Appears in *Proc. of the 20th Int'l Conf. on Parallel Architectures and Compilation Techniques*.
Galveston Island, TX. October 2011.

Coherent Profiles: Enabling Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs

Meng-Ju Wu and Donald Yeung
Department of Electrical and Computer Engineering
University of Maryland at College Park
{mjwu,yeung}@umd.edu

***Abstract*—Reuse distance (RD) analysis is a powerful memory analysis tool that can potentially help architects study multicore processor scaling. One key obstacle though is multicore RD analysis requires measuring *concurrent reuse distance (CRD) profiles* across thread-interleaved memory reference streams. Sensitivity to memory interleaving makes CRD profiles architecture dependent, preventing them from analyzing different processor configurations. For loop-based parallel programs, CRD profiles *shift coherently* to larger CRD 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 profiles feasible. Given the ubiquity and scalability of loop-level parallelism, such techniques will be extremely valuable for studying future large multicore designs.**

This paper investigates using RD analysis to efficiently analyze multicore cache performance for loop-based parallel programs, making several contributions. First, we provide in-depth analysis on how CRD profiles change with core count scaling. Second, we develop techniques to predict CRD profile scaling, in particular employing reference groups [1] to predict coherent shift, and evaluate prediction accuracy. Third, we show core count scaling only degrades performance for last-level caches (LLCs) below 16MB for our benchmarks and problem sizes, increasing to 64–128MB if problem size scales by 64x. Finally, we apply CRD profiles to analyze multicore cache performance. When combined with existing problem scaling prediction, our techniques can predict LLC MPKI to within 11.1% of simulation across 1,728 configurations using only 36 measured CRD profiles.

I. Introduction

Multicore processor performance depends in large part on how well programs utilize on-chip cache to mitigate off-chip accesses. Many studies have investigated this multicore memory bottleneck [2], [3], [4], [5], [6], [7], [8], [9]. 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 more complex as processors scale.

Today, 4–8 state-of-the-art cores or 10s of smaller cores [10], [11] along with 10s of MBs of cache can fit on a single die. Since Moore's law 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) [3], [9]–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 locality responsible for cache performance. For sequential programs, RD profiles are *architecture independent*. They can be acquired on one machine, and then used to predict different cache sizes without additional program runs. This saves time 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 [12], [13], [14], [15]. For parallel programs, not only can RD analysis predict performance across cache scaling, it can potentially predict performance across core count scaling as well [12], [13]. This can provide even greater leverage to save time when evaluating cache designs.

Compared to uniprocessors, RD analysis for multicore processors is more complex. This is because locality in multithreaded programs depends not only on per-thread reuse, but also on how simultaneous threads' memory references interleave. So, analyzing multicore workloads requires extending RD analysis to account for memory interleaving. For example, *concurrent reuse distance (CRD) profiles* quantify reuse globally across thread-interleaved memory reference streams [12], [13], [14], [15].

A major problem is memory interleaving–and hence techniques like CRD profiles–are *architecture dependent*. In particular, scaling core count increases the number of memory streams that interleave. So, CRD 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 profiles are

This research was supported in part by the Defense Advanced Research Projects Agency (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.

not even valid across different cache sizes at the *same* core count. Such architecture dependences prevent a single CRD profile from analyzing different multicore configurations, defeating the predictive benefits of RD analysis.

Recently, researchers have investigated constructing CRD profiles from per-thread RD profiles [12], [13]. By composing an increasing number of threads, CRD profiles for scaled CPUs can be derived and used to predict cache performance. Unfortunately, existing techniques are complex, employing trace-based analyses (some with exponential time complexity) to account for the combinatorially large number of ways that threads can interleave and interfere. Moreover, the techniques require at-scale profiling and traces. Hence, they are impractical for even moderately-sized machine/problem sizes, and completely out of the question for LCMPs.

In this paper, we show the complexity of analyzing memory interleaving depends in large part on how programs are parallelized. Parallel programs generally express either *task-level* or *loop-level* parallelism. In task-level parallel programs, threads often execute dissimilar code in an uncoordinated fashion, giving rise to irregular memory interleavings and complex thread interference. In loop-level 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 with less complex thread interference. This can be exploited to greatly simplify CRD profile prediction, and enable practical RD analysis for LCMP-scale systems.

While analysis techniques borne out of this observation will be specific to loop-level parallel programs, such workloads are pervasive. For example, all 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-level parallel programs are also highly scalable. Most can provide large amounts of parallelism simply by increasing problem size, so they are a good match for LCMPs. For these reasons, RD analysis for loop-level parallel programs will be extremely valuable to future multicore designers.

Our work makes the following contributions. First, we provide an in-depth analysis on how CRD profiles from loop-level parallel programs change with core count scaling. We find that as core count increases, CRD profiles *shift coherently*–*i.e.*, in a shape-preserving fashion–to larger CRD values due to CRD dilation for references to private data. Shifting slows down and eventually stops at large CRD due to overlapping references to shared data. Inter-thread shared references also cause intercepts that tend to spread and distort CRD profiles, but coherent shift is by far the dominant behavior.

Second, we develop techniques to predict the CRD profile movement. We employ reference groups [1], a technique previously used to predict RD profiles across problem scaling, to predict coherent shifting. We also propose uniformly distributing the portion of CRD profiles associated with shared references to predict spreading. To evaluate our techniques, we use the Intel PIN tool [16] to acquire CRD profiles across 9 benchmarks running 4 different problem sizes on 2–256 cores. We find our techniques can predict measured CRD profiles with 90% accuracy.

Third, we study the performance impact of core count scaling. Because CRD profile shifting stops beyond a certain point, core count scaling only impacts cache performance below the stopping point, which we call C_{core}. We measured C_{core} across our benchmarks and problem sizes, and found it is usually $<$ 16MB. But if problem size scales by 64x, C_{core} increases to 64–128MB.

Finally, we demonstrate our techniques' ability to accelerate design analysis. Using the M5 simulator [17], we model a tiled CMP, and simulate our benchmarks on processors with 2–256 cores and 4–128MB of last-level cache (LLC). In total, we simulate 1,728 configurations. Our core count prediction techniques can predict the LLC MPKI (misses per kilo-instructions) for all configurations within 11% of simulation using 72 measured CRD profiles. When combined with existing problem scaling prediction techniques, we can predict all configurations with similar accuracy using 36 measured CRD profiles.

The rest of this paper is organized as follows. Section II discusses CRD profiles and how they change with core count scaling. Then, Section III develops techniques to predict the scaling changes. Next, Section IV studies the performance implications of scaling. Lastly, Section V demonstrates our techniques' ability to accelerate cache evaluation. Sections VI and VII end with related work and conclusions.

II. Concurrent Reuse Distance

Reuse distance 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 LRU), 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 shared caches in multicore processors. Figure 1 shows a typical multicore cache hierarchy with multiple levels of cache on chip. Often, caches near the cores are private while caches near the off-chip interface are shared. The LLC (the focus of our work), is usually shared by *all* cores when it is a shared cache.

RD analysis can be extended for shared LLCs by computing reuse distance across the interleaved memory streams from all cores–*i.e.*, the *concurrent reuse distance (CRD)* [13]. Figure 2 illustrates CRD for a sequence of

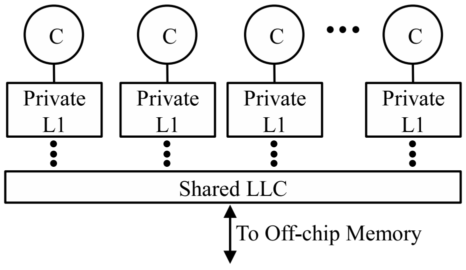

Figure 1. Multicore cache hierarchy.

Time:	1	2	3	4	5	6	7	8
Core1:	A		B	C			A	
Core2:		C			D (A)	E		F

Figure 2. Two interleaved memory reference streams, illustrating dilation, overlap, and intercept among inter-thread memory references.

interleaved memory references from two cores. In Figure 2, core 1 touches blocks A–C, and then re-references block A, while core 2 touches blocks C–F. Core 1's reuse of A has RD = 2, but its CRD = 4. In this case, CRD $>$ RD because some of core 2's interleaving references (D and E) are distinct from core 1's references, causing *CRD dilation*.

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 core 2's reference to C interleaves with core 1's reuse of A, this does not increase A's CRD because core 1 already references C in the reuse interval. Second, data sharing can also introduce *intercepts*. For example, if core 2 references A instead of D at time 5, then core 1's reuse of A has CRD = 1, so CRD actually becomes less than RD.

In the rest of this section, we study how dilation, overlap, and intercepts in loop-level parallel programs change with core count scaling, and show their impact on CRD profiles.

A. Profiling

To facilitate our study, we acquire CRD profiles using the Intel PIN tool. We maintain LRU stacks of memory blocks for each thread and for all threads in a program. (We assume 64-byte memory blocks). When a thread performs a data reference, PIN computes the memory block's depth in the per-thread and global LRU stacks. The former updates a per-thread RD profile, while the latter updates a CRD profile. Then, the referenced blocks are moved to the MRU stack position. Our PIN tool follows McCurdy and Fischer's method [18] and performs functional execution only, context switching between threads after every memory reference. This tends to interleave threads' memory references uniformly in time at the global LRU stack.

Because dilation, overlap, and intercepts 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.[1] 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 parallel regions, we acquire CRD profiles for references to mostly private versus shared data separately. The former, which we call *private CRD profiles* (CRD$_P$), exhibit very few intercepts, so they show the combined effects of dilation and overlap. The latter, which we call *shared CRD profiles* (CRD$_S$), contain frequent intercepts, so they show intercept effects. We employ a single global LRU stack for computing CRD$_P$ and CRD$_S$. To acquire these profiles, we record each memory block's CRD values separately[2] 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 counts into either the CRD$_P$ or CRD$_S$ profiles based on their observed sharing.

Finally, we quantify overlap in CRD$_P$ profiles. We maintain a second global LRU stack in which we artificially remove all overlapping references. This is done by appending each thread's ID to the address of their executed memory references when calculating reuse distance (tracking private vs shared memory blocks still uses the unmodified addresses). Then, we compute CRD$_P$ profiles from the second global LRU stack exactly as described above. We call these *private no-overlap CRD profiles* (CRD$_{PN}$). In CRD$_{PN}$ profiles, inter-thread references are always unique, so CRD never contracts. Comparing CRD$_{PN}$ and CRD$_P$ profiles shows the impact of overlapping references.

B. Scaling

Figures 3 and 4 show how core count scaling affects CRD profiles using FFT from SPLASH2 [19] as an example. CRD profiles are presented for the most important parallel region in FFT. Each profile plots reference count (Y-axis) versus CRD (X-axis). CRD values are multiplied by the block size, 64 bytes, so the X-axis reports distance as 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.

1) Dilation and Overlap: Figure 3A plots FFT's CRD$_{PN}$ profile for a 4-core execution (labeled "CRD$_{PN4}$") along with the summation of per-thread RD profiles of the 4 threads (labeled "RD$_4$"). These two profiles show the dilation effect without overlap at 4 cores. From Figure 3A, we can see CRD$_{PN4}$ shifts RD$_4$ to larger CRD values. The dilation is by exactly a factor 4x. More importantly,

[1]We instrument PIN to recognize each program's barrier function.

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

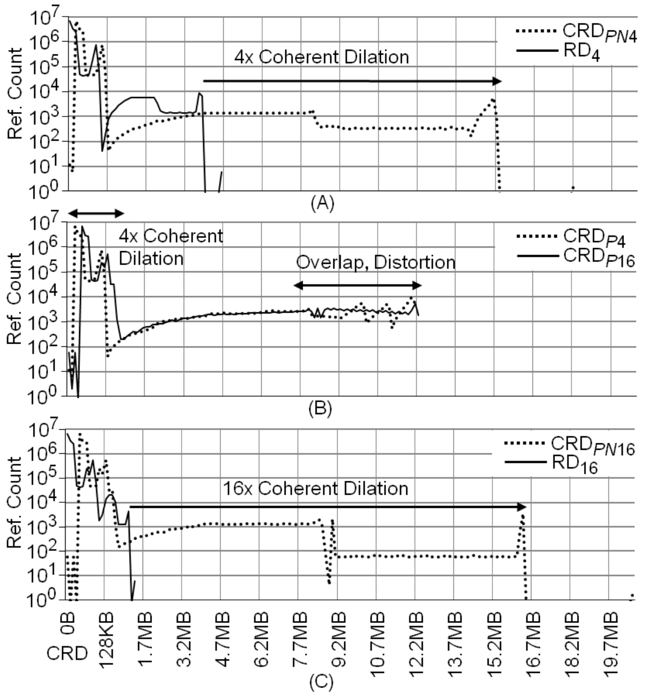

Figure 3. FFT's CRD$_{PN}$ and per-thread RD profiles for 4 (A) and 16 (C) cores; CRD$_P$ profiles for 4 and 16 cores (B).

the CRD$_{PN4}$ profile maintains the shape of the RD$_4$ profile across the shift.

Per-thread RD dilation is shape preserving 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 3 threads tend to interleave RD unique memory blocks each (due to thread symmetry), so CRD $\approx 4 \times RD$. Consequently, all memory references at all per-thread RD values shift together such that CRD$_{PN4}$ is a coherent–*i.e.*, distortion-free–scaling of RD$_4$.

Overlap offsets dilation, reducing its shift. To illustrate, Figure 3B 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$_{PN4}$ are almost identical at small CRD, but CRD$_{P4}$ exhibits less shift at large CRD due to the overlapping references. In our benchmarks, overlap 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 3B plots the CRD$_P$ profile for a 16-core FFT (labeled "CRD$_{P16}$"), and Figure 3C plots the corresponding RD$_{16}$ and CRD$_{PN16}$ profiles. As in the 4-core case, CRD$_{P16}$ coherently scales

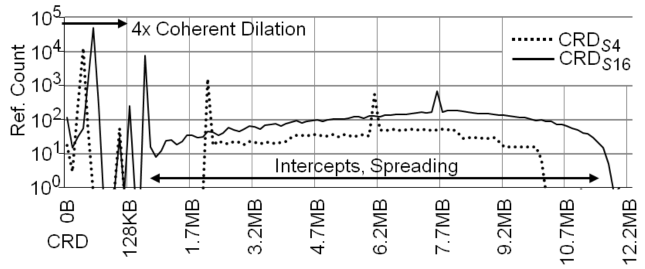

Figure 4. FFT's CRD$_S$ profiles for 4 and 16 cores.

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.

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 reuses. This truncation, along with the overlapping references at large CRD, almost perfectly cancel the additional 4x dilation. So in Figure 3B, 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 is roughly the same.

In summary, 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 only for FFT, we find this behavior is pervasive across all the benchmarks and loops we studied (see Table I).

2) Intercepts: Figure 4 plots FFT's CRD$_S$ profiles at 4 and 16 cores (labeled "CRD$_{S4}$" and "CRD$_{S16}$"). These show how intercepts change with core count scaling. Figure 4 highlights two behaviors. First, at small CRD ($<$ 128KB), core count scaling induces a coherent shift of CRD$_{S4}$ by a factor 4x, similar to CRD$_P$. As mentioned earlier, sharing tends to occur across distant loop iterations. So, like overlap, 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 bisecting thread 1's reuse, making CRD = 1. But if the intercept occurs at time 6, CRD = 0, and if the intercept occurs at time 2, CRD = 4. In general, intercepts spread references with per-thread reuse at distance RD between 0

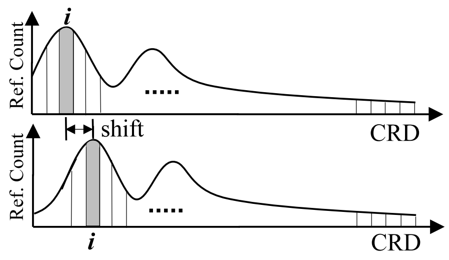

Figure 5. Detecting alignment and shifting using reference groups.

and $P \times RD$, where P is the core count. Although not so for Figure 4, intercepts often introduce significant distortion.

Notice the CRD$_S$ profile contains fewer memory references than the CRD$_P$ profile. In Figure 4, the CRD$_S$ profile only accounts for 1% of the parallel region's references (compared to Figure 3). While the exact percentage is application dependent, we find CRD$_P$ profiles always dominate.

III. Profile Prediction

This section studies techniques to predict CRD profiles across core count scaling. We describe our techniques, discuss evaluation methodology, and then present results.

A. Prediction Techniques

Section II-B shows CRD$_P$ and CRD$_S$ profiles change differently across core count scaling, so we predict them separately. Based on our insights, we employ two techniques–one for coherent shift and another for spread.

1) Coherent Shift: As shown in Section II-B, CRD$_P$ (and to some extent, CRD$_S$) profiles from parallel loops exhibit coherent shift. Coincidentally, Zhong *et al* [1] found RD profiles for sequential programs also exhibit similar shifting due to problem scaling, and proposed *reference groups* to predict the shift. We use reference groups to predict CRD$_P$ and CRD$_S$ profiles. Figure 5 illustrates the technique.

Samples of profiles (either CRD$_P$ or CRD$_S$) are acquired at 2 and 4 cores. These sampled profiles are divided into 200,000 groups along their CRD axis, each containing an equal fraction (0.0005%) of the profile's references. Reference groups across sampled profiles are *aligned* via association: the i^{th} group in the 2-core sample is aligned to the i^{th} group in the 4-core sample. Aligned reference groups "correspond" to each other across the shift and are assumed to shift together (*i.e.*, coherently), maintaining 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. In addition, 3 intermediate shift rates are allowed: cube root, square root, and cube-root squared. These different shift rates support variable shift (*e.g.*, in CRD$_P$ profiles).

The shift between pairs of reference groups in the sampled 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 prediction from the CRD$_P$ samples is the predicted CRD$_P$ profile. The prediction from the CRD$_S$ samples, which we call CRD$_{Sshift}$, is combined with spread prediction below to derive predicted CRD$_S$ profiles.

2) Spread: Section II-B 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 sample the CRD$_S$ profile at 4 cores (the same sample used in shift prediction), and uniformly distribute the reference counts ($CRD_{S\_4core}[k] \times p[k]$) at each CRD between 0 and $k \times \frac{core\_count}{4}$, where k is a particular CRD value, and $p = \frac{k}{C_{max}}$ (C_{max} is the CRD profile's maximum CRD value). We call this prediction 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 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).

B. Prediction Methodology

We use PIN to acquire the CRD profiles for our study. In particular, we acquire the per-parallel region CRD$_P$ and CRD$_S$ profiles, as described in Section II-A, 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$_S$ profiles. Then, we use the techniques from Section III-A to predict the CRD$_P$ and CRD$_S$ profiles for 8–256 cores, in powers of 2, from the 2- and 4-core samples. At each core count, we sum all predicted per-parallel region CRD$_P$ and CRD$_S$ profiles to form a single prediction for the whole-program CRD profile.

As discussed in Section II-B, CRD$_P$ profiles dominate CRD$_S$ profiles. This implies predicting coherent shift alone may be sufficient in many cases. In addition to predicting CRD$_P$ and CRD$_S$ profiles separately, we also employ whole-program CRD profile prediction. We use PIN to acquire the whole program CRD profiles at 2 and 4 cores. Then, we use reference groups to predict the whole-program profiles for 8–256 cores directly from the sampled whole-program profiles. The advantage of this approach is it doesn't require profiling individual parallel regions.

Our study employs 9 benchmarks, each running 4 problem sizes. Table I lists the benchmarks: FFT, LU, RADIX, Barnes, FMM, Ocean, and Water from the SPLASH2 suite [19], KMeans from MineBench [20], and BlackScholes from PARSEC [21]. The 2^{nd} column of Table I specifies the 4 problem sizes, S1–S4. For each benchmark and problem

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	$2^{17}/2^{19}/2^{21}/2^{23}$ keys, radix=2048	23/93/422/1,687	23/90/433/1,729
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

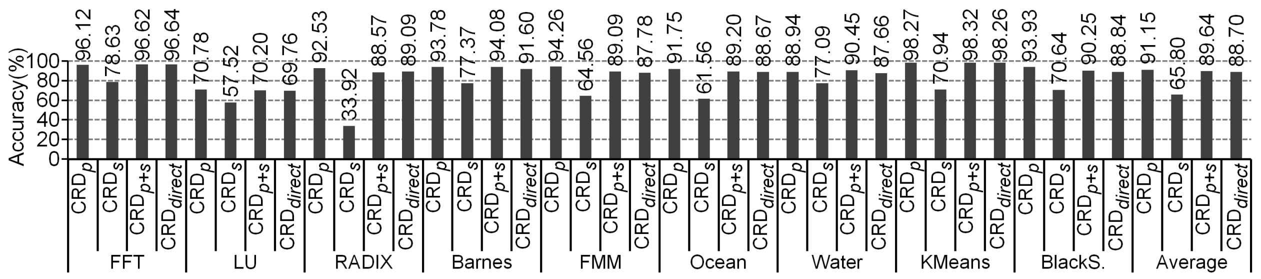

Figure 6. CRD accuracy of predicted CRD$_P$ and CRD$_S$ profiles, and indirectly and directly predicted whole-program CRD profiles.

size, we predict the whole-program CRD profiles at 8–256 cores (either indirectly by predicting CRD$_P$ and CRD$_S$ profiles, or directly), yielding 24 predicted profiles per benchmark. Using PIN, we also acquire the actual whole-program CRD profiles corresponding to these 24 predictions, and compare the measured and predicted profiles.

Profile comparisons use two metrics, *CRD accuracy* and *CMC error*. CRD accuracy is $1 - \frac{E}{2}$, where E is the sum of the normalized absolute differences between every pair of CRD values from a predicted and measured CRD profile. (E can be at most 200%, so CRD accuracy is between 0–100%). CRD accuracy is a similarity metric used in previous work [13], [22]. CMC (cache-miss count) error is computed from CMC profiles which present the number of cache misses predicted by a CRD profile at each of its CRD values (*i.e.*, $CMC[i] = \sum_{j=i}^{N-1} CRD[j]$, where N is the total number of bins). We compute CMC error by averaging the error between pairs of CRD values from the first half of predicted and measured CMC profiles:

$$CMC\ error = \frac{2}{N} \sum_{k=0}^{\frac{N}{2}} \frac{|CMC_{pred}[k] - CMC_{meas}[k]|}{CMC_{meas}[k]}$$

For CRD values, k, in which both $CMC_{pred}[k]$ and $CMC_{meas}[k]$ are $<$ 20,000, we assume the error is 0. Enormous error can occur at such CRD values, but they are inconsequential due to the small cache miss counts. (In our benchmarks, the removed CRD values account for less than 0.5% of the benchmarks' total memory references).

CMC error reflects LLC performance. Because CRD accuracy is an absolute metric, it more heavily weights error at the first few CRD values where reference counts are enormous but which occur well below LLC capacities. Because CMC error is an average metric, it equally weights error across the first half of CMC profiles which usually extend well beyond LLC capacities.

Finally, our study ignores the benchmarks' initialization phases which are sequential; we only acquire and predict CRD 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.

C. Accuracy Results

Figure 6 presents our CRD profile prediction results. In Figure 6, 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 6 reports the average CRD accuracy achieved over the 24 predictions per benchmark. The rightmost bars report the average across all benchmarks.

As Figure 6 shows, CRD$_P$ profiles are predicted with high accuracy. For all benchmarks except LU, CRD$_P$ accuracy is between 88% and 99%. For LU, CRD$_P$ accuracy is 71%. Across all benchmarks, the average CRD$_P$ accuracy is 91%. CRD$_P$ profiles exhibit coherent shift across core count scaling which reference groups can effectively predict. The

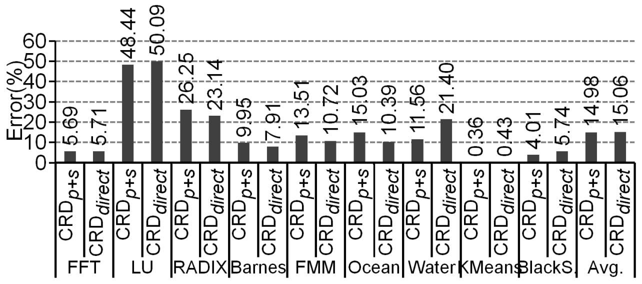

Figure 7. CMC error for predicted whole-program CRD profiles.

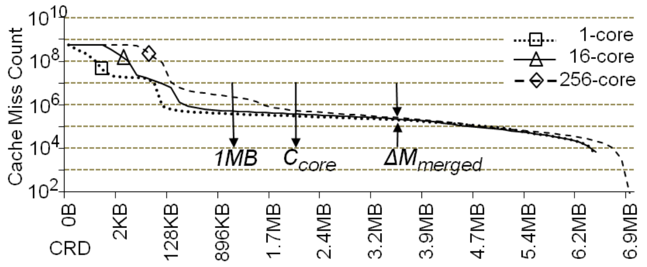

Figure 8. CMC profiles for Barnes on 1, 16, and 256 cores running the S2 problem. C_{core} and ΔM_{merged} are labeled.

results in Figure 6 demonstrate the pervasiveness of coherent shift across our benchmarks, and confirm the accuracy of reference groups for this type of profile movement.

LU is the only benchmark with 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 error when predicting large core counts.

Compared to CRD$_P$ profiles, CRD$_S$ profiles are predicted with lower accuracy. In Figure 6, CRD$_S$ accuracy is between 33% and 79%. Across all benchmarks, the average CRD$_S$ accuracy is only 66%. CRD$_S$ profiles suffer poor spread prediction. While intercepts induce spreading in the range we expect (see Section II-B), the actual distribution across this range is highly application dependent. Unfortunately, our simple uniform spread model does not capture all of the behaviors, leading to lower prediction accuracy.

Although CRD$_S$ profiles are predicted with lower accuracy, the impact on overall prediction accuracy is minimal. In Figure 6, the bars labeled "CRD$_{P+S}$" report the average CRD accuracy for whole-program CRD profiles predicted by combining CRD$_P$ and CRD$_S$ predictions. For all benchmarks except LU, CRD$_{P+S}$ accuracy is between 88% and 99% (for LU, it is 70%). The average CRD$_{P+S}$ accuracy for all benchmarks is 90%. These results confirm CRD$_P$ dominates CRD$_S$. So, predicting CRD$_P$ effectively leads to accurate whole-program CRD profile prediction.

Since CRD$_P$ dominates, one would expect predicting whole-program CRD profiles directly to be the same as (and hence, achieve similar accuracy compared to) predicting CRD$_P$ profiles. This is in fact the case. The last set of bars in Figure 6, labeled "CRD$_{direct}$," report the average CRD accuracy for direct whole-program CRD profile prediction. Figure 6 shows CRD$_{direct}$ is just slightly worse than CRD$_{P+S}$. On average, CRD$_{direct}$ accuracy is 89%, compared to 90% for CRD$_{P+S}$.

Finally, Figure 7 reports the whole-program prediction results from Figure 6 using the CMC error metric (note, smaller values are better). Qualitatively, the CMC error and CRD accuracy results are the same. CRD$_{P+S}$ and CRD$_{direct}$ have similar CMC error. They are between 0.3%–27% for 8 benchmarks, and are roughly 50% for LU. On average, CRD$_{P+S}$ and CRD$_{direct}$ achieve a 10.8% and 10.7% error, respectively, without LU, and 15.0% and 15.1% error, respectively, for all benchmarks. This result suggests our predicted CRD profiles can provide good LLC cache miss predictions. Later, Section V will confirm this result.

IV. Scaling Implications

Thus far, we have shown core count scaling causes predictable coherent shift of CRD profiles. Another point related to core count scaling is that shifting is limited: it slows down and stops at large CRD. This has important cache performance implications. To illustrate, Figure 8 plots the whole-program CMC profiles for the Barnes benchmark running the S2 problem on 1, 16, and 256 cores. Because CRD profiles eventually stop shifting, their associated CMC profiles merge at some point. In this study, we measure this stopping point for 256 cores, and we call it "C_{core}." As Figure 8 shows, C_{core} delineates cache-miss impact. At CRD $< C_{core}$, cache misses increase significantly with core count, but at CRD $> C_{core}$ cache misses do not increase much. In other words, *core count scaling degrades locality in parallel loops, but it only impacts cache sizes below C_{core}.* Caches bigger than C_{core} are not affected by scaling.

We quantify C_{core} and the cache-miss increases caused by shifting across the measured whole-program CRD profiles from Section III-B. This is done as follows. For each benchmark and problem size, we derive the CMC profiles for 1–256 cores (*e.g.,* like Figure 8). At a given CRD value, we define ΔM to be the ratio of cache-miss counts between the 256- and 1-core CMC profiles. We first compute ΔM at CRD = $\frac{C_{max}}{2}$, well beyond C_{core} where the CMC profiles have almost 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}. This CRD value is C_{core}. Lastly, we compute ΔM at every CRD value between 1MB and C_{core}, 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 boundary focuses our analysis on LLC-sized caches.

Table II reports results broken down by benchmark and problem size, with rows labeled "Average" showing averages

Table II
C_{core}, ΔM_a, ΔM_m, AND C_{max} FOR OUR BENCHMARKS.

Benchmark	S1	S2	S3	S4
C_{core}				
FFT	1.1MB	2.3MB	5.5MB	16.0MB
LU	135.8KB	211.8KB	382.1KB	755.9KB
RADIX	-	-	8.0MB	22.8MB
Barnes	555.3KB	1.9MB	4.2MB	7.2MB
FMM	733.9KB	1.6MB	7.6MB	9.3MB
Ocean	707.2KB	1.5MB	4.8MB	17.3MB
Water	254.9KB	628.7KB	1.7MB	2.9MB
KMeans	496.1KB	496.1KB	496.1KB	496.1KB
BlackScholes	352.1KB	352.2KB	352.2KB	368.3KB
Average	540.4KB	1.1MB	3.7MB	8.5MB
C_{max}				
FFT	4.3MB	14.3MB	52.3MB	200.3MB
LU	785.8KB	2.3MB	8.3MB	32.4MB
RADIX	30.2MB	36.2MB	60.2MB	156.2MB
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	5.0MB	19.2MB	76.2MB	304.2MB
BlackScholes	1.7MB	6.2MB	24.2MB	96.2MB
Average	6.2MB	13.3MB	40.6MB	148.9MB
ΔM_a / ΔM_m				
FFT	3.3 / 4.2	4.2 / 5.0	4.1 / 5.5	3.5 / 6.0
LU	- / -	- / -	- / -	- / -
RADIX	- / -	- / -	5.3 / 7.7	3.2 / 6.6
Barnes	- / -	3.7 / 6.2	3.7 /8.6	3.7 / 10.2
FMM	- / -	2.5 / 3.0	2.2 / 3.2	2.3 / 3.4
Ocean	- / -	3.4 / 4.5	1.6 / 2.1	1.2 / 1.8
Water	- / -	- / -	1.8 / 2.2	1.9 / 2.4
KMeans	- / -	- / -	- / -	- / -
BlackScholes	- / -	- / -	- / -	- / -
Average	3.3 / 4.2	3.4 / 4.7	3.1 / 4.9	2.6 / 5.1

across all benchmarks. The top portion of Table II reports C_{core}.[3] As these data show, C_{core} varies between 135KB and 23MB. On average, C_{core} is between 540KB and 9MB for the different problem sizes. These results show the impact of core count scaling for our benchmarks and problem sizes is confined to smaller LLCs, usually $<$ 16MB. Large LLCs ($>$ 16MB) will not experience significant cache-miss increases due to core count scaling.

C_{core} is particularly small for LU, KMeans, and BlackScholes, never exceeding 756KB. The working sets for these benchmarks are extremely small. For programs with such good locality, the profile shift due to core count scaling is minimal. So, core count scaling will never impact the miss rates of reasonably sized LLCs in these programs.

Although our C_{core} values correspond to small LLCs, Table II shows C_{core} generally increases with problem size. This is because problem scaling also shifts CRD profiles [1]. 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.

[3]Results for RADIX at S1/S2 are missing 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.

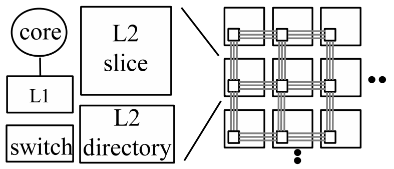

Figure 9. Tiled CMP. Each tile contains a core+L1 cache, an L2 cache and directory "slice," and an on-chip network switch.

To quantify the impact of continued problem scaling, the middle 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, Table II shows C_{core} increases at a sub-linear rate, roughly as the square root of 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 modestly larger problems, core count scaling will impact large LLCs.

Finally, 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 5.3 while ΔM_m varies between 1.8 and 10.2. On average, ΔM_a (ΔM_m) is between 2.6 (4.2) and 3.4 (5.1) across different problem sizes. These results show core count scaling can increase cache misses significantly for LLC sizes below C_{core}. Consider scaling core count by 256x creates a 2 orders of magnitude increase in off-chip bandwidth due to parallelism (assuming linear bandwidth increase with core count). Table II shows the same 256x increase in cores can create up to another order of magnitude in cache misses (and hence, total memory traffic) due to locality degradation.

V. Performance Prediction

We now demonstrate the ability of CRD profile prediction to accelerate multicore design evaluation. We employ our core count prediction techniques from Section III to assess cache performance–in particular, LLC MPKI. We also incorporate previous techniques for predicting problem scaling to further increase the acceleration advantage. In the rest of this section, we describe architectural assumptions, discuss how we predict performance, and then present results.

A. Architecture Assumptions

Our performance study assumes *tiled CMPs* [23]. A tiled CMP, illustrated in Figure 9, consists of several identical replicated tiles, each containing a core, a private L1 cache, an L2 cache "slice," and a switch for a 2-D on-chip point-to-point mesh network. Tiled CMPs are scalable [23], [24], so they permit us to study a large design space on a single multicore platform.

In our study, the L2 slices across all tiles are managed as a single logically shared LLC. We assume the LLC does

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
IL1/DL1	32KB/32KB, 64B block, 8-way, 1 cycle
Total L2 Cache Size	4MB, 8MB, 16MB, 32MB, 64MB, 128MB
L2 Slice	64B blocks, 32-way, 10 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)

not replicate or migrate cache blocks between slices. Each cache block is always placed in the same L2 slice, known as the cache block's "home." We assume cache block homes are page-interleaved (with 8KB page size) across L2 slices according to their physical address.

To address remote-L2 slice latency, we permit replication at the private L1 caches. We employ a directory-based MESI cache coherence protocol for L1 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 access, hops through the network, L2 slice access, 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 and 4–128MB of total L2 cache (LLC).

In addition to performance, our M5 simulator also measures whole-program CRD profiles. We track CRD for the simulated interleaved memory reference stream from all cores using the same approach as our PIN tool. Similar to the PIN tool, CRD is computed at a 64-byte granularity, the block size for both the L1 and L2 caches.

To drive our simulations, we use the same benchmarks and problem sizes from Table I, simulating the same parallel regions described in Section III-B. (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 profiles. For benchmarks running the entire parallel phase, we do not perform any explicit cache warmup.

B. Performance Prediction

Figure 10 illustrates the 3-D architecture-problem space formed by the combination of all core counts, LLC capacities, and problem sizes in our study. There are 192 configurations per benchmark in this architecture-problem space, and 1,728 configurations across our 9 benchmarks. We simulate all of them using our M5 simulator, obtaining their LLC MPKI and CRD profiles. (When computing LLC MPKI, we exclude compulsory misses since reuse distance profiles do not predict them). Similar to our profile prediction study, we use a sub-set of the measured CRD profiles to predict LLC MPKI at all configurations, and compare against the simulated LLC MPKI to assess accuracy.

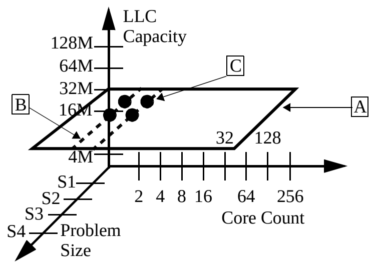

Figure 10. 3-D architecture-problem space.

Performance prediction is a two-step process: first we acquire/predict CRD profiles, and then we use the CRD profiles to predict cache performance. The first step's goal is to 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 10. We use 3 profile prediction strategies for doing this: "No-Pred," "C-Pred," and "CP-Pred."

No-Pred does not perform any profile prediction, and simply uses the complete set of measured CRD profiles in the "A" plane of Figure 10 to predict performance. No-Pred requires sampling 32 CRD profiles to make the 192 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 10. We use our techniques for directly predicting whole-program CRD profiles from Sections III-A and III-B. C-Pred requires sampling 8 CRD profiles to make the 192 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 10. To predict across problem size, CP-Pred uses the same reference groups technique described in Section III-A. 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 [1]). CP-Pred requires sampling 4 CRD profiles to make the 192 LLC MPKI predictions per benchmark.

Once all 32 CRD profiles within the "A" plane have been

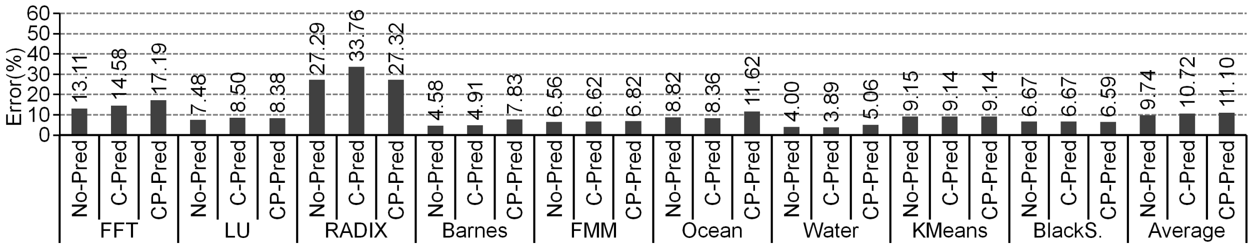

Figure 11. Percent LLC MPKI prediction error.

acquired, the second step is to predict LLC MPKI. We derive the corresponding CMC profile from each CRD profile, and extract cache-miss counts at 4–128MB on the CMC profile. This predicts the number of capacity misses. We use Qasem and Kennedy's model [25] to predict conflict misses. This model takes the CRD profile as input, 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 MPKI. For No-Pred and C-Pred, we use the measured IC at the same configuration that contributed the CRD profile for LLC MPKI prediction–*i.e.*, we assume IC doesn't change across either LLC capacity or core count. 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 profile).

C. Performance Results

Figure 11 reports percent error ($|predicted - measured|$ / $measured$) in our LLC MPKI predictions. When measured LLC MPKI is near zero, the error blows up. To prevent this, we add a small offset, 0.05, to the predicted and measured values before computing error. Each bar in Figure 11 reports the average error across all predictions for a particular prediction strategy and benchmark (*i.e.*, for 192 configurations). The rightmost bars report averages across all benchmarks.

As Figure 11 shows, No-Pred is able to predict LLC MPKI within 14% of simulation on average for 8 out of 9 benchmarks, and within 28% for RADIX. Across all benchmarks, prediction error is 9.7%. This is the baseline CRD profile error (*i.e.*, without profile prediction), and reflects 3 error sources. First, M5 profiles include timing effects (PIN profiles do not). Since LLC capacity scaling alters thread timing, CRD profiles may not accurately predict LLC sizes different from the ones used to measure them. We compared CRD profiles across different LLC sizes and found they are in fact almost identical. This is again due to symmetric threads. Although symmetric threads speedup or slow down with LLC scaling, they do so at the *same rate*, thus preserving memory interleaving and CRD profiles. So, sensitivity to LLC scaling is not a major source of error.

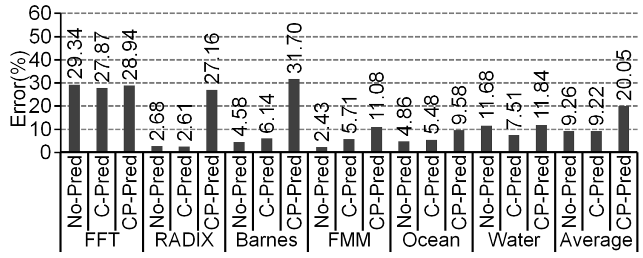

Figure 12. LLC MPKI prediction error for S4 and 4–16MB LLCs.

Second, the cache conflict model introduces error. In particular, processors with large core count and small LLCs can incur pathologic conflicts that the conflict model cannot predict. We find this is the dominant source of error in the No-Pred results. And third, our error metric does not always address numeric instability. In some cases, LLC MPKI is near 0.05. These are not eliminated by our 0.05 offset, but are small enough to make percent error very sensitive to minute prediction errors. This is responsible for the high errors in RADIX. On average, though, No-Pred error is very low, and shows CRD profiles are capable of accurately predicting LLC MPKI for loop-based parallel programs.

Figure 11 also shows our profile prediction techniques are very effective. In Figure 11, C-Pred error is only slightly worse than No-Pred. Furthermore, CP-Pred does not noticeably increase error over C-Pred. Figure 11 shows both techniques are able to predict LLC MPKI within 18% of simulation for 8 out of 9 benchmarks, and within 34% for RADIX. On average, prediction error is within 11.1%. These results confirm profile prediction has high accuracy, as was shown in Section III-C. Another reason C-Pred and CP-Pred perform similarly to No-Pred is because errors often cancel. While the cache conflict model usually under-predicts conflict misses, CRD profile prediction (for both core count and problem scaling) usually over-predicts capacity misses. This also explains why C-Pred and CP-Pred sometimes achieve lower error than No-Pred in Figure 11.

Figure 12 reports LLC MPKI prediction error, just like Figure 11, but only for the S4 problem and 4–16MB LLCs (it still includes 2–256 cores). Most of these configurations have LLC size $< C_{core}$; hence, Figure 12 studies prediction error in the region of CRD profile shift. (LU, KMeans, and BlackScholes are omitted because their C_{core} are always

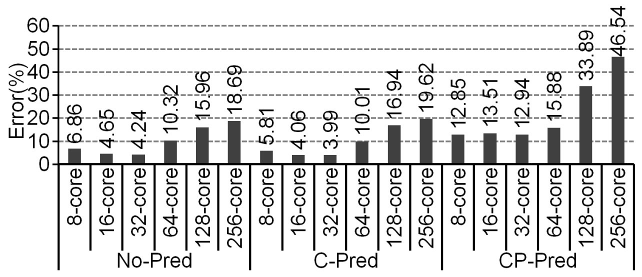

Figure 13. Prediction error for S4 and 4–16MB LLCs by core count.

below our smallest LLC). As Figure 12 shows, prediction error in the shifting region is comparable to the entire space. Error gets worse for FFT and Barnes. But it improves in RADIX, especially for No-Pred and C-Pred because Figure 12 does not include RADIX's poorly predicted cases. Overall, C-Pred has the same accuracy as No-Pred, 9% error, showing our core count prediction techniques are effective in the shifting region. CP-Pred is worse–20.1% error–due to more significant over-prediction of shifting.

Finally, Figure 13 reports results for the same problem and LLC sizes in Figure 12 broken down by strategy and core count. Like Figure 12, Figure 13 shows C-Pred is similar to No-Pred, while CP-Pred is worse. More importantly, Figure 13 also shows prediction error increases with core count, reaching 19% for No-Pred, 20% for C-Pred, and 47% for CP-Pred. This illustrates the cache conflict model errors mentioned earlier which get worse with core count. Even so, Figure 13 shows C-Pred's error is still reasonable when predicting large core counts.

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

VI. Related Work

Several researchers have investigated multicore RD analysis. Ding and Chilimbi [12] and Jiang *et al* [13] 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, only memory references within parallel regions interleave, reducing the number of cases to analyze. Furthermore, within parallel regions, dilation, overlap, and intercepts exhibit simple behavior, allowing simple prediction techniques to achieve good accuracy. We exploit these properties to develop practical techniques that can handle real machines and problem sizes. Another difference is [12], [13] require at-scale profiling. In contrast, we only profile small-scale machines from which CRD profiles of scaled configurations are derived to enable scaling analysis.

Schuff *et al* [15] use sampling and parallelization techniques to accelerate CRD profiling. Our work is orthogonal to these techniques. We reduce the number of needed profiles whereas Schuff *et al* reduce the time per profile run. It is important to note that while Schuff's approach is fast, it still incurs significant overhead: 80X slowdown on average and up to 496X slowdown compared to native execution [15]. And this is for profiling only 4 threads; overheads will certainly be higher for profiling 100s of threads. Hence, even with profiling acceleration, it is still very difficult to exhaustively explore LCMP design spaces that can reach 1000s of configurations.

Another work by Schuff [14] investigates the accuracy of RD analysis for multicore processors. In addition to predicting shared cache performance, they also predict private caches which we do not address. Berg *et al* [26] 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 cannot predict configurations with more cores or larger problems beyond what was profiled, which is the focus of our work.

Chandra *et al* [27] and Suh *et al* [28] have also developed locality models for multicore processors, but they focus on multiprogrammed workloads whereas we focus on multithreaded programs. RD analysis has also been used to analyze uniprocessor caches [1], [22], [29]. As discussed earlier, our work borrows reference groups from Zhong *et al* [1] to predict profile shift across core count scaling.

Finally, profile prediction is related to machine learning for design exploration [30], [31]. The latter tries to model how general features impact performance, whereas we model how memory features impact CRD profiles. Our approach learns more per sample (a CRD profile), reducing the number of needed samples, but we can only optimize memory. ML learns very little per sample (an IPC value), but is very general and can optimize any architecture feature.

VII. Conclusion

This paper shows CRD profiles for loop-based parallel programs change predictably with core count scaling due to thread symmetry. As core count scales, CRD profiles shift coherently to larger CRD values. Using simple techniques, the CRD movement can be predicted with high accuracy, enabling practical RD-based scaling analysis for LCMP-sized machines. We also show that because shifting is confined to smaller CRD values, core count scaling only impacts LLCs below the C_{core} parameter. Lastly, to demonstrate benefits, we use CRD profiles to predict LLC performance across an LCMP design space. Our techniques can predict LLC MPKI to within 11.1% of simulation across 1,728 configurations using only 36 measured CRD profiles.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful comments, Collin McCurdy for his PIN tool, and Abdel-Hameed Badawy for insightful discussion.

References

[1] Y. Zhong, S. G. Dropsho, and C. Ding, "Miss Rate Prediction across All Program Inputs," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[2] J. Davis, J. Laudon, and K. Olukotun, "Maximizing CMP Throughput with Mediocre Cores," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[3] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell, "Exploring the Cache Design Space for Large Scale CMPs," *ACM SIGARCH Computer Architecture News*, vol. 33, 2005.

[4] J. Huh, S. W. Keckler, and D. Burger, "Exploring the Design Space of Future CMPs," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[5] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the International Symposium on Microarchitecture*, 2009.

[6] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron, "CMP Design Space Exploration Subject to Physical Constraints," in *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 2006.

[7] J. Li and J. F. Martinez, "Power-Performance Implications of Thread-level Parallelism on Chip Multiprocessors," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2005.

[8] B. Rogers, A. Krishna, G. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling," in *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[9] L. Zhao, R. Iyer, S. Makineni, J. Moses, R. Illikkal, and D. Newell, "Performance, Area and Bandwidth Implications on Large-Scale CMP Cache Design," in *Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnect*, 2007.

[10] A. Agarwal, L. Bao, J. Brown, B. Edwards, M. Mattina, C.-C. Miao, C. Ramey, and D. Wentzlaff, "Tile Processor: Embedded Multicore for Networking and Multimedia," in *Proceedings of the Symposium on High Performance Chips*, 2007.

[11] Y. Hoskote, S. Vangal, N. Borkar, and S. Borkar, "Teraflop Prototype Processor with 80 Cores," in *Proceedings of the Symposium on High Performance Chips*, 2007.

[12] C. Ding and T. Chilimbi, "A Composable Model for Analyzing Locality of Multi-threaded Programs," Technical Report MSR-TR-2009-107, Microsoft Research, 2009.

[13] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, "Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors?," in *Proceeding of Compiler Construction*, 2010.

[14] D. L. Schuff, B. S. Parsons, and J. S. Pai, "Multicore-Aware Reuse Distance Analysis," Technical Report TR-ECE-09-08, Purdue University, 2009.

[15] D. L. Schuff, M. Kulkarni, and V. S. Pai, "Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[17] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, 2006.

[18] C. McCurdy and C. Fischer, "Using pin as a memory reference generator for multiprocessor simulation," *ACM SIGARCH Computer Architecture News*, vol. 33, 2005.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[20] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A Benchmark Suite for Data Mining Workloads," in *Proceedings of the International Symposium on Workload Characterization*, 2006.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.

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

[23] M. Zhang and K. Asanovic, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," in *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.

[24] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

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

[26] E. Berg, H. Zeffer, and E. Hagersten, "A Statistical Multiprocessor Cache Model," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2006.

[27] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2005.

[28] G. E. Suh, S. Devadas, and L. Rudolph, "Analytical Cache Models with Applications to Cache Partitioning," in *Proceedings of International Conference on Supercomputing*, 2001.

[29] Y. Zhong, X. Shen, and C. Ding, "Program locality analysis using reuse distance," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 6, 2009.

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

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