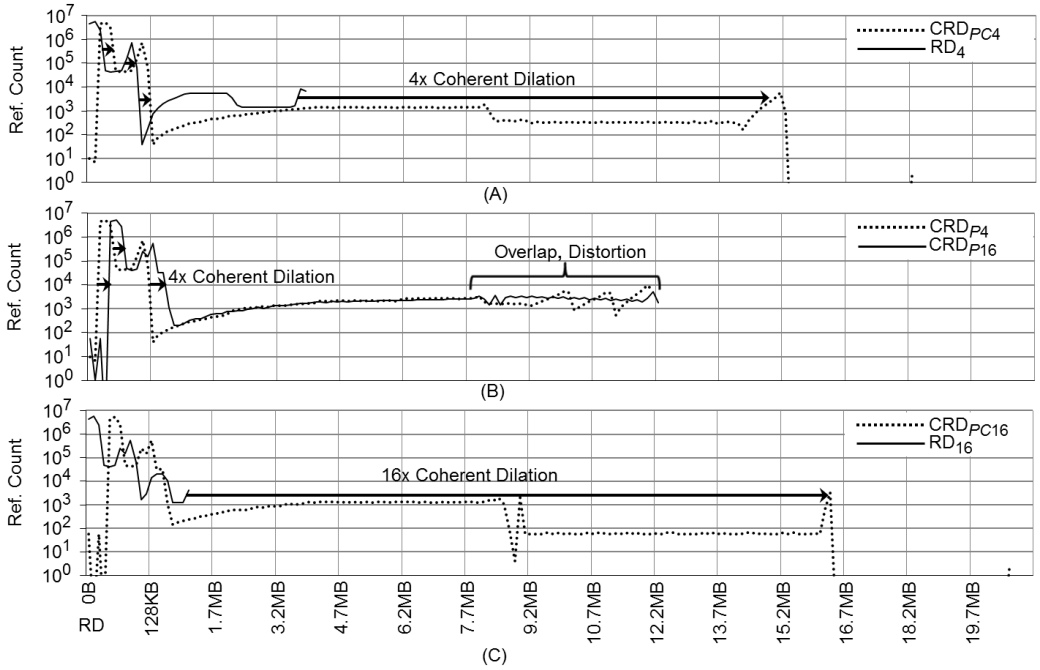

Fig. 4. FFT's CRD$_{PC}$ and RD profiles for 4 (A) and 16 (C) cores; CRD$_P$ profiles for 4 and 16 cores (B).

(Y-axis) versus CRD (X-axis). CRD values are multiplied by the block size, 64 bytes, so the X-axis reports reuse distance in terms of capacity. For each profile, reference counts from multiple adjacent CRD values are summed into a single CRD 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. Dilation and Overlap. Figure 4A plots FFT's CRD$_{PC}$ profile for a 4-core execution (labeled "CRD$_{PC4}$") along with the summation of per-thread RD profiles for 4 threads (labeled "RD$_4$"). As discussed in Section 3.1, CRD$_{PC}$ removes overlap and intercept effects, so comparing CRD$_{PC}$ against RD shows the impact of dilation alone on CRD with respect to per-thread profiles.

From Figure 4A, we can see CRD$_{PC4}$ shifts RD$_4$ to larger CRD values. Here, the dilation factor is 4x. More importantly, the 4x dilation is uniform across all RD values, making CRD$_{PC4}$ an expanded version of RD$_4$. To quantify the effect, we measured the dilation between several corresponding points (*i.e.*, at similar reference counts) on the RD$_4$ and CRD$_{PC4}$ profiles. For the 4 "shift points" indicated by the arrows in Figure 4, the shift rates are 4.1x, 4.0x, 4.0x, and 4.2x (leftmost to rightmost point). We call such consistent dilation *coherent shift*. Coherent shift is shape preserving, allowing the shifted profile to be a distortion-free version of the original profile.

Per-thread RD dilation is coherent because all interleaving threads are from the same parallel loop with very similar locality. For a particular intra-thread reuse at distance RD, the other $P-1$ threads in a P-core execution tend to interleave RD unique memory blocks each (due to thread symmetry), so CRD $\approx P \times RD$.

Overlap offsets dilation, reducing its shift. To illustrate, Figure 4B plots FFT's CRD$_P$ profile (labeled "CRD$_{P4}$"). This shows the combined impact of dilation and overlap at 4 cores. CRD$_{P4}$ and CRD$_{PC4}$ are almost identical at small CRD, but CRD$_{P4}$ exhibits less shift at large CRD due to the overlapping references. In our benchmarks, overlap

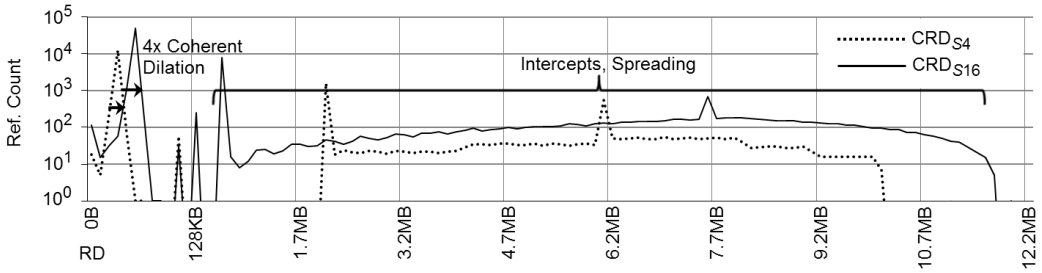

Fig. 5. FFT's CRD_S profiles for 4 and 16 cores.

affects mostly larger CRD because data sharing tends to occur across distant loop iterations. So, overlapping references appear in large reuse windows, but rarely in small reuse windows. More importantly, when it does occur, overlap introduces some distortion but the impact is minimal. So, CRD_{P4} is still a coherent scaling of RD_4.

Because dilation and overlap induce coherent per-thread RD shift, across different core counts, CRD profiles tend to shift coherently as well. To illustrate, Figure 4B plots the CRD_P profile for a 16-core FFT (labeled "CRD_{P16}"), and Figure 4C plots the corresponding RD_{16} and CRD_{PC16} profiles. As in the 4-core case, CRD_{P16} coherently scales RD_{16}, especially at small CRD where overlap rarely occurs. Notice, RD_{16} and RD_4 exhibit the same shape. This is because threads on 16 and 4 cores execute the same loop iterations, so they have similar locality. Hence, CRD_{P16} is not only a coherent shift of RD_{16}, it is also a coherent shift of RD_4 and CRD_{P4}. This time, however, scaling is by a factor 16x. As a result, the net effect is CRD_{P16} coherently scales CRD_{P4} by a factor 4x across the smaller CRD values. (For example, the 3 shift points indicated in Figure 4B all exhibit shift rates of 4.0x).

At larger CRD values, shifting slows down and eventually stops. Although RD_{16} and RD_4 exhibit the same shape, RD_{16} ends earlier because individual threads execute fewer loop iterations, eliminating distant reuse references. This truncation, along with the overlapping references at large CRD, almost perfectly cancel the additional 4x dilation. So in Figure 4B, CRD_{P16} and CRD_{P4} eventually merge, and end at about the same CRD value. This makes sense: because core count scaling does not change the amount of global data, the maximum CRD across different core counts is the same.

To summarize: *core count scaling causes the CRD_P profile of loop-based parallel programs to shift coherently by the scaling factor at small CRD values. Shifting slows down, and eventually stops at large CRD values.* While our detailed analysis is for FFT specifically, we find the same behavior is pervasive across all the benchmarks and loops we studied (see Table I). In particular, all of our benchmarks exhibit similar coherent shift points across different RD values.

3.2.2. Intercepts. Figure 5 plots FFT's CRD_S profiles at 4 and 16 cores (labeled "CRD_{S4}" and "CRD_{S16}," respectively). These profiles show how intercepts change with core count scaling. Notice, shared profiles contain significantly fewer references than private profiles. In particular, CRD_{S4} and CRD_{S16} in Figure 5 contribute only 1% more references compared to their corresponding private profiles in Figure 4B. While the exact balance is application dependent, we find private profiles always dominate shared profiles in our benchmarks. Hence, the insights presented in Section 3.2.1 for private profiles constitute the first-order effects of core count scaling on CRD profiles.

That said, it is still instructive to study intercepts. The CRD_{S4} and CRD_{S16} profiles in Figure 5 exhibit two different patterns. First, at small CRD ($<$ 128KB), core count scaling induces a coherent shift of CRD_{S4} by a factor 4x, similar to CRD_P. (For ex-

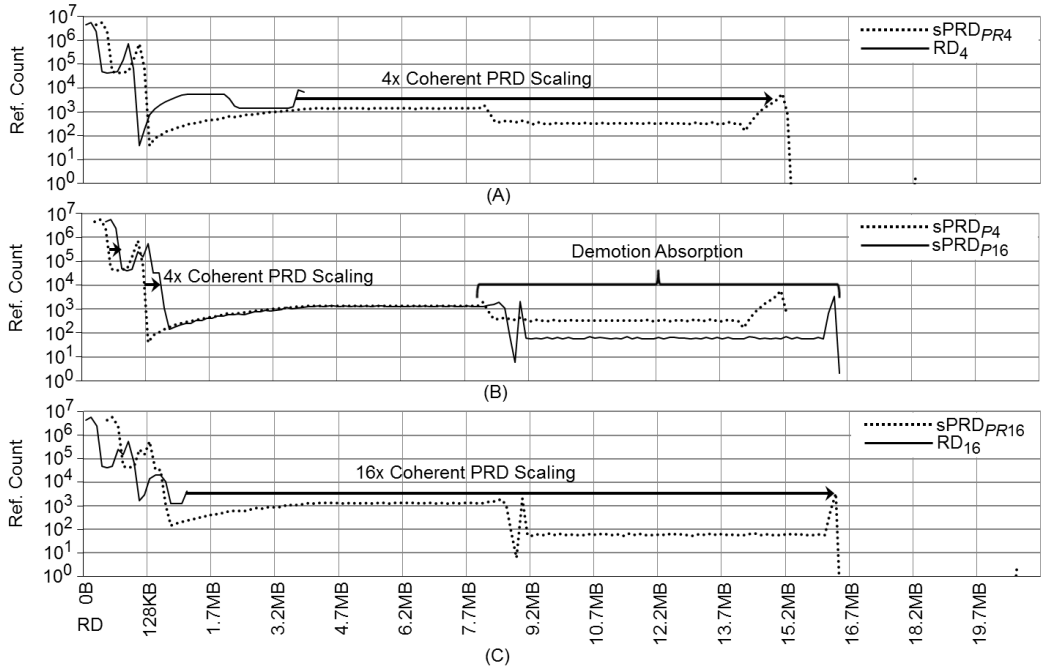

Fig. 6. FFT's sPRD$_{PR}$ and RD profiles for 4 (A) and 16 (C) cores; sPRD$_P$ profiles for 4 and 16 cores (B).

ample, the two shift points indicated in Figure 5 exhibit shift rates of 3.9x and 4.1x). As mentioned earlier, intercepts rarely appear within small reuse windows. Without intercepts, CRD$_S$ scales like CRD$_P$. And second, at larger CRD ($>$ 128KB), core count scaling induces spreading. The spreading stretches CRD$_{S4}$ towards both smaller and larger CRD values. In this portion of the CRD$_S$ profile, intercepts occur frequently and change CRD. By how much depends on *where* intercepts appear within intra-thread reuse windows. For example, Figure 2 shows an intercept almost bisecting thread 1's reuse, making CRD = 3. But if the intercept occurs at time 9, CRD = 0, and if the intercept occurs at time 2, CRD = 6. In general, intercepts spread references with per-thread reuse at distance RD between 0 and $T \times RD$, where T is the number of threads.

3.3. Scaling Impact on PRD/sPRD Profiles

Figures 6 and 7 show how core count scaling affects PRD for the same FFT parallel region from Section 3.2. To account for replication and PRD scaling impact on overall cache capacity, Figures 6 and 7 illustrate our insights on sPRD profiles, and then later relates the observed behaviors back to PRD profiles. Each profile in Figures 6 and 7 plots reference count versus sPRD using the same format as Figures 4 and 5.

3.3.1. PRD Scaling and Demotion Absorption. Because sPRD is by definition a scaling of PRD, sPRD profiles are perfectly expanded versions of per-thread locality profiles. To illustrate, Figure 6A plots FFT's sPRD$_{PR}$ profile for a 4-core execution (labeled "sPRD$_{PR4}$") along with the same RD$_4$ profile from Figure 4A. Since sPRD$_{PR}$ removes demotion absorptions and invalidations (see Section 3.1), comparing sPRD$_{PR}$ against RD shows the impact of PRD scaling alone with repsect to per-thread profiles. As expected, Figure 6A shows sPRD$_{PR4}$ shifts RD$_4$ by a factor 4x, preserving RD$_4$'s shape exactly. Notice, this is the same behavior observed for CRD$_{PC}$ in Figure 4A. In other

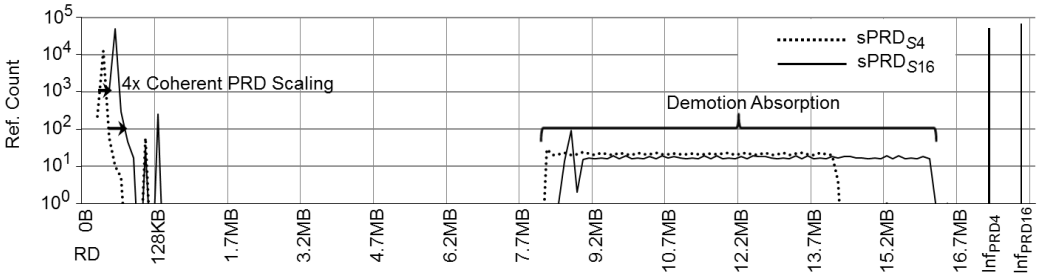

Fig. 7. FFT's sPRD$_S$ profiles for 4 and 16 cores.

words, dilation in CRD profiles and PRD scaling in sPRD profiles are similar effects: both cause coherent profile shift in proportion to the degree of core count scaling.

Figure 6B plots FFT's sPRD$_P$ profile for a 4-core execution (labeled "sPRD$_{P4}$"). As discussed in Section 3.1, this profile includes the effects of both PRD scaling and demotion absorption. Comparing Figures 6A and B, we see the sPRD$_{P4}$ and sPRD$_{PR4}$ profiles are almost identical, suggesting that demotion absorption has very little impact on top of PRD scaling. As discussed in Section 2.2, demotion absorption arises from holes created by invalidations. FFT has very little write sharing, so demotion absorptions are negligible in this application. For applications with more write sharing, demotion absorption can reduce shifting, but we find the contraction is usually small. Hence, sPRD$_P$ profiles are very similar to sPRD$_{PR}$ (and CRD$_{PC}$) profiles. Moreover, compared to CRD$_P$ profiles, sPRD$_P$ profiles always have significantly longer tails, and thus exhibit worse locality. (This is an important point we will revisit in Section 5.2).

Because the main effect in sPRD$_P$ profiles is PRD scaling which is analogous to dilation, sPRD$_P$ profiles also shift coherently across different core counts much like CRD$_P$ profiles. To illustrate, Figure 6B plots the sPRD$_P$ profile for a 16-core FFT (labeled "sPRD$_{P16}$"), and Figure 6C plots the corresponding RD$_{16}$ and sPRD$_{PR16}$ profiles (similar to Figures 4B and C). In Figure 6B, we can see sPRD$_{P16}$ shifts sPRD$_{P4}$ coherently by a factor 4x at small sPRD values. (For example, the two shift points indicated in Figure 6 both exhibit shift rates of 4.0x). As explained in Section 3.2.1, RD$_4$ and RD$_{16}$ are essentially the same profile at small reuse values. Since sPRD$_{P16}$ shifts RD$_{16}$ by a factor 16x (see Figure 6C), it is also a 16x shift of RD$_4$–and hence, a 4x shift of sPRD$_{P4}$–in this small-sPRD region. At larger sPRD values, Figure 6B shows sPRD$_{P16}$'s shift relative to sPRD$_{P4}$ slows down. As explained in Section 3.2.1, RD$_{16}$ ends earlier than RD$_4$, reducing sPRD$_{P16}$'s termination. However, whereas CRD$_{P4}$ and CRD$_{P16}$ co-terminate, sPRD$_{P16}$ still maintains some shifting relative to sPRD$_{P4}$ at large sPRD due to the smaller degree of contraction caused by demotion absorption compared to overlap.

To summarize: *core count scaling causes the sPRD$_P$ profile of loop-based parallel programs to shift coherently by the scaling factor at small sPRD values. Shifting slows down at large sPRD values, but unlike CRD$_P$, never stops completely. As a result, sPRD$_P$ profiles have longer tails (worse locality) than CRD$_P$ profiles.* While our detailed analysis is for FFT specifically, we find the same behavior is pervasive across all the benchmarks and loops we studied (see Table I). In particular, all of our benchmarks exhibit similar coherent shift points across different RD values.

Notice, our analysis for sPRD$_P$ profiles also reveals PRD$_P$ profile behavior since the latter is just a scaled-down version of the former. Like sPRD$_P$ profiles, PRD$_P$ profiles also shift coherently, but the shifting amount is reduced by a factor $\frac{1}{T}$. So, at small PRD values, PRD$_P$ profiles *do not shift at all*. At larger PRD values, shifting commences, but *towards smaller PRD values.* In other words, PRD$_P$ profiles contract towards the left whereas sPRD$_P$ profiles expand towards the right.

3.3.2. Invalidations. Figure 7 plots FFT's sPRD_S profiles at 4 and 16 cores (labeled "sPRD_{S4}" and "sPRD_{S16}," respectively). As with CRD_S profiles, sPRD_S profiles also exhibit the behaviors observed for private profiles discussed in Section 3.3.1. In Figure 7, we see sPRD_S profiles shift proportionally with core count, especially at small sPRD values, due to PRD scaling. (For example, the two shift points indicated in Figure 7 exhibit shift rates of 3.7x and 4.0x). Then, shifting slows down at larger sPRD due to the early termination of per-thread RD profiles with increasing core count. And although not shown in Figure 7, the same behavior–but inverted–occurs for PRD_S profiles: no shifting at small PRD values, and then leftward contraction at larger PRD.

In addition, Figure 7 also reports the number of references with $\text{sPRD} = \infty$ for 4 and 16 cores in the rightmost bars, labeled "Inf_{PRD4}" and "Inf_{PRD16}," respectively. These bars reflect the coherence misses that occur from invalidations. The example parallel region from FFT studied in this section incurs relatively few coherence misses. But notice the number of coherence misses increases from 4 to 16 cores (50,706 misses at 4 cores and 69,210 misses at 16 cores). As core count increases, the amount of sharing increases as well, resulting in more inter-thread communication and hence, more invalidations. In general, we find invalidations increase at a sub-linear rate with core count scaling (as in Figure 7), though the exact behavior is application dependent.

Recall from Section 3.2.2 that shared profiles contain significantly fewer references than private profiles. This is also true for PRD_S/sPRD_S profiles compared to PRD_P/sPRD_P profiles. So, the first-order effects of core count scaling on sPRD profiles follows the insights we've already presented in Section 3.3.1 for the private profiles. The behaviors shown in Figure 7 for shared profiles represent second-order effects.

4. PROFILE PREDICTION

This section studies techniques to predict CRD and PRD/sPRD profiles across core count scaling. We describe our prediction techniques in Section 4.1. Then, Section 4.2 discusses evaluation methodology. Finally, Section 4.3 presents the accuracy of our predictions and Section 4.4 assesses the speedup that prediction provides.

4.1. Prediction Techniques

Sections 3.2 and 3.3 show different multicore locality profiles change in different ways across core count scaling. CRD_P and PRD_P/sPRD_P profiles primarily exhibit coherent shift. CRD_S and PRD_S/sPRD_S profiles exhibit coherent shift as well. But in addition, CRD_S profiles also exhibit spreading while PRD_S/sPRD_S profiles also exhibit invalidation increases. The next three sections (4.1.1–4.1.3) develop techniques to predict these behaviors.

4.1.1. Coherent Shift. We begin by discussing the prediction of coherent shift. Our approach adopts Zhong's *reference groups* technique [Zhong et al. 2003]. Zhong found RD profiles for sequential programs exhibit shape-preserving shifting due to problem scaling similar to what we have observed for core count scaling, and proposed reference groups to predict such coherent shift. In this paper, we use reference groups to directly predict CRD_P and PRD_P profiles across different core counts. We also use reference groups to predict the shifting portions of CRD_S and PRD_S profiles across different core counts as well. Figure 8 illustrates the technique.

Profiles (either CRD_P, PRD_P, CRD_S, or PRD_S) are acquired at 2 and 4 cores. These measured profiles are divided into 200,000 groups along their X-axis, each containing an equal fraction (0.0005%) of the profile's references. Reference groups across measured profiles are *aligned* via association: the i^{th} group in the 2-core profile is aligned to the i^{th} group in the 4-core profile. Aligned reference groups "correspond" to each other across the shift and are assumed to shift together (*i.e.*, coherently), maintaining

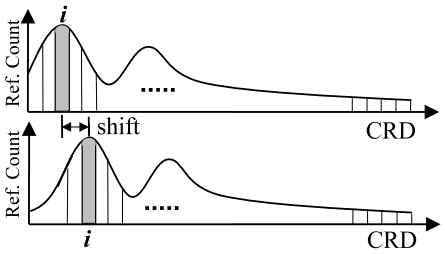

Fig. 8. Detecting alignment and shifting using reference groups.

a fixed shift rate dependence on scaling. This dependence is at least constant (no shift with core count), and at most linear shift with core count. Non-constant shift can be either in the positive direction for CRD profiles or in the negative direction for PRD profiles. In addition, we allow 98 different shift rates in between constant and linear. Specifically, for the 2x increase from 2 to 4 cores, we permit shift rates of 2^k, where $k = 0.0, 0.01, 0.02, ...0.98, 0.99, 1.0$. These different shift rates support variable shift.

The shift between pairs of reference groups in the acquired profiles is measured and compared against each allowed shift rate. The one with the closest match is assigned to the reference group. A prediction is made by shifting each reference group by its shift rate and desired core count scaling factor. The predictions from the measured CRD_P and PRD_P profiles are the predicted CRD_P and PRD_P profiles, respectively. The prediction from the measured CRD_S profile, which we call CRD_{Sshift}, is combined with spread prediction to derive predicted CRD_S profiles. Lastly, the prediction from the measured PRD_S profile, which we call PRD_{Sshift}, is combined with invalidation-increase prediction to derive predicted PRD_S profiles. (See below).

4.1.2. Spreading. Section 3.2.2 shows intercepts spread CRD_S profiles, with individual reuses moving to CRD values between 0 and $P \times RD$. As we will see later, 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 measure the CRD_S profile at 4 cores (the same measured profile used in shift prediction), and uniformly distribute the reference counts ($CRD_{S\_4core}[k] \times p[k]$) at each CRD between 0 and $k \times \frac{core\_count}{4}$, where k is a particular CRD value, and $p[k] = \frac{k}{C_{max}}$ (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:

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

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

4.1.3. Invalidation Increase. Except for invalidations at PRD $= \infty$, the prediction for PRD_{Sshift} *is* the predicted PRD_S profile. At PRD $= \infty$, the number of invalidations increases with core count (as discussed in Section 3.3.2, it usually does so sub-linearly). As we will see later, the actual increase is application dependent. From our observations across our benchmarks and problem sizes, we make the simplifying assumption that the number of invalidations grows *logarithmically* with core count. Hence, to predict PRD_S, we simply add to the measured invalidation count at 4 cores the observed increase from 2 to 4 cores multipled by the log-base-2 of the scaling factor, and set the reference bin at ∞ in PRD_{Sshift} to this predicted invalidation count.

Table I. Parallel benchmarks used in our study.

Benchmark	Problem Sizes (S1/S2/S3/S4)	Insts Profiled(M) (PIN) (S1/S2/S3/S4)	Insts Profiled(M) (M5) (S1/S2/S3/S4)
FFT	$2^{16}/2^{18}/2^{20}/2^{22}$ elements	29/129/560/2,420	32/139/605/2,610
LU	$256^2/512^2/1024^2/2048^2$ elements	43/344/2,752/22,007	72/577/4,625/37,027
RADIX	$2^{18}/2^{20}/2^{22}/2^{24}$ keys	53/211/843/3,372	54/216/864/3,456
Barnes	$2^{13}/2^{15}/2^{17}/2^{19}$ particles	214/1,015/4,438/19,145	614/2,909/12,798/55,233
FMM	$2^{13}/2^{15}/2^{17}/2^{19}$ particles	235/1,006/4,109/16,570	217/931/3,793/15,305
Ocean	$130^2/258^2/514^2/1026^2$ grid	30/107/420/1,636	36/126/494/1,925
Water	$10^3/16^3/25^3/40^3$ molecules	43/143/553/2,099	54/174/642/2,315
KMeans	$2^{16}/2^{18}/2^{20}/2^{22}$ objects, 18 features	186/742/2,967/11,874	246/985/3,939/15,748
BlackScholes	$2^{16}/2^{18}/2^{20}/2^{22}$ options	60/242/967/3,867	94/376/1,506/6,023

4.2. Prediction Methodology

We use PIN to acquire the CRD and PRD profiles for our study. In particular, we acquire the per-parallel region CRD_P, CRD_S, PRD_P, and PRD_S profiles, as described in Section 3.1, for 2- and 4-core executions. During profiling, we accumulate profiles for different dynamic instances of the same static parallel region into a single pair of CRD_P and CRD_S (PRD_P and PRD_S) profiles. Then, we use the techniques from Section 4.1 to predict the CRD_P and CRD_S (PRD_P and PRD_S) profiles for 8–256 cores, in powers of 2, from the measured 2- and 4-core profiles. At each core count, we sum all predicted per-parallel region CRD_P and CRD_S (PRD_P and PRD_S) profiles to form a single prediction for the whole-program CRD (PRD) profile.

As discussed in Section 3.2, CRD_P and PRD_P profiles dominate CRD_S and PRD_S profiles. This implies predicting coherent shift alone may be sufficient. In addition to the above methodology, we also employ whole-program CRD and PRD profile prediction. We use PIN to acquire the whole-program CRD and PRD profiles at 2 and 4 cores. Then, we use the reference groups technique from Section 4.1.1 to predict the whole-program profiles for 8–256 cores directly from the measured whole-program profiles. This approach requires much less bookkeeping during profiling.

Our study employs 9 benchmarks, each running 4 problem sizes. Table I lists the benchmarks: FFT, LU, RADIX, Barnes, FMM, Ocean, and Water from SPLASH2 [Woo et al. 1995], KMeans from MineBench [Narayanan et al. 2006], and BlackScholes from PARSEC [Bienia et al. 2008b]. (The interested reader can find detailed descriptions of these benchmarks in [Bienia et al. 2008a; Barrow-Williams et al. 2009]). The 2^{nd} column of Table I specifies the 4 problem sizes, S1–S4. For each benchmark and problem size, we predict the whole-program CRD and PRD profiles at 8–256 cores (either indirectly by predicting CRD_P, CRD_S, PRD_P, and PRD_S profiles, or directly), yielding 24 predicted profiles per benchmark. Using PIN, we also acquire the actual whole-program CRD and PRD profiles corresponding to these 24 predictions, and compare the measured and predicted profiles.

Profile comparisons use two metrics, *profile accuracy* and *performance accuracy*. Profile accuracy has been used in previous work [Jiang et al. 2010; Ding and Zhong 2003], and is defined as $1 - \frac{E_P}{2}$ where E_P is the sum of the normalized absolute differences between every pair of profile bins from a predicted and measured profile. (E_P can be at most 200%, so profile accuracy is between 0–100%). For example, the profile accuracy for a predicted CRD profile is:

$$Profile\ Accuracy = 1 - \frac{1}{2}\sum_{k=0}^{N-1}\frac{|CRD_{pred}[k] - CRD_{meas}[k]|}{total\ references} \quad (1)$$

where N is the total number of bins in the measured CRD profile, and $total\ references$ is the area under the measured CRD profile ($\sum_{k=0}^{N-1} CRD_{meas}[k]$). Similarly, the pro-

file accuracy for a predicted PRD profile can be computed using Equation 1 by simply replacing $CRD_{pred}[k]$ and $CRD_{meas}[k]$ with $PRD_{pred}[k]$ and $PRD_{meas}[k]$, respectively. Notice, profile accuracy does not consider $k = \infty$, so it does not reflect error in predicting coherence misses for PRD profiles.

Performance accuracy is our own metric computed on cache-miss count (CMC) profiles. A CMC profile presents the number of cache misses predicted by a CRD or PRD profile at every reuse distance value. For example, the CMC profile corresponding to a CRD profile is $CMC[i] = \sum_{j=i}^{N-1} CRD[j]$. Similarly, the CMC profile corresponding to a PRD profile is $CMC[i] = \sum_{j=i}^{N-1} PRD[j] + PRD[\infty]$. Performance accuracy is defined as $1 - E_C$, where E_C is the average error between pairs of profile bins from the first half of predicted and measured CMC profiles:

$$Performance\ Accuracy = 1 - \frac{2}{N} \sum_{k=0}^{\frac{N}{2}} \frac{|CMC_{pred}[k] - CMC_{meas}[k]|}{CMC_{meas}[k]} \tag{2}$$

Compared to profile accuracy, performance accuracy better reflects a profile's ability to predict cache performance since it uses CMC profiles. In addition, it also assesses accuracy across a wider range of cache capacities. Because profile accuracy is an absolute metric, it heavily weights accuracy at the first few profile bins where reference counts are very large but which occur below most capacities in a typical multicore processor. In contrast, performance accuracy is an average metric. It equally weights accuracy across the first half of CMC profiles which usually extend well beyond LLC capacities.

Finally, we only acquire and predict CRD and PRD profiles in the benchmarks' parallel phases. The third column of Table I reports the number of instructions in the parallel phases studied. For FFT, LU, and RADIX, these regions are the entire parallel phase; for the other benchmarks, these regions are 1 timestep of the parallel phase.

4.3. Accuracy Results

4.3.1. Profile Accuracy Metric. Figures 9 and 10 present our CRD and PRD profile prediction results. In Figure 9, 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 9 reports the average accuracy achieved over the 24 predictions per benchmark using the profile accuracy metric. The rightmost bars report the average across all benchmarks. In Figure 10, the exact same results are presented for the PRD_P and PRD_S profiles.

As Figures 9 and 10 show, CRD_P and PRD_P profiles are predicted with high accuracy. For all benchmarks except LU, CRD_P profile accuracy is between 90% and 99%. For LU, CRD_P profile accuracy is 70.4%. PRD_P prediction is even better. For all benchmarks, PRD_P profile accuracy is between 94% and 100%. On average, CRD_P and PRD_P achieve profile accuracies of 91.3% and 96.3%, respectively. Both CRD_P and PRD_P profiles exhibit coherent shift across core count scaling which reference groups can effectively predict. The results in Figures 9 and 10 demonstrate the pervasiveness of coherent shift in our benchmarks, and confirm the high accuracy of reference groups.

LU exhibits lower CRD_P accuracy. In LU, blocking is performed to improve cache locality, but for the S1 and S2 problems, the default blocking factor does not create enough parallelism to keep more than 32 cores busy. This introduces load imbalance and reduces the degree of memory interleaving, especially at large core counts. As a result, LU's CRD_P profile shifts much less than anticipated, so our techniques over-

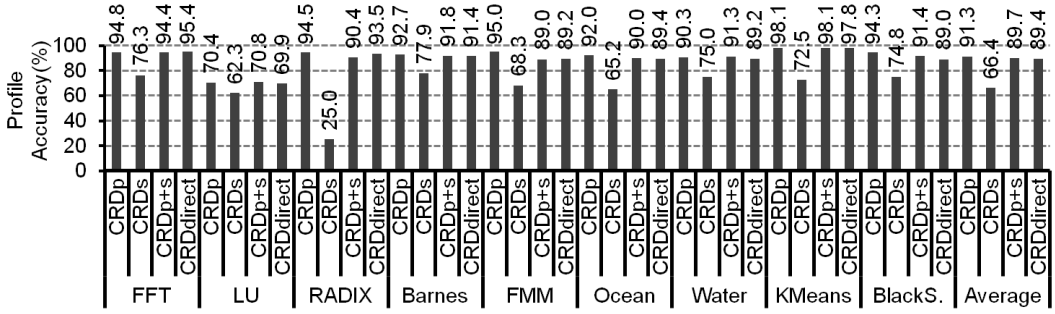

Fig. 9. Profile accuracy of predicted CRD_P and CRD_S profiles, and indirectly and directly predicted whole-program CRD profiles.

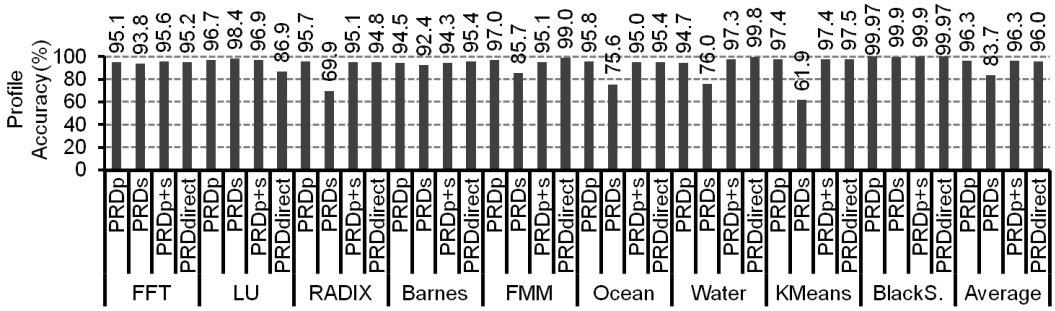

Fig. 10. Profile accuracy of predicted PRD_P and PRD_S profiles, and indirectly and directly predicted whole-program PRD profiles.

predict the shift and incur the error shown in Figure 9. Reduced interleaving also negatively impacts LU's PRD profile prediction, though this is not visible in Figure 10.

Compared to CRD_P profiles, CRD_S profiles are predicted with significantly lower accuracy. In Figure 9, CRD_S profile accuracy is between 25% and 78%. Across all benchmarks, the average CRD_S profile accuracy is only 66.4%. CRD_S profiles suffer poor spread prediction. While intercepts induce spreading in the range we expect (see Section 3.2), the actual distribution across this range is highly application dependent. Unfortunately, our simple uniform spread model does not capture all of the behaviors.

In contrast, predicted PRD_S profiles exhibit reasonable profile accuracy, though still not quite as good as PRD_P profiles. In Figure 10, PRD_S profile accuracy is between 61% and 100%. Across all benchmarks, the average PRD_S profile accuracy is 83.7%. Like intercepts, invalidations are also difficult to predict, which is why PRD_S profile accuracy is lower than PRD_P profile accuracy. However, whereas intercepts occur for both read and write sharing, invalidations only occur for write sharing. Because read sharing is usually more frequent than write sharing, poor intercept prediction impacts accuracy more than poor invalidation prediction. This is why PRD_S profile accuracy is higher than CRD_S profile accuracy.

Although shared profiles are predicted with lower accuracy than private profiles, the impact on overall prediction accuracy is minimal. In Figures 9 and 10, the bars labeled "CRD_{P+S}" and "PRD_{P+S}" report the average profile accuracy for predicted whole-program CRD and PRD profiles, respectively. In these experiments, the whole-program profiles are derived by combining the predicted private and shared profiles (CRD_P and CRD_S for CRD_{P+S}, and PRD_P and PRD_S for PRD_{P+S}). Except LU, CRD_{P+S} profile

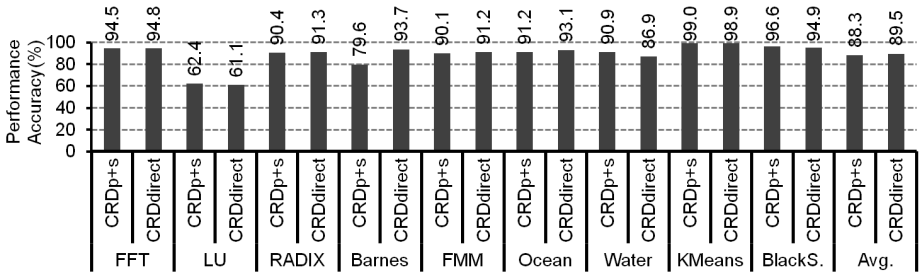

Fig. 11. Performance accuracy for predicted whole-program CRD profiles.

accuracy is between 89% and 99% (for LU, it is 70.8%). The average CRD_{P+S} profile accuracy for all benchmarks is 89.7%. PRD_{P+S} profile accuracy is between 94% and 100%. On average, the PRD_{P+S} profile accuracy for all benchmarks is 96.3%. These results confirm CRD_P and PRD_P dominate, so predicting them effectively is sufficient.

Since CRD_P and PRD_P dominate, one would expect predicting whole-program CRD and PRD profiles directly to achieve similar accuracy as predicting CRD_P and PRD_P profiles. The last set of bars in Figures 9 and 10, labeled "CRD_{direct}" and "PRD_{direct}," report the average CRD and PRD profile accuracy for direct whole-program prediction. Figure 9 shows CRD_{direct} is just slightly worse than CRD_{P+S}. On average, CRD_{direct} accuracy is 89.4%, compared to 89.7% for CRD_{P+S}. Figure 10 shows the same for PRD_{direct}. On average, PRD_{direct} accuracy is 96.0%, compared to 96.3% for PRD_{P+S}.

4.3.2. Performance Accuracy Metric. Figures 11 and 12 present our CRD and PRD profile prediction results using the performance accuracy metric. Because these results are similar to those in Figures 9 and 10, we only report the whole-program experiments (CRD_{P+S}, CRD_{direct}, PRD_{P+S}, and PRD_{direct}).

Our techniques achieve lower performance accuracy as compared to profile accuracy. For all benchmarks except LU, Figure 11 shows CRD_{P+S} performance accuracy is between 79% and 99% (for LU, it is 62.4%). The average CRD_{P+S} performance accuracy for all benchmarks is 88.3%, down from 89.7% under the profile accuracy metric. The accuracy degradation is worse for PRD profiles. For all benchmarks except LU, Figure 12 shows PRD_{P+S} performance accuracy is between 81% and 100% (for LU, it is 71.9%). The average PRD_{P+S} performance accuracy for all benchmarks is 86.6%, down from 96.3% under the profile accuracy metric.

As discussed earlier, profile accuracy heavily weights the first few (huge) profile bins which are typically predicted with higher accuracy than the smaller bins. Because performance accuracy equally weights all bins, it accumulates more error than profile accuracy. In addition, performance accuracy assesses CMC profiles which integrate bins up to ∞. So, any error in predicting the last bin at ∞ is propagated to *all* CMC profile bins. Unfortunately, the bin at ∞ for PRD profiles contains coherence misses which are hard to predict. This is why PRD profiles exhibit a larger reduction in performance accuracy than profile accuracy. The effect is most pronounced for LU. Recall from Section 4.3.1 that LU suffers load imbalance and reduced interleaving. Just as this makes dilation harder to predict for CRD profiles, the reduced interleaving also exascerbates invalidation prediction for PRD profiles by eliminating many invalidations. Hence, we overpredict LU's invalidations, causing the low performance accuracy in Figure 12.

While Figures 11 and 12 suggest we will achieve somewhat lower accuracy for performance prediction than Figures 9 and 10 would lead us to believe, the overall accuracy is still quite good. As mentioned above, the average CRD_{P+S} and PRD_{P+S} performance

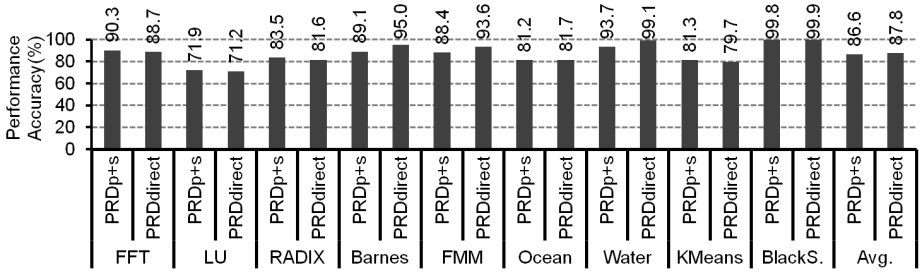

Fig. 12. Performance accuracy for predicted whole-program PRD profiles.

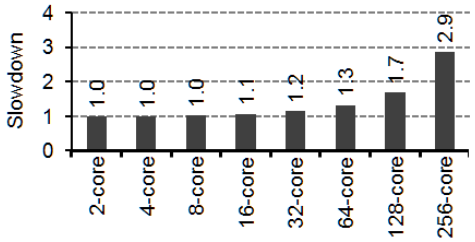

Fig. 13. Average normalized profiling time at different core counts.

accuracies are 88.3% and 86.6%, respectively. As Section 6 will show, this still permits our techniques to predict cache performance with high fidelity.

Finally, as with profile accuracy, whole-program CRD and PRD profiles predicted directly achieve similar performance accuracy as those derived from separately predicted private and shared profiles. Figures 11 and 12 show CRD_{direct} and PRD_{direct} achieve performance accuracies of 89.5% and 87.8%, respectively.

4.4. Speedup Advantage

The savings in time spent acquiring CRD and PRD/sPRD profiles afforded by our profile prediction techniques depends on two factors: the number of profiles predicted, and the core counts at which predictions occur. In our study, the former provides the greater source of speedup. In Section 4.3, we used profiles acquired at 2 and 4 cores to predict the profiles at 8, 16, 32, 64, 128, and 256 cores. So, prediction yielded 8 profiles from only 2 acquired profiles, a speedup of 4x.

Speedup also comes from profiling time variation at different core counts. By predicting the larger core-count profiles from the smaller core-count profiles, additional speedup is accrued since smaller machine sizes are less expensive to profile. This is because the OS overhead for interleaving our PIN threads' memory references increases with core count. Figure 13 reports the normalized profiling time at different core counts across all of the profiles studied in Section 4.3. Each bar in Figure 13 reflects the average for all profiles acquired at the same core count. As Figure 13 shows, profiling 2–8 cores takes the same amount of time, but then profiling time goes up. At 256 cores, profiles take almost 3x longer to acquire compared to 2 cores. Averaged across all the profiles, there is a 1.4x speedup per profile due to profiling time variation.

Taking into consideration both sources of speedup, our prediction techniques provide a net speedup of 5.6x for the study in Section 4.3. Notice, however, the achieved speedup highly depends on the size of the design space being studied. In particular, profiles at additional core counts could be predicted from the same 2- and 4-core pro-

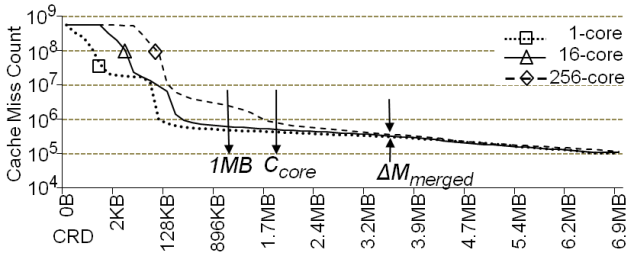

Fig. 14. Barnes' CMC profiles on 1, 16, and 256 cores for the S2 problem. C_{core} and ΔM_{merged} are labeled.

files without acquiring any new profiles. For example, if we were to add the missing core counts to the design space studied in Section 4.3 that are divisible by 4 (and assume linear interpolation of the profiling time variation results from Figure 13 for the missing core counts), the overall speedup would increase to 59x. In general, the larger the predicted design space, the greater the speedup that prediction will provide.

5. ARCHITECTURAL IMPLICATIONS

The insights presented in Section 3 (and confirmed in Section 4) have implications for cache design. This section discusses two that are particularly noteworthy: core count scaling containment in large shared caches and distance-based private-vs-shared cache performance gap. In the following, we discuss these architectural implications and define two parameters, C_{core} and C_{share}, that quantify the effects.

5.1. Core Count Scaling Containment

Section 3.2.1 shows the shifting of CRD profiles under core count scaling is limited: it slows down and eventually stops at large CRD values. This implies the locality degradation in shared caches due to machine scaling is also limited and does not affect larger cache capacities.

Figure 14 illustrates this phenomenon by plotting the CRD-based CMC profiles for the Barnes benchmark running the S2 problem. In Figure 14, cache-miss count is plotted as a function of CRD for 1, 16, and 256-core executions of Barnes. Because CRD profiles eventually stop shifting, their associated CMC profiles also merge at some point, as shown in Figure 14. We call the CRD value at which a multicore CMC profile merges with the single-core CMC profile, "C_{core}." C_{core} delineates the cache-miss impact of core count scaling. At CRD $< C_{core}$, cache misses increase significantly due to core count scaling whereas at CRD $> C_{core}$, cache-miss count is unaffected by core count scaling. In other words, *core count scaling of loop-based parallel programs degrades locality, but the impact is localized to cache capacities below C_{core}. Caches larger than C_{core} "contain" the scaling impact, and experience virtually no performance degradation*. For example, Figure 14 shows a shared cache of size C_{core} = 1.8MB can contain scaling up to 256 cores for the Barnes benchmark running the S2 problem.

Notice, the same does not apply to private caches. As discussed in Section 3.3.1, the shifting of sPRD profiles slows down with increasing sPRD, but it never completely stops. So, core count scaling always increases the number of cache misses in a private cache regardless of the cache's capacity. Although only shared caches can contain core count scaling effects, this containment phenomenon is still significant. Since modern CPUs usually employ shared caches as LLCs, a program's C_{core} value often determines whether core count scaling will increase the program's demand for off-chip bandwidth.

We compute C_{core} across all of the whole-program CRD profiles from Section 4.2. This is done as follows. For each benchmark and problem size, we derive the CMC profiles for 1–256 cores. For a given CMC profile representing P cores, we define ΔM to be

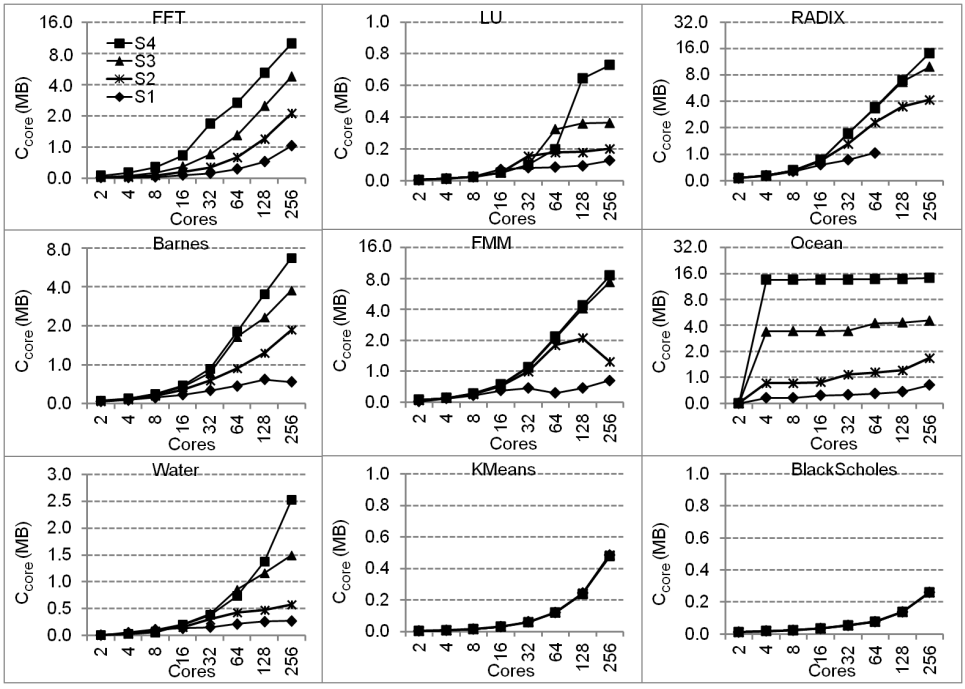

Fig. 15. Measured C_{core} as a function of core count for different benchmarks and problem sizes.

the ratio of cache-miss counts between the P- and 1-core CMC profiles at a particular CRD value. We first compute ΔM at CRD = $\frac{C_{max}}{2}$, where C_{max} is the CMC profile's maximum CRD value. ($\frac{C_{max}}{2}$ always occurs well beyond C_{core} where the CMC profiles have merged). We call this ΔM_{merged}. Then, we identify the CRD closest to $\frac{C_{max}}{2}$ where ΔM =1.5 $\times \Delta M_{merged}$, *i.e.* the tail-end of shifting where very large ΔM transition to ΔM_{merged}. We define this CRD value as C_{core} for the CMC profile representing P cores. We repeat this analysis to derive C_{core} for P = 2–256 cores. (C_{core} is defined for each core count, and represents the containment capacity for scaling up to that core count). This is done first for a particular problem size, and then across different problem sizes.

Figure 15 reports our C_{core} results. In Figure 15, different graphs plot the measured C_{core} values from separate benchmarks. Within each graph, different curves plot the C_{core} values for a different problem size (S1–S4), with each curve plotting C_{core} as a function of core count.[3] As Figure 15 shows, C_{core} usually increases with core count. (In most cases, the rate of increase is linear). However, despite its growth with core count, C_{core} remains relatively small. Across all the configurations in Figure 15, C_{core} varies between only 1.4KB to 13.2MB. These results show the locality degradation due to core count scaling for our benchmarks and problem sizes is confined to smaller shared caches, $<$ 16MB. For LLCs larger than 16MB, core count scaling up to 256 cores will not increase off-chip traffic in our benchmarks running the S1–S4 inputs.

Although our C_{core} results are relatively small, Figure 15 also shows C_{core} usually increases with problem size (with the exception of KMeans and BlackScholes). This is

[3]A few measurements are missing for the RADIX benchmark at S1 because of large per-core data structures that cause C_{max} to increase significantly with core count scaling. This makes a common $\frac{C_{max}}{2}$ across different core counts impossible to define, thus preventing C_{core} calculation.

Table II. C_{max}, ΔM_a, and ΔM_m for our benchmarks.

Benchmark	S1	S2	S3	S4
		C_{max}		
FFT	4.3MB	14.3MB	52.3MB	200.3MB
LU	785.8KB	2.3MB	8.3MB	32.4MB
RADIX	18.3MB	30.3MB	78.3MB	270.3MB
Barnes	2.1MB	6.9MB	26.5MB	105.3MB
FMM	3.9MB	12.2MB	42.7MB	163.0MB
Ocean	6.4MB	18.7MB	63.9MB	237.2MB
Water	1.2MB	3.4MB	11.5MB	45.5MB
KMeans	5.3MB	19.5MB	76.5MB	304.5MB
BlackScholes	1.7MB	6.2MB	24.2MB	96.2MB
Average	4.9MB	12.6MB	42.7MB	161.6MB
		ΔM_a / ΔM_m		
FFT	3.0 / 3.2	3.4 / 3.7	3.3 / 4.0	3.5 / 4.4
LU	- / -	- / -	- / -	- / -
RADIX	- / -	5.1 / 7.2	3.0 / 6.0	2.6 / 5.6
Barnes	- / -	3.3 / 5.2	3.6 / 7.3	3.7 / 8.8
FMM	- / -	2.3 / 2.6	2.0 / 2.8	2.1 / 3.0
Ocean	- / -	2.8 / 3.6	1.5 / 1.9	1.2 / 1.7
Water	- / -	- / -	1.7 / 2.0	1.8 / 2.1
KMeans	- / -	- / -	- / -	- / -
BlackScholes	- / -	- / -	- / -	- / -
Average	3.0 / 3.2	3.4 / 4.5	2.5 / 4.0	2.5 / 4.3

because problem scaling also shifts CRD profiles [Zhong et al. 2003]. When core count and problem size scale together, the shifting region associated with core count scaling will itself shift to larger CRD values due to problem scaling.

To estimate the impact of continued problem scaling, the top portion of Table II reports C_{max} for each benchmark and problem size. As Table II shows, C_{max} increases by roughly 4x with each problem size increment. Table I shows each problem size increment increases data structures by 4x as well, so C_{max} grows linearly with problem size. In contrast, Figure 15 shows C_{core} increases at a sub-linear rate with problem size. While the rate of increase varies across different benchmarks and core counts, in most cases it can be approximated as $\sqrt{C_{max}}$. Assuming the same rate of increase for larger problems, we see that another 64x increase in problem size would cause C_{core} to grow to 64–128MB for many benchmarks. For these larger problems, core count scaling would likely impact the performance of large LLCs.

In addition to quantifying C_{core}, we also assess the cache-miss increases caused by core count scaling in the shifting region below C_{core}. For each CMC profile, we compute ΔM at every CRD value between 1MB and C_{core} (*i.e.*, in the region labeled in Figure 15), recording the average and maximum values. These are the average and maximum cache-miss count increases below C_{core}, ΔM_a and ΔM_m, respectively. The 1MB lower boundary focuses our analysis on LLC-sized caches.

The bottom portion of Table II reports ΔM_a and ΔM_m. Results are only presented for cases where $C_{core} >$ 1MB. As Table II 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 LLC sizes below C_{core}.

5.2. Private-vs-Shared Performance Gap

Section 3.3.1 shows sPRD profiles have longer tails than CRD profiles, resulting in greater miss counts for private caches compared to shared caches. To illustrate, Figure 16 plots the CRD- and sPRD-based CMC profiles for the FFT benchmark running

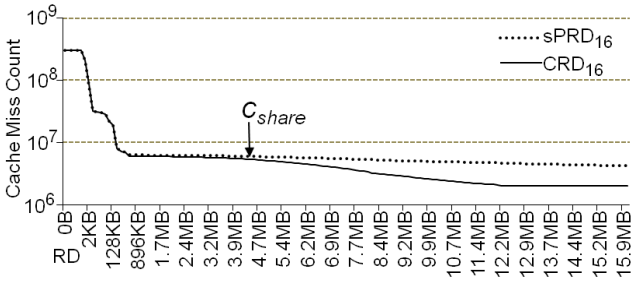

Fig. 16. CRD- and sPRD-based CMC profiles for FFT running the S2 problem on 16 cores.

the S2 problem on 16 cores. Like Figure 14, Figure 16 also plots cache-miss count as a function of CRD (and equivalently, sPRD).

Figure 16 shows the sPRD-based CMC profile is indeed *above or equal to* the CRD-based CMC profile at all CRD/sPRD values, thus confirming the observation from Section 3.3.1. This performance gap is due to the fact that CRD profiles contract more than sPRD profiles. As discussed in Section 3.3.1, overlap-induced contraction of CRD is driven by both shared reads and writes whereas only shared writes contribute to demotion absorption-induced contraction of sPRD. The higher contraction rate in CRD compared to sPRD leads to better locality and lower cache-miss counts for shared caches compared to private caches, as shown in Figure 16.

More importantly, Figure 16 also shows this private-vs-shared cache performance gap varies in a consistent fashion with reuse distance: there is no gap at small CRD/sPRD, but then the two CMC profiles diverge and a gap emerges that grows with increasing CRD/sPRD. Moreover, because this performance gap is due to data sharing, its variation across different CRD/sPRD values is also an indication of how data sharing changes as a function of reuse distance. Hence, another interpretation of Figure 16 is that data sharing is absent at small CRD/sPRD, then "turns-on" at some critical CRD/sPRD value, and increases with reuse distance. We call the turn-on point where data sharing becomes noticeable, "C_{share}." For example, Figure 16 shows our FFT benchmark running the S2 problem on 16 cores exhibits a C_{share} of about 4.6MB.

C_{share} delinates private-vs-shared cache performance. Below C_{share}, both private and shared caches incur the same number of cache missses. Because private caches typically exhibit lower access latency than shared caches [Nayfeh and Olukotun 1994; Huh et al. 2005], private caches outperform shared caches in this region. Above C_{share}, shared caches incur fewer cache misses than private caches. Shared caches can potentially outperform private caches in this region, but it depends on the magnitude of the cache-miss reduction compared to the increase in the shared cache's access latency.

We compute C_{share} across all of the whole-program CRD and sPRD profiles from Section 4.2 as follows. For each benchmark, problem size, and core count, we derive the CRD- and sPRD-based CMC profiles. At a particular CRD/sPRD value, we define ΔM^{-1} to be the ratio of cache-miss counts between the CRD- and sPRD-based CMC profiles. We compute ΔM^{-1} at all CRD/sPRD, and identify the smallest CRD/sPRD value where ΔM^{-1} = 0.9 (*i.e.*, when the CRD- and sPRD-based cache-miss counts are within 10%). We define this CRD/sPRD value as C_{share}.

Figure 17 reports our C_{share} results. In Figure 17, we plot C_{share} for all 9 benchmarks, all 4 problem sizes, and all core counts between 2–256 using the same format as Figure 15. Figure 17 shows C_{share} is highly sensitive to core count scaling. In general, C_{share} starts out larger and becomes smaller as core count increases. This makes sense since sharing usually becomes more frequent (which would cause C_{share} to decrease)

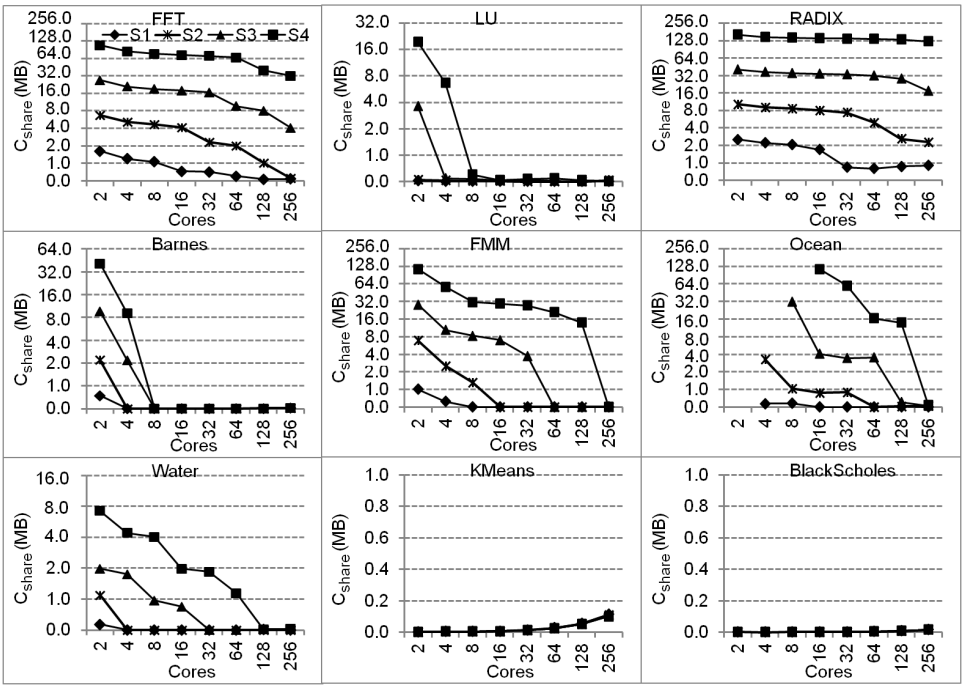

Fig. 17. Measured C_{share} as a function of core count for different benchmarks and problem sizes.

as computation is distributed across a larger number of cores. But the capacities across which C_{share} varies is highly application and data input dependent, so the implication for cache design depends on the benchmark and problem size.

For example, in RADIX running the S4 problem, C_{share} is always around 128MB regardless of core count. In this case, private caches will always provide the best performance at any caching level. (Today, on-chip caches are all below 128MB). For Barnes running the S4 program, and for FMM and Ocean running the S3 and S4 problems, C_{share} starts out at or above 32MB but becomes very small at large core counts. In these cases, the preference for private versus shared caches will depend on the machine size. And for KMeans and Blackscholes, C_{share} is always below 116KB. In these cases, shared caches will likely provide the best performance. These results show no single cache hierarchy can universally achieve the best performance. Instead, our results show architectures that can adapt cache sharing at different levels of the cache hierarchy will likely outperform fixed cache organizations.

6. PERFORMANCE PREDICTION

We now demonstrate our techniques' ability to accelerate multicore design evaluation. We employ our core count prediction techniques from Section 4 to assess cache performance–*i.e.*, private cache MPKI (misses per 1000 instructions) and shared LLC MPKI. We also incorporate previous techniques for predicting problem scaling to further increase the acceleration advantage. The study proceeds in five parts. First, Section 6.1 describes architectural assumptions, and Section 6.2 discusses how we predict performance. Next, Section 6.3 studies the CRD and PRD profiles' dependence on cache capacity. Then, Section 6.4 presents the main prediction accuracy results. Finally, Section 6.5 addresses end-to-end performance and advanced caching techniques.

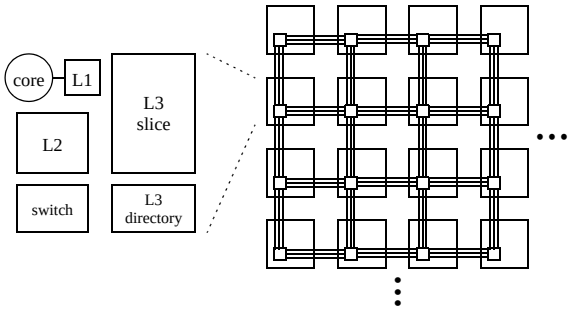

Fig. 18. Tiled CMP assumed in our performance prediction study. Each tile contains a core, a private L1 cache, a private L2 cache, an L3 cache and directory "slice," and an on-chip network switch.

6.1. Architecture Assumptions

Our performance study assumes *tiled CMPs* [Zhang and Asanovic 2005]. A tiled CMP consists of several identical replicated tiles, each with compute, memory, and communication resources. Because tiled CMPs are considered to be scalable architectures [Zhang and Asanovic 2005; Hardavellas et al. 2009], they permit us to study a large parallel cache hierarchy design space on a single multicore platform.

Figure 18 illustrates the tiled CMP assumed in our work. Each tile consists of a core, a private L1 cache, a private L2 cache, an L3 cache and directory "slice," and a switch for a 2-D on-chip point-to-point mesh network. The L3 slices across all tiles are managed as a single logically shared LLC. We assume the LLC does not replicate or migrate cache blocks between slices. Each cache block is always placed in the same L3 slice, known as the cache block's "home." We assume cache block homes are page-interleaved (with 8KB page size) across L3 slices according to their physical address. The L1/L2 private caches and each L3 slice employ an LRU replacement policy.

To mitigate remote-L3 slice latency, we permit replication in the private L1 and L2 caches. We employ a directory-based MESI cache coherence protocol to maintain L1/L2 coherence. The protocol uses a distributed full-map directory where each directory entry is collocated with its cache block on the home tile. In addition, we assume the memory sub-system supports multiple DRAM channels, each connected to a memory controller on a special "memory tile." We use 4 memory tiles evenly spaced on the north and south edges of the chip.

Our study uses the M5 simulator to measure performance. We modified M5 to model the tiled CMP described above. Our simulator's core model is very simple: each core executes 1 instruction per cycle (in the absence of memory stalls) in program order. However, the memory system model is very detailed, accurately modeling L1 and L2 access, hops through the network, L3 slice access, communication with the memory controller, and DRAM access. We also model queuing at the on-chip network switches and memory controllers. Table III lists the parameters used in our simulations. As Table III shows, we simulate processors with 2–256 cores, 8KB private L1 caches, 16–256KB private L2 caches, and 4–128MB of total shared L3 cache (LLC).

In addition to performance, our M5 simulator also measures whole-program CRD and PRD profiles. We track CRD and PRD for the simulated interleaved memory reference stream from all cores using the same approach as our PIN tool. In contrast to the PIN-based profiles, our M5 profiles account for timing effects when forming the interleaved memory reference streams. Similar to the PIN tool, CRD and PRD are computed at a 64-byte granularity, the block size for all caches in our M5 simulations. Although our M5 and PIN profiles are acquired on different architectures (Alpha vs. X86) and differ in timing, we find they are very similar. We compared the M5 and

Table III. Simulator parameters used in the experiments.

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

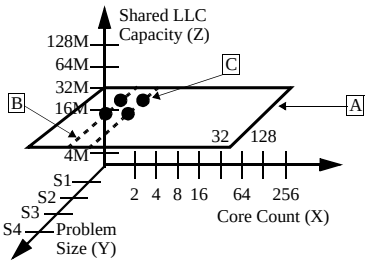

Fig. 19. Architecture-problem space with core count, shared LLC capacity, and problem scaling.

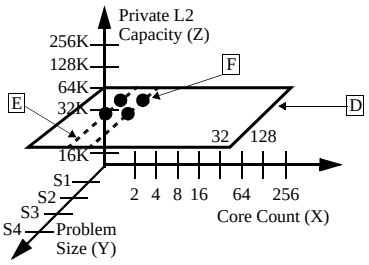

Fig. 20. Architecture-problem space with core count, private L2 capacity, and problem scaling.

PIN CRD and PRD profiles at every core count and problem size (for M5, we picked profiles using 64KB L2 and 32MB L3 caches). Under the performance accuracy metric, the two sets of profiles are 90.1% similar on average across all the benchmarks.

To drive our simulations, we use the same benchmarks and problem sizes from Table I, simulating the same parallel regions described in Section 4.2. (The last column of Table I reports the number of instructions in the parallel regions studied for the M5 experiments). For benchmarks running 1 timestep, we warmup caches in a separate timestep before recording performance and CRD/PRD profiles. For benchmarks running the entire parallel phase, we do not perform any explicit cache warmup.

6.2. Prediction Approach

Our study considers large design spaces formed by different architectural configurations and problem sizes, which we call "architecture-problem spaces." We define two such spaces, one that scales the shared LLC and another that scales the private L2 caches. We use these two spaces to study shared LLC MPKI prediction using CRD profiles and private L2 MPKI prediction using PRD profiles, respectively.

Figure 19 illustrates the architecture-problem space for shared LLC MPKI prediction. This space is formed by the cross product of all core counts and shared LLC capacities from Table III and all problem sizes from Table I. (In these configurations, each private L2 cache is always 64KB). There are 192 configurations per benchmark in this case, and 1,728 configurations across our 9 benchmarks. Figure 20 illustrates the architecture-problem space for private L2 MPKI prediction. This space is formed by the cross product of all core counts and private L2 cache capacities from Table III and all problem sizes from Table I. (In these configurations, the shared LLC is 32MB for the 2–16 core machines and 128MB for the 32–256 core machines). There are 160 configurations per benchmark in this case, and 1,440 configurations across our 9 benchmarks.

We simulate all 3,168 (1,728 + 1,440) configurations using our M5 simulator. For each simulation associated with the shared LLC architecture-problem space, we obtain the shared LLC MPKI and CRD profile. And for each simulation associated with

the private L2 architecture-problem space, we obtain the private L2 MPKI and PRD profile. (Each profile contains between a few thousand and a few million data points–essentially $C_{max}/64$, where C_{max} is given in Table II). Similar to our profile prediction study from Section 4.3, we use a sub-set of the measured CRD profiles to predict shared LLC MPKI at all configurations and a sub-set of the measured PRD profiles to predict private L2 MPKI at all configurations, and then compare the predicted MPKI values against the simulated MPKI to assess accuracy.

The labels in Figure 19 illustrate how we obtain the CRD profiles for shared LLC MPKI prediction (the same process is followed for private L2 MPKI prediction). For the shared LLCs, we obtain the 32 CRD profiles per benchmark for all core counts / problem sizes with a 32MB LLC–*i.e.*, the plane labeled "A" in Figure 19. We use 3 profile prediction strategies for doing this: "No-Pred," "C-Pred," and "CP-Pred." No-Pred does not perform profile prediction, and simply uses the complete set of measured CRD profiles in the "A" plane of Figure 19 to predict performance. No-Pred requires measuring 32 CRD profiles to make the 192 shared LLC MPKI predictions per benchmark.

C-Pred performs core count prediction. At each problem size within the "A" plane, C-Pred predicts the 8- to 256-core CRD profiles from the 2- and 4-core profiles–*i.e.*, the two dotted lines labeled "B" in Figure 19. We use our techniques for directly predicting whole-program CRD profiles from Sections 4.1 and 4.2. C-Pred requires measuring 8 CRD profiles to make the 192 shared LLC MPKI predictions per benchmark.

Finally, CP-Pred combines core count prediction with problem size prediction. Just like C-Pred, CP-Pred predicts across core count to acquire all CRD profiles in the "A" plane. However, within the 2- and 4-core configurations, CP-Pred predicts the S3 and S4 CRD profiles from the S1 and S2 profiles–*i.e.*, the four dots labeled "C" in Figure 19. To predict across problem size, CP-Pred also uses reference groups from Section 4.1. But instead of diffing and shifting profiles across core count, it diffs and shifts across problem size (*i.e.*, the original use of reference groups). CP-Pred requires measuring 4 CRD profiles to make the 192 shared LLC MPKI predictions per benchmark.

An identical process is used to acquire/predict PRD profiles, as shown in Figure 20. For the private L2 caches, we obtain the 32 PRD profiles per benchmark for all core counts / problem sizes with a 64KB L2 cache–*i.e.*, the plane labeled "D" in Figure 20. The No-Pred approach avoids prediction and uses the measured PRD profiles in the "D" plane of Figure 20. The C-Pred approach predicts 8- to 256-core PRD profiles from the 2- and 4-core profiles–*i.e.*, the two dotted lines labeled "E" in Figure 20–using our techniques for directly predicting whole-program PRD profiles from Sections 4.1 and 4.2. And the CP-Pred approach first predicts the S3 and S4 PRD profiles from the S1 and S2 profiles within the 2- and 4-core configurations–*i.e.*, the four dots labeled "F" in Figure 20–before predicting the 8- to 256-core PRD profiles at every problem size. Similar to CRD profiles, No-Pred, C-Pred, and CP-Pred require measuring 32, 8, and 4 PRD profiles, respectively, to make the 160 private L2 MPKI predictions per benchmark.

Once all 32 CRD and PRD profiles within the "A" and "D" planes have been acquired, then MPKI can be predicted. For the shared LLCs, we derive the corresponding CMC profile from each CRD profile, and extract cache-miss counts at 4–128MB on the CMC profile. Similarly for the private L2 caches, we derive the corresponding CMC profile from each PRD profile, and extract cache-miss counts at 16–256K on the CMC profile. This predicts the number of shared LLC and private L2 cache capacity misses, respectively. We use Qasem and Kennedy's model [Qasem and Kennedy 2005] to predict conflict misses. This model takes either the CRD or PRD profile as input (for the shared LLC or private L2 cache, respectively), and uses a binomial distribution to predict the number of conflict misses for a given capacity and associativity.

Finally, we divide the sum of predicted conflict and capacity misses by instruction count (IC) to derive the corresponding cache's MPKI. For No-Pred and C-Pred, we use

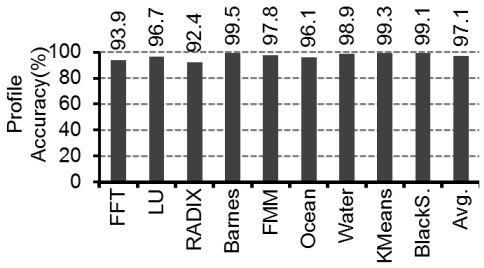

Fig. 21. CRD stability across cache capacity under the profile accuracy metric.

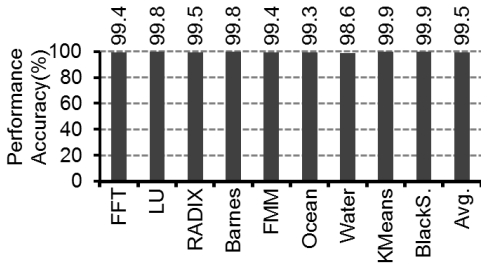

Fig. 22. CRD stability across cache capacity under the performance accuracy metric.

the measured IC at the same configuration that contributed the CRD or PRD profile for MPKI prediction–*i.e.*, we assume IC doesn't change across either cache capacity or core count scaling. We make the same assumption for CP-Pred, except we also predict IC across problem scaling by assuming IC changes linearly with problem size at the same rate observed from S1 to S2. (This ensures we only use measured ICs from configurations where we also measured the CRD or PRD profile).

6.3. Profile Dependence on Cache Capacity

Before presenting our prediction results, we first revisit the issue of architecture dependence. Sections 3 and 4 have already addressed the dependence of CRD and PRD profiles on core count. As discussed in Section 1, another issue is dependence on cache capacity. As cache size changes, the relative speed of individual threads can also change which will alter memory interleaving and CRD/PRD profiles. A basic premise of this article (also held by other researchers [Jiang et al. 2010]) is that relative speed amongst symmetric threads is stable with respect to cache size; hence, CRD and PRD profiles for symmetric threads should be valid across cache capacity scaling.

We study the stability of profiles across cache capacity over all of our M5 simulations. Specifically, we compare the CRD profiles acquired at a particular machine and problem size on LLCs of size 4, 8, 16, 64, and 128MB against the CRD profile acquired at the same machine and problem size on an LLC of size 32MB (*i.e.*, comparisons are along the Z-axis of Figure 19). This yields 5 comparisons per machine and problem size across 32 different machine and problem sizes, or 160 comparisons per benchmark. In addition, we compare the PRD profiles acquired at a particular machine and problem size on private L2 caches of size 16, 32, 128, and 256KB against the PRD profile acquired at the same machine and problem size on private L2 caches of size 64KB (*i.e.*, comparisons are along the Z-axis of Figure 20). This yields 4 comparisons per machine and problem size across 32 different machine and problem sizes, or 128 comparisons per benchmark. All comparisons use the profile simlarity metrics from Section 4.2.

Figures 21 and 22 report our cache capacity dependence results for CRD profiles. Each bar in Figure 21 presents the average similarity across all 160 CRD profile comparisons for a given benchmark under the profile accuracy metric, with the rightmost bar presenting the average across all 9 benchmarks. Figure 22 does the same using the performance accuracy metric. Similarly, Figures 23 and 24 report our cache capacity dependence results for PRD profiles in the same format as Figures 21 and 22.

In Figures 22–24, the average similarity for every benchmark is always within 99%. For Figure 21 (CRD profiles under the profile accuracy metric), the average similarity for each benchmark is always within 92%, and usually within 96%. These results confirm symmetric threads' CRD and PRD profiles are highly insensitive to cache capacity scaling. They also demonstrate CRD and PRD profiles are insensitive to slight changes in memory reference interleaving.

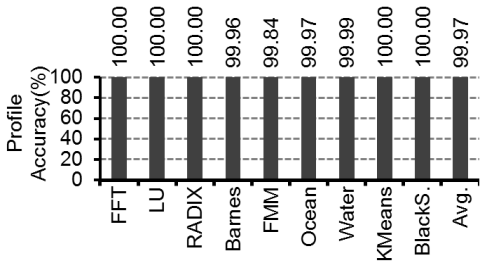

Fig. 23. PRD stability across cache capacity under the profile accuracy metric.

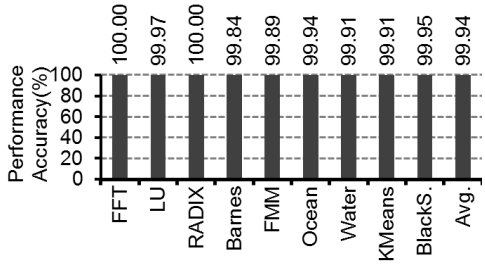

Fig. 24. PRD stability across cache capacity under the performance accuracy metric.

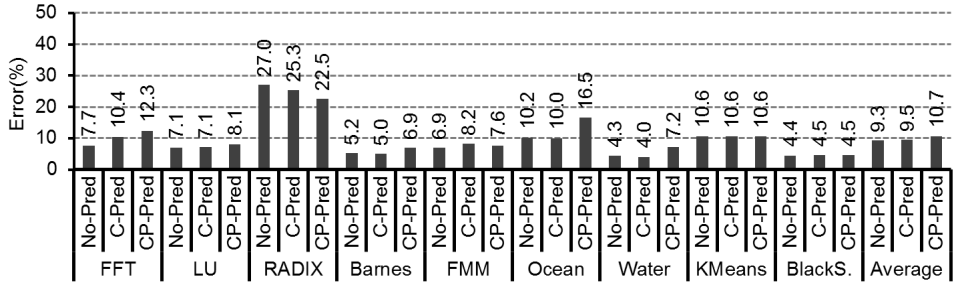

Fig. 25. Percent prediction error for shared LLC MPKI.

6.4. MPKI Prediction Accuracy

Figures 25 and 26 report our MPKI prediction results for the shared LLC and private L2 caches, respectively. These results are based on percent error:

$$PercentError = \frac{|predicted\_MPKI - measured\_MPKI|}{measured\_MPKI} \quad (3)$$

Each bar in Figures 25 and 26 reports the average offsetted percent error (explained below) across all predictions for a particular prediction strategy and benchmark (*i.e.*, for 192 configurations from the shared LLC architecture-problem space for Figure 25, and for 160 configurations from the private L2 architecture-problem space for Figure 26). The rightmost bars in both figures report averages across all benchmarks.

One problem is Equation 3 blows up when the measured MPKI approaches zero. To prevent this, we add a small offset to both predicted and measured MPKI values before computing percent error, in essence masking errors associated with small MPKI that are negligible from a performance standpoint. We use offsets of 0.05 and 1.0 for shared LLC MPKI and private L2 MPKI, respectively, which correspond to a 1% performance degradation assuming CPI = 1.0 and the memory latencies from Table III. (The actual performance impact of these offsets is even less given that CPI $>$ 1.0 in reality).

Figure 25 shows No-Pred is able to predict shared LLC MPKI to within 11% of simulation on average for 8 out of 9 benchmarks, and to within 27% for RADIX. Across all benchmarks, the shared LLC MPKI prediction error is 9.3%. No-Pred is similarly effective for predicting private L2 MPKI. Figure 26 shows No-Pred predicts private L2 MPKI to within 16% for every benchmark. Averaged across all benchmarks, the private L2 MPKI prediction error is 8.5%.

These results reflect baseline prediction errors (*i.e.*, without profile prediction), and include 2 main error sources. First, Qasem and Kennedy's model, which we use to

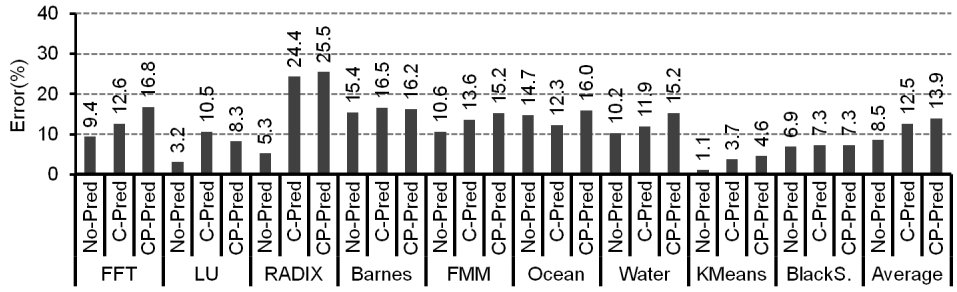

Fig. 26. Percent prediction error for private L2 MPKI.

predict conflict misses (see Section 6.2), incurs some error. In particular, certain configurations exhibit pathologic conflicts that the model cannot predict. This is the dominant source of error in the No-Pred results. And second, our offsetted percent error metric is not always stable. In some cases, shared LLC MPKI and private L2 MPKI are near their offsets. Thus, the percent error metric is still highly sensitive to minute prediction errors. This is the reason for RADIX's poor shared LLC MPKI predictions in Figure 25. Overall, Figures 25 and 26 show No-Pred error is very low, so CRD and PRD profiles are capable of accurately predicting MPKI for loop-based parallel programs.

Figures 25 and 26 also show our profile prediction techniques are very effective. In Figure 25, both C-Pred and CP-Pred are able to predict shared LLC MPKI to within 17% of simulation for 8 out of 9 benchmarks, and to within 26% for RADIX. On average, prediction error is 9.5% and 10.7% for C-Pred and CP-Pred, respectively. Results are slightly worse for L2 MPKI prediction, though still quite good. Figure 26 shows both C-Pred and CP-Pred are able to predict private L2 MPKI to within 17% of simulation for 8 out of 9 benchmarks, and to within 26% for RADIX. On average, prediction error is 12.5% and 13.9% for C-Pred and CP-Pred, respectively.

These results confirm profile prediction has high accuracy, as was shown in Section 4.3. Another reason C-Pred and CP-Pred are effective is because errors often cancel. While the cache conflict model usually under-predicts conflict misses, CRD and PRD profile prediction (across both core count and problem scaling dimensions) usually over-predict capacity misses. This also explains why C-Pred and CP-Pred can sometimes achieve lower error than No-Pred in Figures 25 and 26.

6.4.1. Shifting Region Analysis for Shared LLCs. When measured MPKI is close to zero, our offsetted percent error metric will report zero error. This happens frequently in the shared LLC MPKI prediction results due to the large LLC capacities we assume, which can unfairly bias the results towards lower error rates. (Notice, this problem does not occur for our private L2 MPKI prediction results because the private L2 capacity is relatively small, so MPKI is always fairly large).

Figure 27 reports shared LLC MPKI prediction error for the S4 problem and 4–16MB LLCs only (it still includes 2–256 cores). Most of these configurations have shared LLC size $< C_{core}$. Hence, Figure 27 studies our prediction error in the region of CRD profile shift where cache misses occur frequently, so MPKI is always significant. (LU, KMeans, and BlackScholes are omitted because their C_{core} are always below our smallest LLCs). As Figure 27 shows, prediction error in the shifting region is comparable to the entire architecture-problem space. Error gets worse for FFT and Barnes. But it improves in RADIX, especially for No-Pred and C-Pred because Figure 27 does not include RADIX's poorly predicted cases. Overall, C-Pred has similar error as No-Pred,

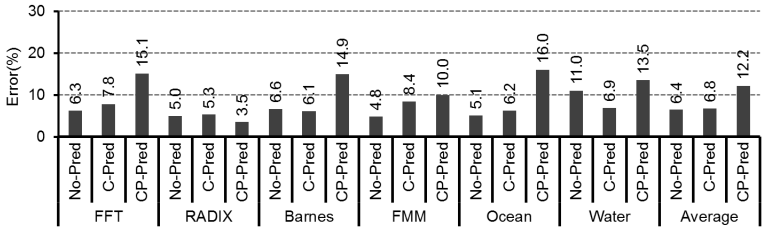

Fig. 27. Shared LLC MPKI prediction error for S4 and 4–16MB shared LLCs.

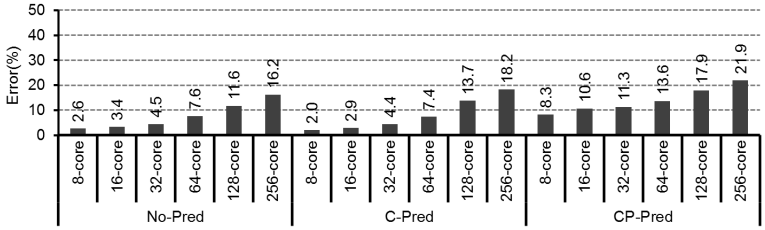

Fig. 28. Prediction error for S4 and 4–16MB shared LLCs by core count.

around 6.5%, showing core count prediction is effective in the shifting region. CP-Pred is worse–12.2% error. This increased error is due to over-prediction of shifting.

Finally, Figure 28 reports results for the same problem and shared LLC sizes in Figure 27 broken down by strategy and core count. Like Figure 27, Figure 28 shows C-Pred is similar to No-Pred, while CP-Pred is worse. More importantly, Figure 28 also shows prediction error increases with core count, reaching 17% for No-Pred, 19% for C-Pred, and 22% for CP-Pred. This illustrates the cache conflict model errors mentioned earlier which get worse with core count. Even so, Figure 28 shows C-Pred's error is still reasonable when predicting large core counts.

Overall, we find our prediction techniques for core count scaling accelerate cache performance analysis without sacrificing much on accuracy. When combined with problem scaling prediction, analysis effort is further reduced, but prediction error increases. Still, prediction errors are sufficiently low to provide meaningful results to designers.

6.5. Discussion

This section focuses on cache performance; however, we believe our techniques can also predict end-to-end application performance. In particular, prior work has developed models for predicting execution time from cache-miss counts [Eyerman et al. 2009; Karkhanis and Smith 2004] even for complex out-of-order cores. Combining our techniques with such previous techniques could yield end-to-end performance prediction.

In addition, this section assumes all caches employ an LRU management policy. Although LRU caches are common, researchers have also investigated caches with more advanced management–for example, dynamic insertion policies [Jaleel et al. 2010]. Unfortunately, our techniques are unlikely to predict the performance of such advanced caches accurately because their hardware recency lists do not match our CRD and PRD stacks. An intriguing possibility is to alter our stacks such that they mimic other caching policies in the hopes of enabling prediction for advanced caches. It would also be interesting to see whether our core count and problem scaling insights are valid for the profiles acquired from the altered stacks. These are all directions for future research that we hope to pursue.

7. PREDICTION VERSUS SAMPLING

CRD and PRD profile prediction accelerate cache design evaluation primarily by reducing the number of profiles that need to be acquired. Alternatively, it is possible to reduce the time to acquire each profile. This is the benefit provided by *profile sampling* [Schuff et al. 2010; Zhong and Chang 2008]. In this section, we compare prediction and sampling, and contrast how they trade off profiling speed for accuracy. We also evaluate their combination. Section 7.1 begins by discussing profile sampling. Then, Section 7.2 presents comparative and combined results.

7.1. Profile Sampling

Profile sampling alternates between profiling and fast-forwarding intervals. In profiling intervals, the profiler selects memory references from threads' memory reference streams, and measures their CRD/PRD as usual. In fast-forwarding intervals, the profiler maintains minimal information, and runs much faster. The sampling rate, $R_{sampling}$, is defined as $\frac{T_p}{T_p+T_f}$ where T_p and T_f are the number of memory references in the profiling and fast-forwarding intervals, respectively. A larger sampling rate tracks more memory references and yields higher accuracy whereas a smaller sampling rate incurs shorter profiling times. Thus, tuning $R_{sampling}$ trades off speed versus accuracy.

Unfortunately, turning off profiling after every T_P memory references will never sample references with reuse windows $> T_p$. To address this problem, profiling intervals can be extended beyond T_P until references that have appeared in the interval have been reused. To prevent profiling intervals from becoming arbitrarily long, references with extremely large RD values can be pruned from sampling [Schuff et al. 2010]. Pruning checks the oldest memory reference in the interval that has not seen a reuse. If this reference's LRU stack depth is sufficiently large, it is pruned from sampling. We implement pruning, but to provide more precise control, we define a pruning rate, $R_{pruning} = \frac{reused\ unique\ references}{total\ unique\ references}$. Our technique terminates a profiling interval after at least T_P memory references and the fraction of reused references in the interval reaches $R_{pruning}$.

We added support for sampling to our PIN-based profiler. During profiling intervals, our PIN tool interleaves memory references uniformly across threads and performs LRU stack updates to acquire CRD and PRD profiles, as described in Section 3.1. During fast-forwarding intervals, our PIN tool executes threads as quickly as possible (*i.e.*, without fine-grain memory interleaving nor LRU stack updates). We also vary the sampling and pruning rates. For some experiments, we use $R_{sampling}$ = 0.1 (T_p = 100K and T_f = 900K) and $R_{pruning}$ = 0.99 to perform sampling frequently. We also try $R_{sampling}$ = 0.01 (T_p = 100K and T_f = 9.9M) and $R_{pruning}$ = 0.9 to perform sampling less frequently. We refer to these as "high-sampling" and "low-sampling" rates, respectively.

Using sampling, we acquire the same profiles studied for core count prediction in Section 4. In particular, for each benchmark, we acquire the sampled whole-program CRD and PRD profiles at 2–256 cores, and at the S1–S4 problem sizes. This is done using both high- and low-sampling rates. Then, we compare the sampling experiments against the full measurement runs to assess accuracy and speedup, just like in Sections 4.3 and 4.4. When assessing accuracy, we do not consider the sampled profiles at 2 and 4 cores to mirror our prediction results.

7.2. Sampling Results

We begin by comparing our core count prediction techniques against sampling. After this baseline comparison, we then evaluate combining prediction and sampling.

Recall from Section 4.4 that our prediction techniques achieve a 5.6x speedup over full profile measurement when predicting across core count for the design space in

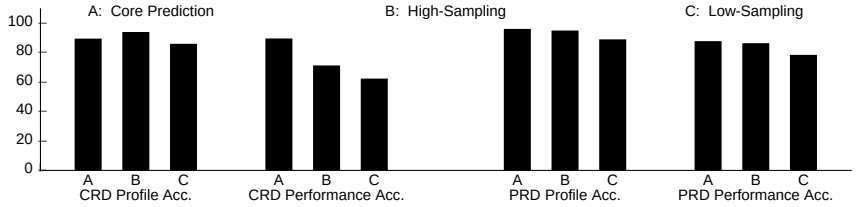

Fig. 29. Accuracy of prediction (A) versus sampling using high- (B) and low-sampling (C) rates for CRD and PRD profiles. Results are presented using the profile accuracy and performance accuracy metrics.

Section 4.3. In comparison, we find sampling achieves a slightly lower speedup, 3.3x, under the high-sampling rate across the same design space. However, under the low-sampling rate, sampling achieves a 34.4x speedup. These results demonstrate sampling's ability to target a wide range of different performance levels by tuning $R_{sampling}$ and $R_{pruning}$. As discussed in Section 4.4, prediction can also achieve higher speedups by predicting more profiles. Although we do not quantitatively evaluate these larger design spaces, we estimate a speedup of 59x when predicting every 4th core count instead of only powers of 2 (see Section 4.4). That said, sampling's speedup mechanism is more general since it does not require having to evaluate large design spaces.

Figure 29 compares the accuracy achieved by prediction and sampling. The bars labeled "A" in Figure 29 report the accuracy achieved by prediction across core count, first for CRD profiles and then for PRD profiles, under both the profile accuracy and performance accuracy metrics. Each A bar represents an average across 216 predictions, *i.e.* the cross product of 6 core counts $\times$ 9 benchmarks $\times$ 4 problem sizes. (Notice, these bars are identical to the average CRD_{direct} and PRD_{direct} bars in Figures 9–12). The bars labeled "B" in Figure 29 report the accuracy achieved by sampling using the high-sampling rate for CRD and PRD profiles under the profile and performance accuracy metrics. Each B bar represents an average across the same 216 profiles predicted in the A bars. And the bars labeled "C" in Figure 29 report accuracy from the same experiments as the B bars, except using the low-sampling rate.

Comparing the A and B bars for CRD profiles in Figure 29, we see prediction and sampling at the high-sampling rate achieve similar profile accuracy, but prediction achieves much higher performance accuracy–89.5% versus 71.2%. Due to pruning, sampling cannot sample memory references with extremely poor temporal reuse. As a result, there is significant distortion in sampled CRD profiles at large CRD values. This is reflected by the performance accuracy metric which equally weights error at both small and large CRD values. Looking at the C bars for CRD profiles in Figure 29, we see the problem gets worse under the low-sampling rate–performance accuracy drops to 62.3%–since even more pruning occurs in this case.

Comparing the A and B bars for PRD profiles in Figure 29, we see prediction and sampling at the high-sampling rate achieve similar accuracy under both profile and performance accuracy metrics. The same pruning problem occurs when sampling PRD profiles, giving prediction an advantage over sampling. On the other hand, prediction poorly predicts invalidations and coherence misses (see Section 4.3). Since sampling directly measures invalidations, it does not incur this error, giving it an advantage over prediction. Figure 29 shows these two sources of error tend to balance, resulting in similar prediction and sampling accuracies for PRD profiles. Looking at the C bars for PRD profiles in Figure 29, we see a similar result, though prediction achieves noticeably higher accuracy due to greater pruning error under the low-sampling rate.

Based on our results, we conclude prediction is superior to sampling for the 216-profile design space. Prediction achieves slightly higher speedup and significantly

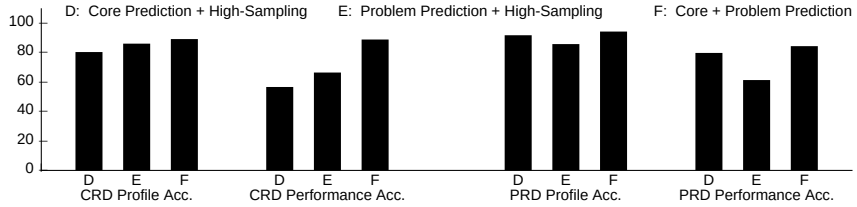

Fig. 30. Accuracy of combined core count prediction and sampling (D) versus combined problem size prediction and sampling (E) versus combined core count and problem size prediction (F).

higher accuracy for CRD profiles compared to sampling under the high-sampling rate. Sampling under the low-sampling rate can significantly outperform prediction, but yields abyssmal accuracy for CRD profiles and borderline accuracy for PRD profiles. Note, however, the best choice depends on the user's needs. For exploration of large design spaces, prediction is the better choice. But when acquiring a small number of profiles, there will be no advantage to using prediction, so sampling will be better.

We also try combining prediction and sampling. In particular, we use sampling under the high-sampling rate to acquire the 2- and 4-core profiles, and then predict the remaining core counts from the sampled profiles. The bars labeled "D" in Figure 30 report the accuracy achieved by this hybrid technique in a format identical to Figure 29. We find this approach achieves very good speedup, 19.5x, but inherits sampling's accuracy problems shown in Figure 29. Similar to sampling alone, prediction and sampling in combination achieve low CRD performance accuracy, only 56.6%. Even PRD performance accuracy suffers, reaching only 79.7%.

For completeness, we also try combining sampling with problem size prediction. We use sampling under the high-sampling rate to acquire the 2–256 core profiles at the S1 and S2 problems, and then predict the S3 and S4 problems at every core count from the sampled profiles. The bars labeled "E" in Figure 30 report the accuracy achieved by this hybrid technique. For comparison, the bars labeled "F" in Figure 30 report the accuracy achieved by our combined core count and problem size prediction technique from Section 6.4. We find sampling combined with problem size prediction achieves the highest speedup, 87.5x. But again, accuracy is an issue. CRD performance accuracy is only 66.4%. Worse, PRD performance accuracy drops to 61.3%. In all cases, the hybrid technique achieves noticeably lower accuracy then our combined core/problem prediction technique. These results show combining prediction and sampling has great promise in terms of speedup, but further work is needed to address accuracy.

8. RELATED WORK

This article is based on our own earlier work [Wu and Yeung 2011] which introduced the idea of coherent profile shift with core count scaling. While our previous research focused on CRD profiles, this article extends the analysis to PRD profiles. In addition, while our previous work originally defined the C_{core} parameter, this article adds to that the C_{share} parameter. This article also conducts a more extensive evaluation of the proposed techniques.

Several other researchers have investigated multicore RD analysis. Ding and Chilimbi [Ding and Chilimbi 2009] and Jiang *et al* [Jiang et al. 2010] present techniques to construct CRD profiles from per-thread RD profiles by analyzing memory traces. These techniques are general in that they can handle non-symmetric threads. But they are very complex because they consider all possible memory interleavings, limiting their use to small machine and problem sizes. Our work shows combinatorial analysis is unnecessary for loop-based parallel programs. For these programs, thread

interactions are relatively simple, allowing simple prediction techniques to achieve good accuracy. We exploit these properties to develop practical techniques that can handle real machines and problem sizes.

Schuff *et al* [Schuff et al. 2010] use sampling and parallelization techniques to accelerate profile acquisition. Their work considers profiling only 4 threads. Our work conducts a quantitative comparison between profile prediction and profile sampling for LCMPs (up to 256 threads). Although we do not consider parallelization, this is unlikely to provide much speedup due to the overheads associated with synchronizing a large number of threads. Our results suggest that as the number of profiles to acquire grows, prediction will develop a significant speedup advantage over sampling, and will yield more accurate profiles as well.

Another work by Schuff [Schuff et al. 2009] studies the accuracy of RD analysis for multicore processors. In addition, Berg *et al* [Berg et al. 2006] present a statistical model for computing miss rate from a CRD profile, and evaluate its accuracy. Both Schuff and Berg predict performance at different cache sizes, but they do not consider predicting configurations with more cores or larger problems beyond what was profiled, which is the focus of our work.

Finally, Chandra *et al* [Chandra et al. 2005] and Suh *et al* [Suh et al. 2001] have also developed locality models for multicore processors. And Xiang *et al* [Xiang et al. 2011] have developed an efficient method for computing working set footprint sizes. However, all of this prior work focused on multiprogrammed workloads consisting of sequential programs whereas we focus on multithreaded parallel programs. RD analysis has also been used to analyze uniprocessor caches [Zhong et al. 2003; Ding and Zhong 2003; Zhong et al. 2009]. As discussed earlier, our work borrows reference groups from Zhong *et al* [Zhong et al. 2003] to predict profile shift across core count scaling.

9. CONCLUSION

This article shows CRD and PRD profiles for loop-based parallel programs change predictably with core count scaling due to thread symmetry. As core count scales, both CRD and thread aggregated PRD (sPRD) profiles shift coherently to larger reuse distance values. We provide detailed analysis on how dilation, overlap, and intercepts for CRD profiles and PRD scaling, demotion absorption, and invalidations for PRD profiles contribute to the coherent shift. Using simple techniques, the profile shift can be predicted with high accuracy, enabling practical RD-based scaling analysis for LCMP-sized machines. Our results show predicted CRD and PRD profiles are 90% and 96% accurate, respectively, compared to directly measured profiles.

From our scaling analysis, we also define two application parameters that impact multicore cache design. For CRD profiles, because shifting is confined to smaller CRD values, core count scaling only impacts shared caches below a certain capacity. We define this capacity as C_{core}. We find a program's C_{core} value is much smaller than its total memory footprint, and grows roughly as the square-root of problem size. Moreover, because dilation and scaling are equivalent in the absence of sharing, CRD and sPRD profiles are coincident at small RD values where memory references are destined to mostly private data. Only at larger RD values does overlap occur, reducing CRD shift relative to sPRD shift. The reuse distance where CRD and sPRD profiles diverge, which we define as C_{share}, is the smallest capacity at which shared caches provide a cache-miss reduction compared to private caches. We find C_{share} is highly core count and problem size dependent, suggesting architectures that can adapt cache sharing at different levels of the cache hierarchy may outperform fixed organizations.

To demonstrate our techniques' ability to accelerate design space analysis, we use CRD and PRD profiles to predict shared LLC and private L2 cache performance across different configurations of a tiled multicore architecture. When combined with prob-

lem size prediction, our techniques can predict shared LLC MPKI to within 10.7% of simulation across 1,728 configurations; we can also predict private L2 MPKI to within 13.9% of simulation across 1,440 configurations. All predictions are performed from only 36 measured CRD and PRD profiles. Lastly, we compare profile prediction against profile sampling. For our design space, prediction achieves lower profiling time and higher accuracy compared to sampling. However, since sampling speeds up individual profiling runs, it is more desirable for quickly acquiring a small number of profiles.

REFERENCES

AGARWAL, A., BAO, L., BROWN, J., EDWARDS, B., MATTINA, M., MIAO, C.-C., RAMEY, C., AND WENTZLAFF, D. 2007. Tile Processor: Embedded Multicore for Networking and Multimedia. In *Proceedings of the Symposium on High Performance Chips*.

BARROW-WILLIAMS, N., FENSCH, C., AND MOORE, S. 2009. A Communication Characterisation of Splash-2 and Parsec. In *Proceedings of the IEEE International Symposium on Workload Characterization*.

BERG, E., ZEFFER, H., AND HAGERSTEN, E. 2006. A Statistical Multiprocessor Cache Model. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*.

BIENIA, C., KUMAR, S., AND LI, K. 2008a. PARSEC vs. SPLASH2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on ChipMultiprocessors. In *Proceedings of the IEEE International Symposium on Workload Characterization*.

BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008b. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

BINKERT, N., DRESLINSKI, R., HSU, L., LIM, K., SAIDI, A., AND REINHARDT, S. 2006. The M5 Simulator: Modeling Networked Systems. *IEEE Micro 26,* 4.

CHANDRA, D., GUO, F., KIM, S., AND SOLIHIN, Y. 2005. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

DAVIS, J., LAUDON, J., AND OLUKOTUN, K. 2005. Maximizing CMP Throughput with Mediocre Cores. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*.

DING, C. AND CHILIMBI, T. 2009. A Composable Model for Analyzing Locality of Multi-threaded Programs. Technical Report MSR-TR-2009-107, Microsoft Research.

DING, C. AND ZHONG, Y. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*.

EYERMAN, S., EECKHOUT, L., AND KARKHANIS, T. 2009. A Mechanistic Performance Model for Superscalar Out-of-Order Processors. *ACM Transactions on Computer Systems 27,* 2.

GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9,* 2.

HARDAVELLAS, N., FERDMAN, M., FALSAFI, B., AND AILAMAKI, A. 2009. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th International Symposium on Computer Architecture*.

HOSKOTE, Y., VANGAL, S., BORKAR, N., AND BORKAR, S. 2007. Teraflop Prototype Processor with 80 Cores. In *Proceedings of the Symposium on High Performance Chips*.

HSU, L., IYER, R., MAKINENI, S., REINHARDT, S., AND NEWELL, D. 2005. Exploring the Cache Design Space for Large Scale CMPs. *ACM SIGARCH Computer Architecture News 33*.

HUH, J., KECKLER, S. W., AND BURGER, D. 2001. Exploring the Design Space of Future CMPs. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*.

HUH, J., KIM, C., SHAFI, H., ZHANG, L., BURGER, D., AND KECKLER, S. W. 2005. A NUCA Substrate for Flexible CMP Cache Sharing. In *Proceedings of the International Conference on Supercomputing*. Boston, MA.

JALEEL, A., THEOBALD, K. B., JR., S. C. S., AND EMER, J. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the International Symposium on Computer Architecture*. Saint Malo, France.

JIANG, Y., ZHANG, E. Z., TIAN, K., AND SHEN, X. 2010. Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors? In *Proceeding of Compiler Construction*.

KARKHANIS, T. S. AND SMITH, J. E. 2004. A First-Order Superscalar Processor Model. In *Proceedings of the International Symposium on Computer Architecture*. Munich, Germany.

LI, J. AND MARTINEZ, J. F. 2005. Power-Performance Implications of Thread-level Parallelism on Chip Multiprocessors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*.

LI, S., AHN, J. H., STRONG, R. D., BROCKMAN, J. B., TULLSEN, D. M., AND JOUPPI, N. P. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the International Symposium on Microarchitecture*.

LI, Y., LEE, B., BROOKS, D., HU, Z., AND SKADRON, K. 2006. CMP Design Space Exploration Subject to Physical Constraints. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

MCCURDY, C. AND FISCHER, C. 2005. Using pin as a memory reference generator for multiprocessor simulation. *ACM SIGARCH Computer Architecture News 33*.

NARAYANAN, R., OZISIKYILMAZ, B., ZAMBRENO, J., MEMIK, G., AND CHOUDHARY, A. 2006. MineBench: A Benchmark Suite for Data Mining Workloads. In *Proceedings of the International Symposium on Workload Characterization*.

NAYFEH, B. A. AND OLUKOTUN, K. 1994. Exploring the Design Space for a Shared-Cache Multiprocessor. In *Proceedings of 21st International Symposium on Computer Architecture (ISCA-21)*. ACM, Chicago, IL, 166–175.

QASEM, A. AND KENNEDY, K. 2005. Evaluating a model for cache conflict miss prediction. Technical Report CS-TR05-457, Rice University.

ROGERS, B., KRISHNA, A., BELL, G., VU, K., JIANG, X., AND SOLIHIN, Y. 2009. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proceedings of the 36th International Symposium on Computer Architecture*.

SCHUFF, D. L., KULKARNI, M., AND PAI, V. S. 2010. Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*.

SCHUFF, D. L., PARSONS, B. S., AND PAI, J. S. 2009. Multicore-Aware Reuse Distance Analysis. Technical Report TR-ECE-09-08, Purdue University.

SUH, G. E., DEVADAS, S., AND RUDOLPH, L. 2001. Analytical Cache Models with Applications to Cache Partitioning. In *Proceedings of International Conference on Supercomputing*.

WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*.

WU, M.-J. AND YEUNG, D. 2011. Coherent Profiles: Enabling Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs. In *Proc. of the 20th International Conference on Parallel Architectures and Compilation Techniques*. Galveston Island, TX.

XIANG, X., BAO, B., DING, C., AND GAO, Y. 2011. Linear-time Modeling of Program Working Set in Shared Cache. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*. Galveston Island, TX.

ZHANG, M. AND ASANOVIC, K. 2005. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*.

ZHAO, L., IYER, R., MAKINENI, S., MOSES, J., ILLIKKAL, R., AND NEWELL, D. 2007. Performance, Area and Bandwidth Implications on Large-Scale CMP Cache Design. In *Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnect*.

ZHONG, Y. AND CHANG, W. 2008. Sampling-based Program Locality Approximation. In *Proceedings of 7th International Symposium on Memory Management (ISMM-7)*. Tuscon, AZ, 91–100.

ZHONG, Y., DROPSHO, S. G., AND DING, C. 2003. Miss Rate Prediction across All Program Inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*.

ZHONG, Y., SHEN, X., AND DING, C. 2009. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems 31,* 6.